Imaginary Traversals and Leftmost Linear Reduction [Draft]

William Blum

September 3, 2017

Abstract

We introduce a method to normalize untyped lambda terms, when the normal-form exists, by combining the theory of traversals, a term-tree traversing techniques inspired from Game Semantics [11, 7], and judicious use of the eta-conversion rule of the lambda calculus.

Little modifications to the traversal theory of the simply-typed lambda calculus [7] is needed. A notable improvement is that eta-long transformation is unnecessary: we traverse the original unmodified term tree. Blindly applying the original traversal rules on untyped terms, which do not have a (finite) eta-long form, causes traversals to get 'stuck' when an application operand is lacking; whereas in a typed setting, eta-long transform ensures that application operands necessarily exist. Relaxing this structural requirement altogether induces the notion of "imaginary" nodes: those are nodes that would exist if the subterm with the missing operand were to be eta-expanded. This allows the traversal to proceed.

We show how, by bounding the non-determinism of the free variable rule, one can effectively compute a subset of traversals that characterizes the set of paths in the beta-normal form of the term. This yields a normalization algorithm for untyped terms with a normal form.

Finally we define *leftmost linear reduction* as a generalization of the head linear reduction of Danos-Regnier and show that the traversals theory implements it, which prove correctness of the normalization procedure.

1 Background

Traversals were originally introduced in the context of higher-order recursion schemes [11] used as generator of order-0 structures such as trees. The notion was then extended to the more general setting of simply-typed languages and, in particular, to the simply-typed lambda calculus [7], where the non-deterministic rule (IVar) is introduced to account for the presence of free variables. The relationship with Game Semantics is well understood in the typed setting: the set of traversals is in bijection with (the plays of) the interaction game denotation of a term, further the *core projection* operation yields a bijection between the standard game denotation and its standard innocent game denotation (Theorem 3.1). This correspondence yields a method to evaluate beta-redexes of a simply-typed lambda term without performing the traditional β-reduction [9, 7, 5, 6].

In this paper we show how the game-semantic traversals of STLC [7] naturally extend to ULC, and thus yield a method to normalize untyped lambda terms. In the absence of types, eta-long expansion is simply not an option: it would yield an infinite term. We thus opt instead for a more reasonable option: *on-demand eta-expansion*. The idea stems from the following observation: attempting to apply the original STLC traversal rules on an untyped lambda term can lead to situations where the traversal gets "stuck". This happens when a sub-term in operator position has an insufficient number of operands to be able to resolve a parameter. For example, STLC traversals would get stuck when trying to resolved the "y argument" of f in term $(\lambda f.f)(\lambda y.y)$. The absence of η-long transform of the term is a notable difference with the traversals for recursion scheme [11] or the simply-typed lambda calculus [7]. This simplifies the presentation as the traversed tree is just a direct representation of the original, unmodified, lambda term.

This adaptation of the game-semantic traversals to the untyped setting has several new ingredients that we will detail in the rest of the paper:

- The 'lambda' and 'variable' rules are augmented to support eta-expansion.
- Eta-expansion is performed 'on-the-fly' at each point where the traditional STLC traversal would normally get stuck: whenever there is insufficiently many operands in an application to continue traversing the term.
- The concept of "ghost" nodes is introduced: those are are imaginary tree nodes that progressively appear as eta-expansion is performed on the sub-terms.
- The term tree itself does not get modified during eta-expansion. Ghost nodes are only added to the traversal and are only "visible" in the context where they get introduced.
- The rule used to model free variables is constrained by a quantity called the *arity threshold*. This quantity, calculated in linear time from the traversal itself, limits the non-deterministic branching factor of the rule. This guarantees that the normalization procedure terminates when the beta-normal form exists.

Normalization can then be implemented by enumerating traversals. Although the set of all traversals can be infinite, we show that for the sake of normalization, it is sufficient to consider sufficiently enough traversals to "cover" all paths of the tree representation of the beta-normal form.

The normalization algorithm presented in this paper was implemented in the HOG tool [4]. The last section of the paper provides some examples of terms normalized with the algorithm.

1.1 Related work

1.1.1 Head linear reduction

In 2004, Danos and Regnier showed the connection between Game Semantics and *head linear reduction*. The various notions of traversals discussed in this paper, including Berezun-Jones, are all essentially methods based on evaluating the head linear reduction sequence of a lambda term. Traversals start by performing a depth-first search to locate the *head occurrence* of the *hoc redex*. They implement linear substitution by "jumping" to the node in the tree that represents the term to be substituted for the head occurrence. At this point the traversal continues as if the head occurrence in the tree was replaced by the subtree representing the argument of the redex. Since all the terms obtained in the head linear reduction sequence are made of sub-terms of the original term, there is no need for environments or closure to evaluate the beta-redex. Instead those can be replaced by some pointer mechanism to capture a given context.

Despite the connection established in this 2004 paper, it is not expliciteley shown how the head linear reduction helps normalize a term. The *Pointer Abstract Machine* introduced in the same paper, for instance, yields what they call "*quasi-head normal form (qhn)*" of the term, not the normal form itself. The resulting *qhn* still needs to be first reduced to head normal form (using head reduction) and then normalized by appealing to the standard λ-calculus theory that tells us that head reduction terminates. In this paper we generalize this notion to *leftmost linear reduction*, a strategy that recrusively applies head linear reduction very much like how the standard reduction recursively applies head reduction. We show how *imaginary traversals* introduced in the present paper effectively implement the leftmost linear reduction, which proves soundness of the traversal normalization procedure.

1.1.2 Traversal of STLC

In [7] we extended the theory of traversals to the simply-typed lambda calculus by introducing the new traversal rule (IVar) to model free variables present in lambda terms. (Such rule is not needed in the original presentation of traversals because higher-order recursion schemes are necessarily closed terms of ground type[11].) We then formalized the correspondence between the theory of traversals and Game semantics by establishing a bijection between traversals and the game denotation of a term. The (fairly technical) proof relies on several key ingredients [7]:

Free variables The introduction of the traversal rule (IVar) modeling free variables of the lambda calculus. (In the original traversal setting, higher-order recursion schemes being necessarily closed and of ground type, such rule is not needed [11]);

Interaction Game Semantics Traversals do not directly correspond to the standard game denotation. Instead they correspond to a more verbose game denotation that preserves all the internal moves played while composing the strategy denoting its subterms, whereas the standard game denotation hides those internal moves;

Traversal core Various operations and transformations on traversals need to be introduced to prove the correspondence. In particular the *projection with respect to the root of the tree* which gives the "core" of a traversal.

This correspondence with Game Semantics yields a method to reduce beta-redexes in simply-typed lambda-terms. This normalization procedure was studied in [7, 5, 4, 12] and implemented in the HOG tool[5, 7]. We recall these results in section 3.4.

The work presented in the present paper can be viewed as a generalization of this work to the untyped case. Proposition 3.3 shows that the two definition coincide for simply-typed terms. A notable difference, however, is that the normalization procedure introduced here produces the beta-normal form of the term rather than its beta-eta-long normal form. Furthermore, we proof soundness without appealing to Game Semantics.

1.1.3 Berezun-Jones traversals

Berezun and Jones introduced the first notion of traversals for the *untyped* lambda calculus (ULC) [3]. Although they took inspiration from the Game Semantic traversals of [11, 5], their definition is more operational in nature and does not directly related to the game denotation of a term. Starting from an operational semantics of the untyped lambda calculus they derive a normalization method that, very much like the traversal of [11, 7], proceeds by traversing some tree representation of the term.

The tree representation used in Berezun-Jones' traversals are direct abstract syntax tree representation of the lambda-term, whereas in the present setting we use a more 'compressed' form where consecutive lambdas and applications are merged into a single node. On the other hand, unlike the original definition of traversals for STLC where eta-long transformation is performed prior to traversing a term, both Berezun-Jones' traversals and the traversals introduced in the present paper require no prior syntactic transformation of the lambda term.

The Berezun-Jones traversals distinctively rely on two justification pointers: each 'token' of the traversal can have one *binding pointer* as well as one *control pointer*. They also involve the use of a boolean parameter (the 'flag') associated with every token of the traversal. In contrast to Berezun-Jones, the traversals presented in this paper necessitate a single justification pointer per node occurrence.

Another notable difference is that the normalization algorithm of Berezun-Jones requires a single traversal of the term. Our normalization algorithm, however, produces one traversal for every branch in the tree representing the beta-normal form. Each branching point corresponds to some occurrence of a variable in the normal form, and each branch corresponds to one operand of the variable. One can conceivably see how such non-deterministic branching could be eliminated by adding appropriate auxiliary pointers to implement backtracking: after the first operand is evaluated, backtrack to the operator variable in the traversal and explore the remaining operands.

1.2 Rest of the paper

In the remaining of this paper we will introduce basic definition from the traversal theory and introduce a new notion of traversals for the untyped lambda calculus. We briefly discuss the correspondence with game models of the Untyped Lambda Calculus [10] and show how they yield an algorithm to normalize untyped lambda terms. Finally, we illustrate the normalization algorithm on some examples of lambda terms.

2 Definitions

2.1 Untyped lambda calculus

We consider the set of terms Λ of the untyped lambda calculus constructed from the following grammar:

$$\Lambda := x \mid (\Lambda\ \Lambda) \mid \lambda x.\Lambda$$

where x ranges over a countable set of variable names.

For conciseness when writing lambda terms, we will abbreviate "consecutive" lambda abstraction $\lambda x_1 \ldots \lambda x_n.U$ for some $n \geq 0$ and term U as just $\lambda x_1 \ldots x_n.U$. If $\overline{x}$ denote the list of variables $x_1 \ldots x_n$ we will just write $\lambda \overline{x}.U$.

2.2 Computation tree and enabling relation

Given an untyped lambda term M we define its ***computation tree*** as the abstract syntax tree (AST) representation of the lambda term where consecutive lambda abstractions are merged into a single node labelled by the list of bound variables and with a single child, and similarly consecutive applications are represented by a single node labelled @ with multiple children: one for the operator and one for each application operand. Formally:

Definition 2.1 (Computation tree of an untyped term)**.** Let M be an untyped lambda term with variable names in $\mathcal{V}$.

- We define the set of labels:
$$\mathcal{L} = \{@\} \cup \mathcal{V} \cup \{\lambda x_1 \ldots x_n \mid x_1, \ldots, x_n \in \mathcal{V}, n \in \mathbb{N}\}$$

- The ***computation tree*** $CT(M)$ is a labelled tree with labels in $\mathcal{L}$ defined inductively on the syntax of M:
$$\begin{array}{rcl} CT(\lambda \overline{x}.z s_1 \ldots s_m) & = & \lambda \overline{x} \ \langle \ \underline{z} \ \langle CT(s_1), \ldots, CT(s_m) \rangle \rangle \\ & & \text{where } m \geq 0,\ z \in \mathcal{V}, \\ CT(\lambda \overline{x}.(\lambda y.t) s_1 \ldots s_m) & = & \lambda \overline{x} \ \langle \ @ \ \langle CT(\lambda y.t), CT(s_1), \ldots, CT(s_m) \rangle \rangle \\ & & \text{where } m \geq 1,\ y \in \mathcal{V}. \end{array}$$
where the expression '$l\langle t_1, \ldots, t_m \rangle$' for $m \geq 0$ denotes a labelled tree with root labelled l and m ordered children trees t_1, ..., t_m. The underlined label z represents a node label for $m > 0$, and a leaf label if $m = 0$.

- We write N to denote the set of nodes of the computation tree. We write N_{var} for the set of variable nodes, N_λ for lambda nodes, and $N_@$ for application nodes.

- For any lambda node $\alpha \in N_\lambda$ we write $\mathsf{Ch}(\alpha)$ to denote its unique child node (either an @ or variable node).

Note that any lambda term can indeed be written in one of the two forms above. In particular, applicative terms are handled by the case $n = 0$ of the form $\lambda.N$ for some term N where 'λ.' is referred to as a "dummy lambda". This compact representation turns out to be useful to maintain alternation between lambda nodes (at odd level, counting from 1 onwards) and variable nodes (at even level).

Observe that the definition is similar to that of STLC [11, 7] with the notable difference that the term does not get eta-long expanded prior to constructing the computation tree.

We define the ***enabling relation*** $\vdash$ between nodes of the computation tree as the relation associating each lambda node to all the variable nodes that it binds, the root of the tree to all the free variable nodes, and every variable and application node to each of their child node. We write $m \vdash_k \alpha$ to indicate that a lambda node α is the kth child of a variable or application node m for some $k \geq 0$; and $\alpha \vdash_k m$ to indicate that the node m is labelled by the kth variable bound by α for $k \geq 1$.

We define ***hereditarily enabling*** as the reflexive transitive closure $\vdash^*$ of the enabling relation, and we say that node m ***hereditarily enables*** node n if $m \vdash^* n$. We write $N^{\vdash^*}$ to denote the set of nodes hereditarily enabled by the root, that is the image of the tree root by the reflexive transitive closure of the enabling relation.

A consequence of the definition of the enabling relation is that every node of the tree is either hereditarily justified by the root or by some application node @. Variable nodes that are hereditarily enabled by the root are called ***external variable nodes***, and variables hereditarily justified by an application node are called ***internal variable nodes***.

We defined the ***arity*** of a node as follows: the arity of a lambda node $\lambda x_1 \cdots x_k$ for $k \geq 0$ is denoted by $|\lambda x_1 \cdots x_k|$ and is defined as k; the arity of a variable node x, denoted $|x|$ is the number of children of x in the computation tree; the arity of a @ node is the number of its children nodes minus 1 (*i.e.*, the number of operands in the application).

2.3 Ghost nodes

We extend the concept of computation tree nodes by introducing two infinite classes of "imaginary" nodes called ***ghost nodes***: one class of "ghost variable nodes", denoted θ, and one class of "ghost lambda nodes", denoted $\lambda\!\!\lambda$. By abuse of notation we will use θ and $\lambda\!\!\lambda$ to refer to individual elements of the two respective classes. In the upcoming definition of traversals, ghost nodes will be used as placeholders representing lambda nodes and variable nodes from eta-expanded subterms.

We simultaneously define ghost nodes and extend the enabling relation $\vdash$ for them by induction: for every (ghost) variable or application node m and for all $k > |m|$ there is a ghost lambda node $\lambda\!\!\lambda$ such that $m \vdash_k \lambda\!\!\lambda$; for all (ghost) lambda node α and all $k > |\alpha|$ there is a ghost variable node θ such that $\alpha \vdash_k \theta$. Thus, each ghost variable and ghost lambda node is uniquely defined by its enabler node (possibly itself a ghost node) and associated label $k \geq 1$.

We refer to nodes N of the computation tree as ***structural nodes*** and $\tilde{N}$ for the extended set $N + \theta + \lambda\!\!\lambda$ where θ and $\lambda\!\!\lambda$ denote the sets of ghost nodes uniquely determined by N together with the enabling relation $\vdash$. Nodes in $\lambda\!\!\lambda$ and θ are called ***ghost nodes***. We write $N_\lambda^{\lambda\!\!\lambda}$ as a shorthand for $N_\lambda + \lambda\!\!\lambda$ and N_{var}^θ as a shorthand for $N_{\mathsf{var}} + \theta$.

By convention ghost variables and lambda nodes are assigned arity 0.

2.4 Justified sequence of nodes

A ***justified sequence*** is a sequence of nodes from the computation tree where every node occurrence n in the sequence, except the first one, has an associated link (the "justification pointer") pointing to some previous node occurrence j in the sequence (called its "justifier") with an associated "link label" $l \geq 0$, such as the justifier node enables the node pointing to it by the enabling relation $\vdash$ with the corresponding label (*i.e.*, $j \vdash_l n$). Such link is represented as follows:

$$\begin{tikzcd}[column sep=small] s = \cdots & j & \ldots & n \arrow[ll, bend right] \end{tikzcd}$$

Whereas the enabling relation is *statically* induced by the structure of the tree, the justification relation is defined on *occurrences* of nodes for a specific justified sequence.

We define the set of justified sequences of nodes over M, written $\mathcal{J}(M)$, as the justified sequences over $\tilde{N}$ (the nodes from the computation tree augmented with ghost nodes).

It is sometimes convenient to represent a justification sequence using the ***triplet encoding***: a sequence of triplet in $\tilde{N} \times \mathbb{N} \times \mathbb{N}$ where the first component gives the node with label, the second component encodes the distance between a node and its justifier in the sequence (0 for no pointer), and the third is the link label.

The ***structure*** of a justified sequence is defined as the sequence of triples $\{@, V, \lambda\}\mathbb{N} \times \mathbb{N}$ where the first component indicates the type of each occurrence (@-node, variable or lambda node) and the last two encodes the justification pointers as in the triplet encoding.

Two justification sequences s and u over M are equal, written $s = u$, if their triplet encoding are the same: their underlying sequences of nodes are identical (same node) and the associated justification pointers are identical (same justifier and same pointer label). Two sequences s and u over two distinct terms M and N are ***equivalent***, written $s \equiv u$ just if they have the same structure: the types of the underlying nodes in the two sequences match each other and have the same justification pointers.

Example 2.1. We have $\begin{tikzcd}[column sep=tiny] \lambda xy & \cdot & x \arrow[ll, bend right] \end{tikzcd} \equiv \begin{tikzcd}[column sep=tiny] \lambda zy & \cdot & z \arrow[ll, bend right] \end{tikzcd}$. But we also have $\begin{tikzcd}[column sep=tiny] \lambda xyzwt & \cdot & x \arrow[ll, bend right] \end{tikzcd} \equiv \begin{tikzcd}[column sep=tiny] \lambda x & \cdot & x \arrow[ll, bend right] \end{tikzcd}$ since they have same structure $(\lambda, 0) \cdot (V, 1)$.

We write $I \cong J$ to denote two subsets $I \subseteq \mathcal{J}(M)$ and $J \subseteq \mathcal{J}(N)$ such that there exists an isomorphism (*i.e.*a structure-preserving bijection) between I and J. That is there exists a bijection $\phi : I \longrightarrow J$ such that for any $j \in J$, $j \equiv \phi(j)$.

A justified sequence verifies the ***alternation condition*** if the first node is a lambda node and subsequent nodes occurrences alternate between (i) a variable or application node (ii) a lambda node.

We say that an occurrence of a node in a justified sequence is ***hereditarily justified by some other occurrence*** if recursively following justification pointers starting from it leads to that other occurrence in the sequence. Because justification pointers must honor the enabling relation $\vdash$ induced by the term structure, if a node occurrence n is hereditarily justified by some occurrence of a node $m \in N$ then n is necessarily hereditarily enabled by m. Further if m occurs only once in the justified sequence then the occurrences hereditarily justified by m are precisely the occurrences of nodes that are hereditarily *enabled* by m.

For any justified sequence t we write t^{ω} to denote the last occurrence in t. The notion of sequence prefix naturally extends to justified sequences. For any occurrence n in t we write $t_{\leq n}$ for the prefix subsequence of t ending at n, and $t_{<n}$ for the prefix ending at the occurrence immediately preceding n (or the empty sequence if n is the first occurrence in t). We say that t is an ***extension*** of justified sequence u if u is a strict prefix of t sharing the same justification pointers.

We use the standard operations borrowed from Game Semantics on justified sequences [1].

Definition 2.2 (Projection)**.** Let s be a justified sequence of nodes.

- Let n be a node occurring in s we write $s \upharpoonright n$ to denote the subsequence of s obtained by keeping only nodes that are hereditarily justified by n in s. We call it the ***projection of*** s with respect to n;
- Let A be a subset of nodes in N, we write $s \upharpoonright A$ to denote the subsequence of s obtained by keeping only nodes that are in A;
- We will consider the subsequence $s \upharpoonright N^{\vdash^*}$ of external nodes (*i.e.*, nodes hereditarily enabled by the root). Observe that if $r_1, \ldots r_n$, $n \geq 1$ are the occurrences of the root in s then it is also given by the projection of s with respect to those occurrences: $s \upharpoonright N^{\vdash^*} = s \upharpoonright r_1 \upharpoonright \ldots \upharpoonright r_n$.

The P-view of a justified sequence is the sub-sequence obtained by reading the sequence backwards and following the justification pointer every other node: (i) if the node being read is a variable node then keep that node and follow its justification pointer (*i.e.*, skip all the occurrences "underneath that pointer") (ii) if the node is a lambda node then keep that node and move to the preceding node. Formally:

Definition 2.3 (Views)**.** The P-view of a justified sequence s, denoted $\ulcorner s \urcorner$ is defined recursively as follows

$$\begin{array}{rcll}
\ulcorner \epsilon \urcorner & = & \epsilon & \\
\ulcorner s \cdot n \urcorner & = & \ulcorner s \urcorner \cdot n & \text{if } n \text{ is a variable or @ node;} \\
\ulcorner s \cdot \overset{\curvearrowleft}{m \ldots n} \urcorner & = & \ulcorner s \urcorner \cdot \overset{\curvearrowleft}{m \cdot n} & \text{if } n \text{ is a lambda node;} \\
\ulcorner s \cdot n \urcorner & = & n & \text{if } n \text{ is a lambda node with no pointer.}
\end{array}$$

The O-view, denoted $\llcorner s \lrcorner$, is defined as the dual of the P-view:

$$\begin{array}{rcll} \llcorner \epsilon \lrcorner & = & \epsilon & \\ \llcorner s \cdot n \lrcorner & = & \llcorner s \lrcorner \cdot n & \text{if } n \text{ is an } \lambda\text{-node;} \end{array}$$

$$\begin{tikzcd}[column sep=tiny] \llcorner s \cdot & m & \cdot \ldots \cdot & n \lrcorner \arrow[ll, bend right] & = & \llcorner s \lrcorner \cdot & m & \cdot & n \arrow[ll, bend right] & \text{if } n \text{ is a variable node;} \end{tikzcd}$$

$$\begin{array}{rcll} \llcorner s \cdot n \lrcorner & = & n & \text{if } n \text{ is an } @ \text{ node.} \end{array}$$

Given two node occurrences occurrence n and m in a justified sequence s, we say that n is ***visible at*** m just if n occurs in the P-view $\ulcorner s_{\leq m} \urcorner$.

Justified paths of the term tree We will consider paths of a term tree as a set of justified sequences. For any term M we define the ***set of justified paths*** $\mathcal{P}aths(M)$ as the set of justified sequences in $\mathcal{J}(M)$ whose underlying sequences of nodes are paths in the labelled-tree $CT(M)$ with associated justification pointers induced by the enabling relation: occurrence of bound variable nodes are justified by their binder node with link label determined by the variable index in the binding node; free variables are justified by the root (the first node in the sequence) with label index determine by the free variable index; and lambda nodes are justified by their parent (the immediate predecessor in the sequence) with label index given by the child index.

This notion is well defined because a variable binder necessarily occurs in the path from it to the root.

Property 2.1 (Path characterization)**.** (i) An untyped term M is uniquely determined by the subset of maximal justified paths of $\mathcal{P}aths$. (ii) Further M is uniquely determined, up to α-conversion, by the *structure* (*i.e.*, node types and pointers) of the maximal justified paths in $\mathcal{P}aths$.

Proof. (i) follows from the fact that computation trees are in one-to-one correspondence with the standard tree representation of a lambda term, and because $\mathcal{P}aths(M)$ is prefix-closed it's uniquely determined by its maximal elements. (ii) is due to the fact that variables names are uniquely determined by the justification pointers and their associated label index. $\square$

Example 2.2. $\mathcal{P}aths((\lambda x.xy)(\lambda z.z))$ is the prefix closure of

$$\begin{tikzcd}[column sep=tiny] \{ & \lambda & @ & \lambda x \arrow[l, bend right, "0"'] & x \arrow[l, bend right, "1"'] & \lambda \arrow[l, bend right, "1"'] & y, \arrow[lllll, bend right, "1"'] & \lambda & @ & \lambda z \arrow[l, bend right, "1"'] & z \arrow[l, bend right, "1"'] & \}. \end{tikzcd}$$

3 Traversals

3.1 Definition and properties

Definition 3.1 (ULC traversals)**.** The set of ***traversals*** of an untyped lambda term M, denoted $\mathcal{T}rav(M)$, abbreviated $\mathcal{T}rav$ when the term is clear from context, is the set of justified sequences of nodes over M recursively defined by the rules of Table 1.

A traversal that does not have any extension is a ***maximal traversal***.

Some immediate property that can be shown by induction on the rules:

Property 3.1. For any traversal t

1. t verifies the alternation condition;
2. $t \upharpoonright N^{\vdash^*}$ is a valid justified sequence (with respect to the enabling relation $\vdash$) verifying the alternation condition.

PROGRAM – Structural rules

- (Root) The singleton sequence r is in $\mathcal{T}rav$ where r is the root of the tree.

- (App) If $t \cdot @$ is a traversal then so is $\begin{tikzcd} t \cdot @ & \cdot & \alpha \arrow[ll, bend right, "0"'] \end{tikzcd}$ where $\alpha \in N_\lambda$ is @'s 0th child λ-node.

- (Lam) If $t \cdot \alpha$ is a traversal where $\alpha \in N_\lambda$ then so is$t \cdot \alpha \cdot n$ where n denotes α's unique child. Furthermore:
  - ($\mathsf{Lam}^{@}$) If n is an @-node then it has no justifier,
  - ($\mathsf{Lam}^{\mathsf{var}}$) If n is a free variable node then it points to the only occurrence of the root in $\ulcorner t \cdot \alpha \urcorner$. If n is a bound variable then it points to the only occurrence of its binder in $\ulcorner t \cdot \alpha \urcorner$.

- ($\mathsf{Lam}^{\lambda\!\!\lambda}$) If $\begin{tikzcd} t \cdot \alpha \cdot & n & \cdot \ldots \cdot & \lambda\!\!\lambda \arrow[ll, bend right, "i"'] \end{tikzcd} \in \mathcal{T}rav$ for some prefix t, $\alpha \in N_\lambda^{\lambda\!\!\lambda}$ and $n \in N_{\mathsf{var}}^{\theta}$ then

$$\begin{tikzcd} t \cdot & \alpha & \cdot & n & \cdot \ldots \cdot & \lambda\!\!\lambda \arrow[ll, bend right, "i"'] & \cdot & \theta \arrow[llllll, bend right, "|\alpha| + i - |n|"'] & \in \mathcal{T}rav \end{tikzcd}$$

PROGRAM – Copy-cat rules

- (Var) If $\begin{tikzcd} t \cdot m \cdot & \alpha & \ldots & n \arrow[ll, bend right, "i"'] \end{tikzcd} \in \mathcal{T}rav$ for $i > 0$, $n \in N_{\mathsf{var}}^{\theta}$ hereditarily justified by an @-node; $m \in N_{\mathsf{var}}^{\theta} \cup N_@$; and $\alpha \in N_\lambda^{\lambda\!\!\lambda}$ then:

$$\begin{tikzcd} t \cdot & m & \cdot & \alpha & \ldots & n \arrow[ll, bend right, "i"'] & \cdot & \beta \arrow[llllll, bend right, "i"'] & \in \mathcal{T}rav \end{tikzcd}$$

Concrete $i \leq |m|$: $\beta \in N_\lambda$ is the ith child of m;

Eta-expanded $i > |m|$: β is a ghost lambda node $\lambda\!\!\lambda$.

DATA – Input-variable rules

- (IVar) If $t \cdot n$ is a traversal where $n \in N_{\mathsf{var}}^{\theta}$ is hereditarily justified by the root. For every node $m \in N_{\mathsf{var}}^{\theta}$ occurring in $\llcorner t \cdot n \lrcorner$ and every $i \geq 1$ we have $t \cdot n \cdot \alpha \in \mathcal{T}rav$ with α pointing to m with label i, where:

Concrete $i \leq |m|$: $\alpha \in N_\lambda$ is the ith child of m;

Eta-expanded $i > |m|$: α is a ghost lambda node $\lambda\!\!\lambda$.

Table 1: Imaginary traversals $\mathcal{T}rav$ of the untyped lambda calculus

On-the-fly expansion The following definition explicit the notions of *on-the-fly eta-expansion* implemented by the traversal rules (Var) and (IVar):

Definition 3.2 (On-the-fly eta-expansion)**.** Let M be an untyped term.

- Let n be a node of the tree of M and N denote the subterm of M rooted at n. We write $ETA(M, n)$ to denote the term obtained by substituting the subterm N in M with the term $\lambda\theta.N\theta$ for some variable θ fresh in N.
- Let M be an untyped term and t be a *finite* traversal of M. We define the ***eta-expansion of** M **with respect to** t*, written M^t, as a term obtained by eta-expanding its subterms according to the rules used to traverse M. It is defined by induction on the traversal rules: By convention we define $M^\epsilon = M$. Suppose $t = u \cdot n$ for some node n. Let m denote the justifier of n if it exists and i denote its link label. Consider the last rule used to traverse t:

$$M^{u\cdot n} = \begin{cases} (\mathsf{Root}) & M^u \\ (\mathsf{App}) & M^u \\ (\mathsf{Lam}) & M^u \\ (\mathsf{Lam}^{\lambda\!\!\lambda}) & M^u \\ (\mathsf{Var}) & M^u & \text{if } i \leq |m|; \\ & ETA(M^u, m) & \text{if } i > |m|; \\ (\mathsf{IVar}) & M^u & \text{if } i \leq |m|; \\ & ETA(M^u, m) & \text{if } i > |m|. \end{cases} \tag{1}$$

Observe that the tree obtained after eta-expansion with respect to t contains the tree of M itself, therefore paths in the tree of M are also paths in the tree of M^t.

Example 3.1. Take $M = (\lambda u.u\ (y_1\ u))(\lambda v.v\ y_2)$. Its computation tree is shown in Ex. 4.3. A valid traversal is

$$t = \begin{tikzcd}[column sep=small] \lambda & @ \arrow[l, bend right] & \lambda u \arrow[l, bend right] & u \arrow[l, bend right] & \lambda v \arrow[lll, bend right] & v \arrow[l, bend right] & \lambda \arrow[lll, bend right] & y_1 \arrow[lllllll, bend right] & \lambda \arrow[l, bend right] & u \arrow[lllllll, bend right] & \lambda v \arrow[lllllllll, bend right] & v \arrow[l, bend right] & \lambda\!\!\lambda^1 \arrow[llll, bend right] & \theta^1 \arrow[llll, bend right] & \lambda\!\!\lambda^1 \arrow[l, bend right] & \theta^1 \arrow[lll, bend right] & \lambda \arrow[lllll, bend right] & y_2 \arrow[lllllllllllllllll, bend right] \end{tikzcd}$$

. The eta expansion of M with respect to t is $M^t = (\lambda u.u\ (y_1\ (\lambda\alpha.u(\lambda\beta.\alpha\beta)))(\lambda v.v\ y_2)$.

Path-View correspondence We now generalize a known result from the theory of traversals for higher-order grammars [11] and simply-typed lambda terms [7, Proposition 4.29] to the untyped setting: The game-semantic concept of 'Proponent view' from definition 2.3 corresponds to the concept of 'tree path' in the following sense:

Proposition 3.1 (Path-View correspondence for ULC)**.** Let M be an untyped term and t be a *finite* traversal of M then $\ulcorner t \urcorner$ is a path (with associated justification pointers) in the computation tree of the eta-expansion of M with respect to t. In particular, if t does not contain any ghost node then $\ulcorner t \urcorner$ is a path in the computation tree of M.

Proof. Proven by induction on the traversal rules of t. □

Property 3.2. The traversal rules are well-defined.

Proof. (i) Rule (Lam): By Proposition 3.1, the P-view is a path in the tree of the eta-expansion of M with respect to t. But the last node of t is a structural node therefore, since the eta-expanded tree contains the tree of M, this path is necessarily also the path to t^ω in the tree of M (and thus $\ulcorner t \urcorner$ does not contain any ghost node, even though t itself may contain ghost nodes). Consequently, if the last node is a variable node, its enabler necessarily occurs exactly once in the P-view.

(ii) Rule (Var): In the concrete sub-case, m is necessarily itself a structural node since $|m| \geq i > 0$, and its ith child exists in N since m as arity greater than i. □

Traversal core We now generalize the notion of *traversal core* from [7] to the imaginary traversals. In the typed setting, the core of a traversal is the subsequence consisting of external nodes. In the untyped setting it is obtained by taking the subsequence of external nodes followed by a relabelling operation on lambda nodes.

We will consider sequence of variable names in $\mathcal{V}^*$ as stack. For any sequence of variables $\overline{x} = x_1 \ldots x_n$, we define the stack pop operation pop_j, $n, j \geq 0$ that removes the first j elements of the sequence: $pop_j(x_1 \ldots x_n) = x_{j+1} \ldots x_n$ if $j < n$, and $pop_j(x_1 \ldots x_n) = \epsilon$ otherwise. And for any sequence $\overline{y} = y_1 \ldots y_m$, $m \geq 0$ we write $\overline{xy}$ for $x_1 \ldots x_n y_1 \ldots y_n$, the stack obtained after pushing the variables $\overline{x}$ onto $\overline{y}$.

Definition 3.3 (Core projection)**.** We first define the partial function $\pi\colon \mathcal{T}rav(M) \longrightarrow \mathcal{J}(M)$ for traversals that do not contain any ghost occurrences then extend the notion to all traversals.

- For any stack $\overline{y} \in \mathcal{V}^*$ of variable names and traversal $t \in \mathcal{T}rav(M)$ we define $\pi(t)$ by induction on t:

$$\begin{aligned} \pi_{\overline{y}}\colon \mathcal{T}rav(M) &\longrightarrow \mathcal{J}(M) & \\ t \cdot x &\longmapsto \pi_\epsilon(t) \cdot x & \text{if } x \in N^{\vdash^*} \\ t \cdot x &\longmapsto \pi_{\overline{y}}(t) & \text{if } x \notin N^{\vdash^*} \\ t \cdot @ &\longmapsto \pi_{pop_k(\overline{y})}(t) & \text{where } k = |@| \\ t \cdot \lambda\overline{x} &\longmapsto \pi_\epsilon(t) \cdot \lambda\overline{xy} & \text{if } \lambda\overline{x} \in N^{\vdash^*} \\ t \cdot \lambda\overline{x} &\longmapsto \pi_{\overline{xy}}(t) & \text{if } \lambda\overline{x} \notin N^{\vdash^*} . \end{aligned}$$

- We extend π to all traversals as follows:

$$\begin{aligned} \pi_{\overline{y}}\colon \mathcal{T}rav(M) &\longrightarrow \mathcal{J}^\eta(M) \\ t &\longmapsto \pi_{\overline{y}}(\eta(t)) \end{aligned}$$

where $\mathcal{J}^\eta(M)$ be the union of the sets of justified sequences of nodes over eta-expansions of M with respect to all possible traversals:

$$\mathcal{J}^\eta(M) = \bigcup_{t \in \mathcal{T}rav(M)} \mathcal{J}(M^t) .$$

We call $\pi_\epsilon(t)$ the ***core*** of t which we abbreviate as $\pi(t)$. We define the ***core P-view*** as $\ulcorner \pi(t) \urcorner$, the P-view of the traversal core.

The stack parameter $\overline{y}$ from the above definition represents the sequence of abstractions $\lambda\overline{y}$ and is therefore called the ***stack of pending lambdas***. Thus, in words, the *core projection* is given by the subsequence of external nodes where each lambda node label is suffixed with the *pending lambdas* of the maximal consecutive sequence of internal nodes following it.

Proposition 3.2. Let M be an untyped term and t be a *finite* traversal of M. Let N and N_t denote the nodes of tree $CT(M)$ and $CT(M^t)$ respectively. Then there exists a one-to-one mapping:

$$\eta_t : \tilde{N} \longrightarrow \tilde{N}_t$$

with implicit *element-wise pointer-preserving* extension to justified sequences $\eta_t : \mathcal{J}(M) \longrightarrow \mathcal{J}(M^t)$ such that:

(i) For any $u \in \mathcal{T}rav(M)$, $\eta_t(u)$ is a traversal of M^t.

(ii) If v is a *finite* traversal of M^t that does not contain any ghost node then there exists a traversal u of M (with possibly ghost nodes) such that $v = \eta_t(u)$.

(iii) The restriction of η_t to $\mathcal{T}rav(M)$ defines a bijection with $\mathcal{T}rav(M^t)$. Further the bijection is strong in the sense that the traversal structure (node type and pointers, but no necessarily labels) are preserved.

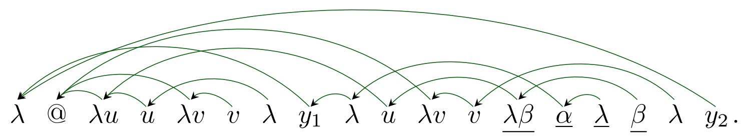
Proof. The tree $CT(M)$ being by definition a subtree of $CT(M^t)$, it induces a one-to-one mapping from $N(M)$ to $N(M^t)$. We extend it to ghost nodes by mapping ghost occurrences in t to the corresponding node resulting from the eta-expansions of subterms of M in M^t, and all other ghost nodes not occurring in t to the corresponding ghost nodes in $CT(M^t)$. The map η_t can be formally defined by induction on the traversal t, observing that in the variable rules (Var) and (IVar), the on-the-fly eta-expansion of Definition 3.2 increments the arity of the justifier node m, therefore the corresponding structural node must exist in M^t.

(i) By induction on u. Reusing the rule used to traverse u in order to traverse v. For the case (Var) and (IVar), the *concrete* sub-case is used instead of the *eta-expanded* subcase when traversing a ghost variable that also appeared in t.

(ii) By induction on v, applying to u each rule that is applied to v. For the case (Var) and (IVar), if the index i is greater than the arity of the node m in M then the lambda node from v gets replaced by a ghost lambda node in u.

(iii) By (i) the function is well defined, it is injective by definition and by (ii) it is surjective. $\square$

Example 3.2. Continuing with the example above: the following traversal obtained from t is a valid traversal of M^t where the occurrences corresponding to ghosts in t are underlined:


$\lambda \ @ \ \lambda u \ u \ \lambda v \ v \ \lambda \ y_1 \ \lambda \ u \ \lambda v \ v \ \underline{\lambda\beta} \ \underline{\alpha} \ \underline{\lambda} \ \underline{\beta} \ \lambda \ y_2 .$

3.2 Imaginary traversals subsume STLC traversals

Traversals for simply-typed languages were previously studied in [7]. The traversal rules defined in Table 1 closely match those of the simply-typed lambda calculus and PCF from [7] with some important differences:

No interpreted constants Unlike PCF, there are no interpreted constants in the present setting therefore the rule (Value) and ($\mathsf{InputValue}$) from the original presentation are not needed.

No η-long expansion In the original STLC traversals, the term is eta-long expanded prior to calculating the set of traversals. This guarantees that the operand of an application always exists in the tree. Imaginary traversals, on the other hand, are defined on the unmodified tree representation of the term.

On-the-fly η-expansion In the untyped setting, eta-long expansion is an infinite process, so instead of eta-long expanding the term prior to traversing it, imaginary traversals perform eta-expansion 'on the fly'. Eta-expansion occurs in rule (Var) when the arity of a variable in operand position is too low to statically determine the operand of an application (case $i > |n|$). When the variable arity is high enough ($i \geq |n|$), the definition of the rule coincides with STLC and the static tree representation of the term dictates the next node to visit.

Free variables Eta-expansion can also occur on free variables, so like for rule (Var), the input-variable rule allows for infinitely many eta-expansions ($k > 0$).

Traversl core In the typed setting, the core is obtained by just filtering nodes with respect to the tree root. In the untyped settings, additionally, lambda nodes are relabelled.

Traversing ghost nodes There is an additional rule ($\mathsf{Lam}^{\lambda\!\!\lambda}$) for the case where a traversal ends with a ghost lambda node. In the concrete sub-case, rule (Lam) visits the unique child node of the last lambda node in the traversal. In the eta-expanded sub-case where the last node is a ghost lambda node, there is no such child node, so we visit an imaginary one: the variable node that would be created if we were to eta-expand the sub-term under the lambda.

The ghost placeholder θ thus represents an occurrence of the jth variable that would be bound by lambda node α if the sub-term at node α were eta-expanded $i - |n|$ times: hence $j = |\alpha| + i - |n|$. (Observe that in this case we necessarily have $i > |n|$ since the ith child of n is a ghost variable node.)

Let's fix a simply-typed term-in-context $\Gamma \vdash M : T$, with typed-context Γ (a set of typed variables), and simple type T. Its ***eta-long form*** is defined inductively on the type T and is obtained by recursively eta-expanding every subterm as many times as possible with respect to the type of the subterm [11, 7]. By abuse of language we will say that an untyped term M is in eta-long form if it inhabits a simple type T such that the eta-long expansion of it with respect to T is M itself.

The rules of the STLC traversals from [7] consist of a subset of the imaginary traversal rules of Table 1.

Definition 3.4 (STLC traversal [7])**.** Given a simply-typed term in context $\Gamma \vdash M : T$ we write $\mathcal{T}rav_{\mathsf{STLC}}(\Gamma \vdash M : T)$ to denote the set of justified sequences of nodes from the tree of the *eta-long form* of M obtained with the rules of Table 1 with the exclusion of ($\mathsf{Lam}^{\lambda\!\!\lambda}$) and without the 'Eta-expanded' sub-cases of rule (Var) and (IVar).

The above definition is well-defined:

Proposition 3.3 (ULC and STLC traversals coincide)**.** Let M be an untyped term that inhabits the simple type T, that is for some context Γ we have $\Gamma \vdash M : T$. Let $\eta_{\mathsf{lf}}(M)$ denote the *eta-long normal form* with respect to T:

(i) Traversals in $\mathcal{T}rav(\eta_{\mathsf{lf}}(M))$ do not contain any ghost node.

(ii) $\mathcal{T}rav_{\mathsf{STLC}}(\Gamma \vdash M : T) = \mathcal{T}rav(\eta_{\mathsf{lf}}(M))$.

Proof. (i) Because the term is eta-long expanded, the condition $i > |m|$ in the rules (Var) and (lVar) never holds while traversing the term. This is shown by induction on the traversal rules, observing that the arity of an @ node is necessarily equal to the arity of the its 0th child lambda node and that the arity of a variable node with binding index i is necessarily equal to the arity of the ith child lambda node of its binder's parent.

(ii) By definition of $\mathcal{T}rav_{\mathsf{STLC}}$, the remaining traversals rules of STLC and ULC coincide, therefore the equality holds. □

3.3 Property of ghost nodes

It is helpful to think of ghost nodes as the counterpart of complex numbers sometimes used in mathematics to prove trigonometry identities: they are introduced intermediately to perform some computation or calculation (e.g. using De Moivre's Theorem) but do not appear in the final result. Just like the imaginary number i is created out of the impossibility of calculating the square root of -1, ghost nodes are defined from the impossibility of "traversing" a beta-redex of a lambda term due to an insufficient number of operands.

Ghost nodes appear in a traversal when the arity of a node is too low to continue a structural traversal of the tree:

Property 3.3. If $\begin{tikzcd} x & \cdots & y \arrow[ll, bend right, "i"'] \end{tikzcd} \in \mathcal{T}rav$ then $y \in \theta \cup \lambda\!\!\lambda \iff i > |x|$. In particular for all $n \in N_{\mathsf{var}}$ and $\alpha \in N_{\lambda}$:

1. $\begin{tikzcd} n & \cdots & \lambda\!\!\lambda \arrow[ll, bend right, "i"'] \end{tikzcd} \in \mathcal{T}rav \implies i > |n|$

2. $\begin{tikzcd} \alpha & \cdots & \theta \arrow[ll, bend right, "i"'] \end{tikzcd} \in \mathcal{T}rav \implies i > |n|$.

Definition 3.5. We call ***ghost materialization*** any application of a traversal rule where the last occurrence in the traversal prior to applying the rule is a ghost node (θ or $\lambda\!\!\lambda$), and the node traversed after applying the rule is a structural node of the tree (in N).

Remark 3.1 ((Var) materialization)**.** Observe that among all the rules defined in Table 1, the rule (Var) is the only rule that can materialize a structural node in N from a traversal ending with a ghost node. This means that after traversing ghost nodes, the only way to 'come back' to structural nodes is to visit a ghost variable node θ with an application of rule (Var) of the following form:

$$(\mathsf{Var}) \quad \begin{tikzcd} t \cdot \beta \cdot & y \cdot & \alpha \ldots & \theta \cdot \arrow[l, bend right, "i"'] & \lambda\overline{x} \arrow[lll, bend right, "i"'] \end{tikzcd} \in \mathcal{T}rav$$

where

- $y \in N_{\mathsf{var}}$,
- $\lambda\overline{x} \in N_\lambda$ is the ith child lambda node of $y \in N_{\mathsf{var}}$,
- α is either a structural lambda node in N_λ or a ghost lambda node in $\lambda\!\!\lambda$,
- $0 \leq \alpha < i \leq |y|$.

3.4 Correspondence with Game semantics

In [7] we formalized the correspondence between the theory of traversals and Game Semantics in the setting of simply-typed languages: there is a bijection between the set of traversals of M and the revealed interaction game denotation of M. Furthermore, the 'core projection' yields a bijection with the standard innocent game denotation of M. In other words, the traversal cores are precisely plays from the game denotation of the term. Formally:

Theorem 3.1 (Traversal-Play Correspondence for STLC (Theorem 4.96 in [7]). The following two bijections hold for every simply-typed term $\Gamma \vdash M : T$:

$$\begin{array}{rcl} \mathcal{T}rav(\eta_{\mathsf{lf}}(M)) & \cong & \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle \\ \mathcal{T}rav^{\pi}(\eta_{\mathsf{lf}}(M)) & \cong & [\![\Gamma \vdash M : T]\!] \ . \end{array}$$

where $\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$ and $[\![\Gamma \vdash M : T]\!]$ denote respectively the *revealed game denotation* and *innocent game denotation* of $\Gamma \vdash M : T$; and $\mathcal{T}rav^{\pi}$ denotes the image of $\mathcal{T}rav$ by π (*i.e.*, set of justified sequences that are *core* of some traversal of M).

Game semantics Correspondence for ULC What would be the equivalent of Theorem 3.1 in the untyped case? In his thesis, Andrew Ker defined and studied Game models for the untyped lambda calculus [10]. We conjecture that the traversal-game semantics isomorphism for STLC also yields between the ULC using traversals from Table 1 and the game model of ULC from Ker's thesis: that there is an isomorphism between the set of imaginary traversals and the revealed game semantics of the ULC term, and further, an isomorphism between the standard game semantics and the set of traversal cores:

Conjecture 3.1 (Traversal-Play Correspondence for ULC). For every untyped lambda-term M we have the two bijections:

$$\begin{array}{rcl} \mathcal{T}rav(M) & \cong & \langle\!\langle M \rangle\!\rangle \\ \mathcal{T}rav^{\pi}(M) & \cong & [\![M]\!] \end{array}$$

where $[\![M]\!]$ denotes the innocent *effectively almost everywhere copycat(EAC)* game denotation of M defined in [10], $\langle\!\langle M \rangle\!\rangle$ denotes the corresponding interaction game denotation where internal moves are not hidden during strategy composition, and $\mathcal{T}rav^{\pi}(M)$ denotes the set of traversal cores of M.

Proof idea. The argument should follow the same structure as for STLC [7] but using the Ker's game model of ULC instead of the innocent game model of STLC. The "on-the-fly" eta-expansion of imaginary traversals relates to the $Fun : U \to (U \Rightarrow U)$ morphism of Ker's game category where U denotes the maximal arena. □

4 Normalizing with Traversals

We will now show how traversals can be used to normalize lambda terms. The crux of it lies in the *Path Characterization* result (Proposition 4.2 and Theorem 4.2)) which states that the set of traversals *core P-views* captures what is strictly required from traversals in order to reconstruct the normal form of a term. For STLC, the characterization result follows from the Game Semantics correspondence; we prove its untyped counterpart using a term-rewriting argument.

This characterization result suggests a normalization method based on enumerating the entire set of traversals. The caveat is that the set of traversals may be infinite! This is because the non-determinism of the free variable rule can give rise to arbitrarily long traversals.

Fortunately, not all traversals need to be enumerated: it is sufficient to enumerate a subset of traversals that covers all the core P-views. Some traversals are 'redundant' in the sense that there exist shorter traversals with identical core P-view. We capture this notion by introducing equivalence classes on traversals: two traversals are in the same class just if they have the same core P-view. It then suffices to exhibit a traversal subset that is *complete* for the equivalence classes, in the sense that it contains at least one traversal for each equivalence class. If such subset is effectively computable and finite then it yields a normalization procedure.

We first show how this can be done first for the simply-typed lambda calculus. We introduce the subset of *branching traversals* which verifies this property. For simply typed terms in eta-long form this subset is finite. This yields a normalization procedure for STLC. Soundness follows from the characterization theorem which is an immediate consequence of the Game Semantics correspondence of [7].

We then introduce the subset of *normalizing traversals* which yields a similar normalization procedure for ULC to calculate β-normal forms of untyped lambda terms when they exist.

4.1 Quotienting

We introduce a quotient relation that eliminates 'redundant' traversals that have identical core P-views:

Definition 4.1 (Quotienting)**.** We define the core P-view function as the composition of $\ulcorner\_\urcorner : \mathcal{J}^\eta \longrightarrow \mathcal{J}^\eta$ with the core projection $\pi : \mathcal{T}rav \longrightarrow \mathcal{J}^\eta$:

$$\begin{aligned}\ulcorner\pi(\_)\urcorner : \mathcal{T}rav &\longrightarrow \mathcal{J}^\eta\\ t &\longmapsto \ulcorner\pi(t)\urcorner\end{aligned}$$

We define $\sim$ as the equivalence relation over $\mathcal{T}rav$ induced by $\ulcorner\pi(\_)\urcorner$ up to relabelling (same structure but not necessarily the same labels). Formally:

$$t \sim u \quad \text{iff} \quad \ulcorner\pi(t)\urcorner \equiv \ulcorner\pi(u)\urcorner.$$

We write $\mathcal{T}rav/{\sim}$ for the set of equivalence classes of $\mathcal{T}rav$. We identify a $\sim$-equivalence class with the structure of the P-view core of the traversals it contains.

A subset $T \subseteq \mathcal{T}rav$ is ***$\sim$-complete*** if it contains at least one element for each $\sim$-equivalence class; that is if $T/{\sim} = \mathcal{T}rav/{\sim}$.

In the next section we will explore a $\sim$-complete traversal subset that can be effectively computed for certain typed lambda terms.

4.2 Branching traversals

A normalization procedure based on enumerating all traversals is not practical because the set of traversals can be infinite. This is expected for non-normalizing term such as $\Omega = (\lambda x.xx)(\lambda y.yy)$ which has infinitely long traversals of the form $\lambda x \cdot x \cdot \lambda y \cdot y \cdot x \cdot \lambda y \cdot y \cdot \ldots$ But this is also the case for terms having a normal form. Take for example $M = \lambda f.f(\lambda x.x)$, then for all $k \geq 0$ the justified sequence $t_k = \lambda f \cdot f \cdot (\lambda x \cdot x)^k$ (with appropriate pointers) is a traversal.

Such infinite traversal is possible because of the (IVar) rule used to traverse free variables which has two non-deterministic choices:

(J) The variable to pick in the O-view (the justifier),

(L) The child lambda node to pick amongst the children of variable picked in (J).

The first choice allows the repeated pattern described above where the same node λx in the O-view is picked again and again within a single traversal.

Remark 4.1 (Game-semantic intuition)**.** The Game Semantics correspondence explains why traversals can be infinite: In the game denotation of a lambda term M, at every point in a play where it is Opponent's turn to play, all possible Opponent moves must be accounted for. More generally, the game denotation must allow Opponent's moves modeling the behaviour of *all* contexts that can possibly interact with M. The traversals t_k thus accounts for all the possible denotations of the function parameter f of M: for each k, there exists a term that applies its argument k times: $F_k = \lambda g.g(g(\ldots(gz)))$ with k applications of g, and there got to be plays in the game denotation of the term M to accounts for those possible values of argument f. Fortunately, traversals accounting for all those contexts are redundant for normalization: due to the absence of side-effects, calling the same argument multiple times always involves the same underlying computation in M. We formalize this intuition with the notion of *branching traversals* (Definition 4.2) which prevent such repetitive behaviour while still covering all $\sim$-equivalence classes (Proposition 4.1).

One may view traversals as a mechanism to explore the beta-normal form of the term in a dept-first search manner. Under such a view, one can interpret the non-determinism in (IVar) as a 'branching point' in the exploration. In the absence of side-effect, it is sufficient to explore each possible branch only once. *Branching traversals* implement this idea by restricting the rule (IVar) so as to traverse only nodes leading to paths in the computation tree that are yet unexplored: when choosing the next lambda node to visit, it forces choice '(J)' to be the *latest* variable node in the traversal, and restrict choice '(L)' to be some child lambda node of that variable node.

Definition 4.2. We define the set of ***branching traversals*** $\mathcal{T}rav^{\mathsf{branch}}$ as the subset of $\mathcal{T}rav$ defined by induction with the rules of Table 1 where the justifier node in the input-variable rule (IVar) is restricted to be necessarily the last node in the traversal ($m = n$).

We will use the subscript '$_{\mathsf{branch}}$' to refer to the rule system thus obtained. Using a 'derivation rule'-based presentation, the input variable rule can thus be stated follows:

$$\frac{t \cdot n \in \mathcal{T}rav^{\mathsf{branch}} \qquad n \in N^{\vdash^*} \cap N_{\mathsf{var}} \qquad n \vdash_i \alpha \qquad i \geq 1}{t \cdot \overset{\curvearrowleft i}{n \cdot \alpha} \in \mathcal{T}rav^{\mathsf{branch}}} \quad (\mathsf{IVar}_{\mathsf{branch}})$$

Remark 4.2 (Game semantic intuition)**.** From a game semantic point of view, restricting the opponent moves can be interpreted as restricting the set of contexts in which the term can be used, and more particularly the set of terms that can be applied to it. The branching restriction prevents the context from calling the same parameter twice, which eliminates the Kierstead contexts (in which the argument f is called twice). It also excludes the context $\lambda f \lambda g.f(\lambda f(\lambda x.g(\lambda y.x)))$ where the the two variables f and g are bound by two consecutive lambdas.

Property 4.1. Let $t \in \mathcal{T}rav$ be a traversal which does not contain any ghost occurrence, and m be an occurrence in t of an external λ-node (*i.e.*, $m \in N_\lambda \cap N^{\vdash^*}$). Then $\pi(t_{<m}) = \pi(t)_{<m}$.

Proof. By an easy induction on t using the fact that when recursively calculating $\pi(t)$, a lambda node hereditarily enabled by the root resets the stack of pending lambdas. □

Proposition 4.1 (Branching traversals are $\sim$-complete)**.** $\mathcal{T}rav^{\mathsf{branch}}$ is $\sim$-complete.

Proof. Let $t \in \mathcal{T}rav$ and let's first assume that t does not contain any ghost occurrence. We show by strong induction on t that there is a subsequence $u \in \mathcal{T}rav^{\mathsf{branch}}$ of t such that $\ulcorner \pi(t) \urcorner = \ulcorner \pi(u) \urcorner$, and so in particular $\ulcorner \pi(t) \urcorner \equiv \ulcorner \pi(u) \urcorner$. We do a case analysis on the last node t^ω of t.

- $t^\omega \in N^{\vdash^*}$: Suppose t^ω is a variable or an @ node then we can conclude immediately from the induction hypothesis on $t_{<t^\omega}$ and using the rules (Lmd) of $\mathcal{T}rav^{\mathsf{norm}}$.

  Suppose t^ω is a lambda node. If it has no justifier then it is the root in which case we conclude by taking $u = t = t^\omega$. Otherwise let m denote t^ω's justifier in t. By the I.H. on $t_{\leq m}$ there is $u' \in \mathcal{T}rav^{\mathsf{norm}}$ such that $\pi(u') = \pi(t_{\leq m})$. Take $u = u' \cdot t^\omega$ where t^ω points to its immediate predecessor m. Then u is clearly a $\mathcal{T}rav^{\mathsf{norm}}$-traversal by rule (IVar) and:

  $$\begin{aligned} \ulcorner \pi(u) \urcorner &= \ulcorner \pi(u') \urcorner \cdot t^\omega && \text{Def. of } \pi \\ &= \ulcorner \pi(t_{\leq m}) \cdot t^{\omega} \urcorner && \text{By I.H. on } t_{\leq m} \\ &= \ulcorner \pi(t)_{\leq m} \cdot t^{\omega} \urcorner && \text{By Prop. 4.1 since } m \text{ is necessarily followed in } t \text{ by a } \lambda\text{-node in } N^{\vdash^*}. \end{aligned}$$

  Now because m justifies t^ω, by Def. of P-view, *up to renaming of lambda variables* the sequences $\ulcorner \pi(t)_{\leq m} \cdot t^\omega \urcorner$ and $\ulcorner \pi(t) \urcorner$ are equal: $\ulcorner \pi(t)_{\leq m} \cdot t^\omega \urcorner \equiv \ulcorner \pi(t) \urcorner$. But because m is a variable node, its label is kept untouched by the transformation π and therefore the equality holds.

- $t^\omega \notin N^{\vdash^*}$: Let n be the last occurrence in t that is in $N^{\vdash^*}$, and let $m_1 \ldots m_q$, $q > 0$ be the occurrences of internal nodes following n in t (so that $m_q = t^\omega$). By definition of the traversal rules, a variable in $N^{\vdash^*}$ is necessarily followed by a lambda node in $N^{\vdash^*}$, therefore n is necessarily a lambda node. By definition of π, n is therefore also the last occurrence in $\pi(t)$ therefore $\pi(t) = \pi(t)_{\leq n}$. But *up to relabelling*, $\pi(t)_{\leq n} = \pi(t_{\leq n})$; more precisely, $\pi(t)_{\leq n}$ is obtained from $\pi(t_{\leq n})$ by prepending to n's label the stack of pending lambdas of $t_{>m}$ (the internal nodes following n in t). Applying the induction hypothesis on $t_{\leq n}$ gives $\pi(t_{\leq n}) = \pi(u')$ for some $u' \in \mathcal{T}rav^{\mathsf{norm}}$. To conclude it therefore suffices that to show that $u = u' \cdot m_1 \ldots m_q$ is also a traversal of $\mathcal{T}rav^{\mathsf{norm}}$.

  We prove by finite induction on q that $u_{\leq m_q} \in \mathcal{T}rav^{\mathsf{norm}}$ and m_q's justifying node and label are same in u and t. For $q = 0$, we just apply rule (Lam) on u'. For $q > 0$: by case analysis on the rule used to visit m_q in t. For structural rules (Root), (App) and (Lam) it follows immediately by induction. Rule (Var): If $m_q \in N_{\mathsf{var}}$ then by the Path-View correspondence, $\ulcorner u_{<m_q} \urcorner$ is a path in the tree from the root to m_q therefore m_q's binder necessarily occur in u, we can therefore conclude using the I.H. on $u_{<m_q}$ and applying (Var) on $u_{<m_q}$ to get $u_{\leq m_q}$.

Suppose that t contains ghost occurrences then we consider the term M^t. By Prop. 3.2(i), $\eta_t(t)$ is a traversal of M^t. Hence by the above, there exists $u' \in \mathcal{T}rav^{\mathsf{norm}}(M^t)$ such that $\ulcorner \pi(u') \urcorner = \ulcorner \pi(\eta_t(t)) \urcorner$. By Prop. 3.2(ii), there exists $u \in \mathcal{T}rav(M)$ such that $u' = \eta_t(u)$. Recall that the normalizing traversals is defined as the subset of traversals where external lambda nodes always point to their immediate predecessor. Therefore since η_t is pointer-preserving u must necessarily belong to $\mathcal{T}rav^{\mathsf{norm}}(M)$.

By definition of π we have $\pi(t) = \pi(\eta_t(t))$, and since u is a subsequence of t we also have $\pi(u) = \pi(\eta_t(u))$. Hence $\ulcorner \pi(t) \urcorner = \ulcorner \pi(\eta_t(t)) \urcorner = \ulcorner \pi(u') \urcorner = \ulcorner \pi(\eta_t(u)) \urcorner = \ulcorner \pi(u) \urcorner$. □

The following result will be useful to prove termination of the normalization procedure for STLC:

Property 4.2 (Infinite branching traversals)**.** Let $t \in \mathcal{T}rav^{\mathsf{branch}}$ be a branching traversal. If t is infinite then it necessarily contains a ghost node occurrence.

Proof. Suppose t does not contain any ghost node then t is obtained without using ($\mathsf{Lam}^{\lambda\!\!\lambda}_{\mathsf{branch}}$) and without the 'eta-expansion' subcases of the variable rules. The remaining rules correspond precisely to the traversal rules of [11] (where (IVar) is renamed (Sig)). We can therefore appeal to the Spinal Decomposition Lemma [11, Lemma 14] which shows that if t is infinite then there is an infinite sequence of prefixes of t whose P-views (the *spine* of t) is strictly increasing. By the Path correspondence this means that there is an infinite path in the computation tree of M which gives a contradiction. Hence t is either finite or contains a ghost node. □

Property 4.3. Let M be an untyped term in eta-long form (*i.e.*, with respect to some simple type that it inhabits). Then:

(i) All traversals in $\mathcal{T}rav^{\mathsf{branch}}(M)$ are finite;

(ii) $\mathcal{T}rav^{\mathsf{branch}}(M)$ is finite.

Proof. By Proposition 3.3, traversals in $\mathcal{T}rav^{\mathsf{branch}}(M)$ do not contain any ghost node therefore Prop. 4.2 implies (i). (ii) Traversal rules that are not involving ghost nodes all have bounded non-determinism therefore a traversal only has a finite number of immediate extensions. Since traversals are finite by (i), this implies that the set of traversals is itself finite. □

4.3 Normalization procedure for eta-long forms (STLC)

The normalization procedure for STLC is given by Algorithm A.

Correctness

We now state the Paths Characterization of simply-typed terms which is an immediate consequence of the Game-Semantic Correspondence Theorem for STLC [7]:

Proposition 4.2 (Normalized Paths Characterization for STLC)**.** For every simply-typed term-in-context $\Gamma \vdash M : T$, let $\eta\beta_{\mathsf{lnf}}(M)$ denote the eta-long beta-normal form of M. We have the following equality:

$$\mathcal{T}rav(\eta_{\mathsf{lf}}(M))/\!\sim \; = \mathcal{P}aths(\eta\beta_{\mathsf{lnf}}(M))$$

Algorithm A Eta-long normalization by traversals for STLC

Input: A term M admitting an eta-long form (and thus inhabiting some simple-type).
Output: The tree of the eta-long beta-normal form of M.

1. Calculate the eta-long form $\eta_{\mathsf{lf}}(M)$ of M;
2. Enumerate maximal *branching* traversals of $\eta_{\mathsf{lf}}(M)$ from Def. 4.2:
    - For each traversal t, get the *traversal core* $\pi(t)$ by removing internal nodes and keeping external nodes;
    - Calculate the P-view of the traversal core $\ulcorner \pi(t) \urcorner$;
    - Interpret $\ulcorner \pi(t) \urcorner$ as a path in the tree representation of the eta-long β-nf of M;
3. Aggregate all paths thus obtained to construct the tree representation of the eta-long β-nf of M.

Proof. By soundness of Game Semantics, beta-eta equivalent terms have the same game denotation, therefore by Theorem 3.1, the set of traversals of the eta-long beta-normal form of M is also given by the set of cores of traversals of $\eta_{\mathsf{lf}}(M)$. The Path-View correspondence (Proposition 3.1) and Propostion 3.3 then show that the P-views of traversal cores of $\eta_{\mathsf{lf}}(M)$ give the tree paths of the eta-long beta normal form. □

Theorem 4.1 (Correctness of STLC normalization)**.** Algorithm A terminates and returns the eta-long beta-normal form of the input term.

Proof. Soundness A beta-normal term is uniquely determined by the set of maximal paths in its tree representation (Property 2.1). By Proposition 4.2, this set corresponds precisely to the set of core P-views, and by Proposition 4.1 its is also given by the core P-views of *branching* traversals. *Termination* By Property 4.3 for terms in eta-long form the set of branching traversals is finite and each traversal is itself finite, therefore the enumeration in Algorithm A terminates. □

Implementation The normalization procedure from algorithm A was first implemented in the HOG tool[5, 7]. The tool takes as input any simply-typed lambda term and lets the user interactively generate all the traversals by "playing the traversal game" over the tree representation of the term. The tool offers a 'worksheet' environment for performing various operations over the traversals, including filtering, views and core projection.

4.4 Arity threshold and normalizing traversals

We showed in the previous section that for eta-long forms, there is a finite number of branching traversals which implies termination of the normalization procedure of Algorithm A. In the general case of untyped terms, however, branching traversals can still be infinite because *on-the-fly eta-expansion* in rule (IVar) is unbounded: the justification label value $i > 0$ is unbounded. In this section, we show that, for the purpose of computing beta normal forms, this value can be bounded by a computable quantity called the ***arity threshold*** of a traversal.

When traversing a lambda term, ghost nodes are introduced on-demand each time an eta-expansion is deemed necessary.

Recall that our goal is to normalize the term (when it exists) by generating some finite representation of its normal form. We are *not* interested in generating the possibly infinite set of traversals corresponding to the game-semantic denotation of the term. We therefore don't want to eta-expand ad-infinitum: visiting ghost nodes will be useful only if they eventually lead to traversing some structural node of the tree.

Intuitively, after a sufficiently large number of eta-expansions, we are guaranteed to keep on traversing ghost nodes that never materialize back to structural nodes. This section formalizes this intuition by introducing the ***arity threshold*** as the maximum number of times necessary to eta-expand a given subterm (using rule (lVar)) in order to compute the set of paths of the beta-normal form of the term. We will show that such limit is sufficient to characterize the normal form of a lambda term when it exists: the traversal subset is $\sim$-complete.

Definition 4.3 (Strand)**.** We call ***strand*** of a traversal t, any sub-sequence of consecutive nodes from t, with even length, starting with a lambda nodes and finishing with a variable node both hereditarily justified by the root, and such that all the occurrences in-between are hereditarily justified by an application node (*i.e.*, not by the root).

From the parity property of traversals, a strand consists of alternations of lambda nodes and variable/application nodes. It is convenient to represent a strand as follows for some $k \geq 1$ where for j ranging from k down to 1, α_j is a lambda node in $N_\lambda^{\lambda\!\!\lambda}$ and n_j is a variable/application node in N_{var}^θ:

$$t = \cdots \underline{\alpha_k}\ n_k\ \alpha_{k-1}\ n_{k-1}\ \cdots\ \alpha_2\ n_2\ \alpha_1\ \underline{n_1} \cdots$$

The first and last occurrences of the strand are underlined; the $2k-2$ nodes occurring in between are those that are not hereditarily justified by the root.

For any occurrence n in t that is hereditarily justified by the root, we call ***strand ending at*** n the sequence of nodes ending at n that constitutes a strand. It can be obtained by taking the longest subsequence of nodes preceding n that are not hereditarily justified by the root.

Definition 4.4 (Traversal arity threshold)**.** Let t be traversal ending with a variable node hereditarily justified by the root. Let $\underline{\alpha_k}\ n_k\ \alpha_{k-1}\ n_{k-1}\ \cdots\ \alpha_2\ n_2\ \alpha_1\ \underline{n_1}$ be the strand of t ending at t^ω, (so that $n_1 = t^\omega$) for some $k > 0$ where $\alpha_j \in N_\lambda^{\lambda\!\!\lambda}$, and $n_j \in N_{\mathsf{var}}^\theta$ for all $1 \leq j \leq k$.

We define the ***arity threshold*** of t as:

$$\mathsf{arth}(t) = \max_{q=1..k-1} \left(|n_q| + \sum_{j=1..q-1} (|n_j| - |\alpha_j|) \right) .$$

Remark 4.3 (Calculating the arity threshold)**.** The arity threshold can be rewritten as:

$$\mathsf{arth}(t) = \max_{q=1..k-1} b_q .$$

where for $1 \leq q \leq k-1$ we define $b_q = \sum_{j=1..q}(|n_j| - |\alpha_j|) + |\alpha_q|$.

The b_qs verify the following induction:

$$\begin{aligned} b_1 &= |n_1| \\ b_{q+1} &= b_q + |n_{q+1}| - |\alpha_q| \qquad\qquad 1 \leq q \leq k-2 \end{aligned}$$

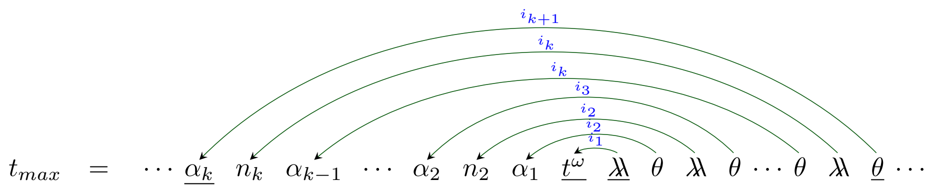
In other words: $b_q = |n_1| - |\alpha_1| + |n_2| - |\alpha_2| + \ldots + |n_{q-1}| - |\alpha_q|$. So an algorithm to calculate the arity threshold consists in adding and subtracting the arity of the nodes starting from the last occurrence of the traversal and reading backwards until reaching an external lambda node. The maximal value of the accumulator while performing this calculation is the arity threshold.

Observe that traversal rule (IVar) leaves infinitely many choices for the link label: any value greater than 1. The following property shows that if the link label exceeds the arity threshold then the traversal ends up traversing only ghost nodes that never materialize back to a structural node.

Property 4.4 (Weaving)**.** Let $t \in \mathcal{T}rav^{\mathsf{branch}}$ be a branching traversal ending with a variable node hereditarily justified by the root and $t_{max} \in \mathcal{T}rav^{\mathsf{branch}}$ be a maximal traversal extension of t. We consider the strand ending at t^ω: $\underline{\alpha_k}\ n_k\ \alpha_{k-1}\ n_{k-1}\ \cdots\ \alpha_2\ n_2\ \alpha_1\ \underline{n_1}$ of length $2k$, for some $k \geq 1$, with $n_1 = t^\omega$. By rule ($\mathsf{IVar}_{\mathsf{branch}}$), the occurrence t^ω is necessarily immediately followed by an external lambda node justified by t^ω with some label $i \geq 1$.

We then have:

(i) If $i > \mathsf{arth}(t)$ then t^ω is followed in t_{max} by a strand of length $2k$ consisting only of ghost nodes defined as follows:



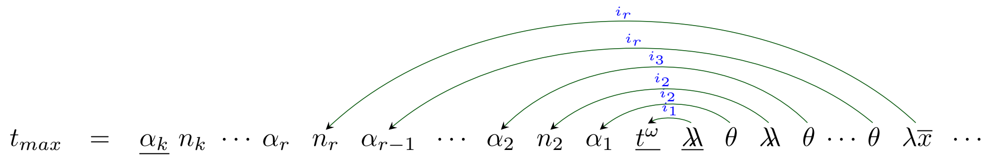
$$t_{max} = \cdots\ \underline{\alpha_k}\ n_k\ \alpha_{k-1}\ \cdots\ \alpha_2\ n_2\ \alpha_1\ \underline{t^\omega}\ \underline{\lambda\!\!\lambda}\ \theta\ \lambda\!\!\lambda\ \theta \cdots \theta\ \lambda\!\!\lambda\ \underline{\theta}\ \cdots$$

$$i_{q+1} = i + \sum_{j=1..q} (|\alpha_j| - |n_j|), \qquad \text{for } 0 \leq q \leq k$$

where underlined occurrences indicate external nodes.

(ii) If $i > \mathsf{arth}(t)$ then *all the nodes* following t^ω in t_{max} are ghost occurrences.

(iii) If $i \leq \mathsf{arth}(t)$ then for some $0 \leq r < k$ and link labels i_q, $0 \leq q \leq r$ defined as in (i), we have:



$$t_{max} = \underline{\alpha_k}\ n_k\ \cdots\ \alpha_r\ \ n_r\ \ \alpha_{r-1}\ \cdots\ \alpha_2\ n_2\ \alpha_1\ \underline{t^\omega}\ \underline{\lambda\!\!\lambda}\ \theta\ \lambda\!\!\lambda\ \theta \cdots \theta\ \lambda\overline{x}\ \cdots$$

where $\lambda\overline{x}$ is a structural internal lambda node, and consequently $i_r \leq |n_r|$.

Proof. (i) By the alternation property of traversals, the last strand of t consists of a succession of lambda nodes α_q and variable or @-nodes n_q, with indices going from k down to 1.

We show by finite induction on $1 \leq q \leq k$ that the first $2k$ nodes after t^ω are successive pairs of ghost lambda and ghost variable nodes justified in order by $n_1, \alpha_1, n_2, \alpha_2, n_3, \ldots, n_k, \alpha_k$ with respective labels $i_1, i_2, i_2, i_3, i_3, \cdots, i_k, i_k, i_{k+1}$ defined by:

$$\begin{cases} i_1 = i \\ i_{q+1} = i_q + |\alpha_q| - |n_q|,\ 1 \leq q \leq k. \end{cases}$$

- Base case $q = 1$: Because t^ω is a ghost variable node hereditarily justified by the root, the only next rule that can be applied is $(\mathsf{IVar}_{\mathsf{branch}})$, and by assumption the following node is a ghost lambda node justified by t^ω with label $i_1 = i$. Then by rule $(\mathsf{Lam}^{\lambda\!\!\lambda}_{\mathsf{branch}})$ the next node is a ghost variable justified by α with label $i_2 = i + |\alpha_1| - |t^\omega|$.

$$\begin{tikzcd}
t_{max} = & \cdots & \alpha_1 & t^\omega & \lambda\!\!\lambda \arrow[l, bend right, "i"'] & \theta \arrow[lll, bend right, "i_2"'] & \cdots
\end{tikzcd}$$

- For $1 < q \leq k$, by the induction hypothesis we have:

$$\begin{tikzcd}
t_{max} = & \cdots & \alpha_q & n_q & \alpha_{q-1} & n_{q-1} & \cdots t^\omega & \cdots & \lambda\!\!\lambda \arrow[lll, bend right, "i_{q-1}"'] & \theta \arrow[lllll, bend right, "i_q"'] & \cdots
\end{tikzcd}$$

  We then have:

$$\begin{aligned}
i_q &= i + \sum_{j=1..q-1} (|\alpha_j| - |n_j|) && \text{(by induction hypothesis)} \\
&> \mathsf{arth}(t) + \sum_{j=1..q-1} (|\alpha_j| - |n_j|) && \text{(assumption } i > \mathsf{arth}(t)) \\
&= \max_{r=1..k-1} \left(|n_r| + \sum_{j=1..r-1} (|n_j| - |\alpha_j|) \right) + \sum_{j=1..q-1} (|\alpha_j| - |n_j|) && \text{(definition of } \mathsf{arth}) \\
&\geq |n_q| + \sum_{j=1..q-1} (|n_j| - |\alpha_j|) + \sum_{j=1..q-1} (|\alpha_j| - |n_j|) && \text{(take } r = q) \\
&= |n_q|
\end{aligned}$$

  By definition of the strand, α_{q-1} is hereditarily justified by an @ node. Since $i_q > |n_q|$, by rule $(\mathsf{Var}_{\mathsf{branch}})$ the next node is necessarily a ghost lambda node justified by n_q with label i_q. With rule $(\mathsf{Lam}^{\lambda\!\!\lambda}_{\mathsf{branch}})$ the following node is a ghost variable node θ justified by α_q and labelled by $i_{q+1} = i_q + |\alpha_q| - |n_q|$.

(ii) By (i) the next strand following t consists solely of ghost nodes and ends with a ghost variable θ hereditarily justified by the root. Since ghost nodes have arity 0, the arity threshold at that point is 0. Hence, any label value j chosen to extend the traversal at that point will be strictly greater than the arity threshold: thus by (i) the next strand consists solely of ghost nodes. Applying the argument repeatedly shows that all the occurrences following t are necessarily ghost nodes.

(iii) Take r to be the smallest $q \geq 1$ such that $i_q \leq |n_q|$, it exists because:

$$\begin{aligned}
i \leq \mathsf{arth}(t) &\iff i \leq \max_{q=1..k-1} \left(|n_q| + \sum_{j=1..q-1} (|n_j| - |\alpha_j|) \right) \quad \text{(Definition of } \mathsf{arth}(t)\text{)} \\
&\implies i \leq |n_r| + \sum_{j=1..r-1} (|n_j| - |\alpha_j|) \quad \text{for some } 1 \leq r \leq k-1 \\
&\iff i + \sum_{j=1..r-1} (|\alpha_j| - |n_j|) \leq |n_r| \quad \text{for some } 1 \leq r \leq k-1 \\
&\iff i_r \leq |n_r| \quad \text{for some } 1 \leq r \leq k-1 \quad \text{(Definition of } i_r\text{)}.
\end{aligned}$$

We then conclude by applying the same argument as (i) for all $q < r$. □

This weaving property suggests that traversals in which the variable rule exceeds the arity threshold are not relevant for normalization: this leads to a stricter definition of traversals:

Definition 4.5 (Normalizing traversals)**.** We define the set of ***normalizing traversals***, noted $\mathcal{T}rav^{\mathsf{norm}}$ as the subset of branching traversals defined by the rules of Def. 4.2 where the index i in rule (lVar) is bounded by the arity threshold of the traversal.

$$\frac{t \cdot n \in \mathcal{T}rav^{\mathsf{norm}} \qquad n \in N^{\vdash^*} \cap N^{\theta}_{\mathsf{var}} \qquad n \vdash_i \alpha \qquad 1 \leq i \leq \mathsf{arth}(t)}{t \cdot \overset{\overset{i}{\curvearrowleft}}{n \cdot \alpha} \in \mathcal{T}rav^{\mathsf{norm}}} \quad (\mathsf{lVar_{norm}})$$

where $\alpha = \lambda\!\!\lambda$ just if $i > |n|$.

Table 2 recapitulates the rules of $\mathcal{T}rav^{\mathsf{norm}}$ using judgement derivation rules where the judgement notations "$\models t$" means "$t \in \mathcal{T}rav^{\mathsf{norm}}$".

Remark 4.4. Observe that $\mathcal{T}rav^{\mathsf{norm}}$ is *not* $\sim$-complete. Indeed take $M = \lambda x.x$ and the traversal $t = \lambda x \cdot x \cdot \lambda\!\!\lambda \cdot \theta$ in $\mathcal{T}rav^{\mathsf{branch}}$. We have $\ulcorner \pi(t) \urcorner = t$ which is not equivalent to $\ulcorner \pi(u) \urcorner$ for any u in $\mathcal{T}rav^{\mathsf{norm}} = \{\epsilon, \lambda x, \lambda x \cdot x\}$.

4.5 Normalization procedure for ULC

Algorithm B describes the ULC normalization procedure ULC that computes the β-normal of an untyped lambda term if it exists. It is a natural generalization of Algorithm A based on *normalizing* traversals instead of *branching* traversals.

Correctness

Like in the STLC case, correctness is show via a Path Characterization Theorem. Unlike in the STLC case, however, we cannot rely on the correspondence with Game Semantics to show this characterization since we have not formally proven such correspondence in the untyped setting (Conjecture 3.1). For ULC, we will instead use a term-rewriting argument based on the *leftmost linear reduction* strategy *leftmost linear reduction* from Section 5. We first show that $\sim$-equivalence classes of traversals of M are preserved by *leftmost-linear reduction.*

Algorithm B Normalization by traversals for the Untyped Lambda Calculus

Input: An untyped term M having a normal form.
Output: The computation tree of the normal form of M.

1. Enumerate maximal *normalizing* traversals $\mathcal{T}rav^{\mathsf{norm}}(M)$ using the rules of Table 2:
    - For each traversal, apply transformation π to get the traversal core;
    - Take the P-view of the traversal core;
    - Interpret the resulting sequence as a path in the tree representation of the β-nf of M.
2. Aggregate all the paths thus obtained to reconstruct the tree structure of the beta-normal form of M,
3. Assign labels to every node of the tree such that two variable nodes with same binder and same link label have the same variable name.

Property 4.5. Let M be an untyped term and $yA_1 \ldots A_n$ be a subterm of M for $n \geq 1$ where y is a variable not involved in any generalized redex then:

$$lloc_M(yA_1 \ldots A_n) = \begin{cases} lloc_M(A_j) & \text{where } j = \min\{1 \leq i \leq n \mid lloc_M(A_i) \neq \bot\} \\ \bot & \text{if } lloc_M(A_i) = \bot \text{ for all } 1 \leq i \leq n. \end{cases}$$

Proof. Immediate from the definition of *lloc* since $\lambda_l(y) = \epsilon$. □

Definition 4.6. A subsequence of a traversal is called a

- ***spinal descent*** if it is a path in the computation tree consisting solely of lambda and @-nodes, and where each lambda node (except the first one, if the first occurrence is a lambda node) is the 0th child of the preceding @-node.
- ***pending argument lookup*** if it consists of an alternation of ghost lambda nodes and ghost variables nodes, starting with an external ghost lambda node and terminated by a structural internal lambda node in N_λ.
- ***branching descent*** if it consists of an alternation of external structural lambda nodes and external structural variable nodes;

For any node n of the tree of M we write $M^{(n)}$ for the subterm of M rooted at n, and $lloc_M(n)$ to denote $lloc_M(M^{(n)})$.

We say that a traversal t is ***head-normal*** if it ends with a structural lambda node with undefined lloc (*i.e.*, $t^\omega = \lambda\overline{\eta}$ and $lloc_M(t^\omega) = \bot$).

Property 4.6. Let t be a non-empty traversal of M. Consider any subsequence u of t of the form $u = \underline{\lambda\overline{x}} \cdot v \cdot \underline{x}$ for some subsequence v of t, external structural variable node x and external structural lambda node $\lambda\overline{x}$.

(i) If v is empty then the node following u in t, if it exists, is necessarily a structural child lambda node of x (not a ghost node).

Proof. If v is empty then the strand ending at x is just $\underline{\lambda\overline{x}} \cdot \underline{x}$ therefore the arity threshold of t is precisely $|x|$ and so rule (IVar) can only be used to visit a structural child of x, and not ghost nodes. □

Proposition 4.3 (Qnf strand decomposition)**.** Let M be a term in quasi-normal form. Let t_{max} be a maximal traversal of M. Then t_{max} consists of a succession of strands of the following forms:

If the previous strand is	Then next strand is	Case
ϵ	$\underline{\lambda\overline{\eta_1}} \cdot u_1 \cdot \underline{x_1}$	A
$\underline{\lambda\overline{\eta_1}} \cdot u_1 \cdot \underline{x_1}$	$\underline{\lambda\overline{\eta_2}} \cdot u_2 \cdot \underline{x_2}$	B1
	$\underline{\lambda\!\!\lambda} \cdot v \cdot \lambda\overline{y_2} \cdot u_2 \cdot \underline{x_2}$	B2
$\underline{\lambda\!\!\lambda} \cdot v \cdot \lambda\overline{y_1} \cdot u_1^+ \cdot \underline{x_1}$	$\underline{\lambda\overline{\eta_2}} \cdot u_2 \cdot \underline{x_2}$	C1
	$\underline{\lambda\!\!\lambda} \cdot v \cdot \lambda\overline{y_2} \cdot u_2 \cdot \underline{x_2}$	C2

where

- $\lambda\overline{\eta_1}, \lambda\overline{\eta_2}$ are structural external lambda nodes verifying $lloc_M(\lambda\overline{\eta}_1) = lloc_M(\lambda\overline{\eta}_2) = \bot$,
- u_1 and u_2 are possibly empty *spinal descents* (internal nodes),
- x_1, x_2 are structural external variable nodes,
- u_1^+ is a non-empty *spinal descent* (internal nodes),
- v is a *pending parameter lookup* (ghost nodes) of length shorter than u_1^+,
- $\lambda\overline{y_1}$ and $\lambda\overline{y_2}$ are internal lambda nodes verifying $lloc_M(\lambda\overline{y_1}) = lloc_M(\lambda\overline{y_2}) = \bot$,
- $\lambda\overline{y_2}$ is justified by some @-node occurring in u_1^+.

Proof. (A) The first strand is obtained by applying the rule (Root) followed by repeated applications of (App) and (Lam), which yields u_1, a path to $\lambda\overline{\eta_1}$ in the tree of M. To show that u_1 is a spinal descent we need to prove that it does not contain any internal node (only @-node). This must be the case otherwise, since internal variables are by definition involved in some generalized redex, we would have $lloc_M(\lambda\overline{\eta_1}) = z$ where z is the first internal variable occurring in u_1, which contradicts $lloc_M(\lambda\overline{\eta}) = lloc_M(M) = \bot$ since M is in *qnf*.

(B1) If the node following x_1 is a structural node $\lambda\overline{\eta_2}$ then by rule (IVar), it's a child lambda node of x_1. Because u_1 is a spinal descent, the subterm rooted at $\lambda\overline{\eta}_1$ is of the form $\lambda\overline{\eta}_1.x_1 A_1 \ldots A_q$ for some $q \geq 1$, therefore we have $lloc_M(\lambda\overline{\eta}_1.x_1 A_1 \ldots A_q) = \bot$. Since x_1 is external, by Property 4.5 we have $lloc_M(A_j) = \bot$ for all $1 \leq j \leq q$, so in particular $lloc_M(\lambda\overline{\eta_2}) = \bot$. The same logic used for (A) permits to conclude that u_2 is a spinal descent to the external node x_2.

(B2) If the node following x_1 is a ghost node then by Property 4.6 u_1^+ is necessarily non-empty: we have $u_1^+ = @_1 \cdot \lambda\overline{\xi_1} \cdots @_q \cdot \lambda\overline{\xi_1}$ for some $q \geq 1$. By the weaving property 4.4, v is a *pending parameter lookup* that is necessarily shorter than u_1^+, and $\lambda\overline{y_2}$ is a structural lambda node justified by some application node $@_r$ in u_1^+ for some $1 \leq r \leq q$:

$$\begin{tikzcd}[column sep=small] t = \underline{\lambda\overline{\eta}} \cdot @_1 \cdot \lambda\overline{\xi_1} \cdot \cdots \cdot & @_r & \lambda\overline{\xi_r} \cdot \cdots \cdot @_q \cdot \lambda\overline{\xi_q} \cdot & \underline{x} & \cdot \lambda\!\!\lambda & \cdot v \cdot & \lambda\overline{y_2} \arrow[llllll, bend right, "k_r"'] \\ \end{tikzcd}$$

$$\begin{tikzcd} \underline{x} & \lambda\!\!\lambda \arrow[l, bend right, "k"'] \end{tikzcd}$$

$$v = \theta_{k_q} \cdot \lambda\!\!\lambda_{k_q} \cdot \theta_{k_{q-1}} \cdot \lambda\!\!\lambda_{k_{q-1}} \cdot \cdots \cdot \lambda\!\!\lambda_{k_{r-1}} \cdot \theta_{k_{r-1}}$$

where for all $r \leq i \leq q$ the ghost nodes θ_{k_i} and $\lambda\!\!\lambda_{k_i}$ points respectively to $\lambda\overline{\xi_i}$ and $@_i$ with link label $k_i \geq 1$ uniquely determined from k and the arity of the nodes in u_1^+ as follows:

$$k_i = k + |\lambda\overline{\xi}_q| - |x| + \sum_{i \leq j \leq q-1} (|\lambda\overline{\xi}_j| - |@_{j+1}|) \tag{2}$$

$$= k - |x| + |\lambda\overline{\xi}_i| + \sum_{i+1 \leq j \leq q} (|\lambda\overline{\xi}_j| - |@_j|) \tag{3}$$

Further, since all the nodes in v are ghost nodes and $\lambda\overline{y_2}$ is not, by Property 3.3 we have:

$$k > |x| \tag{4}$$

$$k_i > |@_i| - |\lambda\overline{\xi_i}| \quad \text{for all } r+1 \leq i \leq q \tag{5}$$

$$k_r \leq |@_r| - |\lambda\overline{\xi_r}| \tag{6}$$

We show by induction on $q - r$ that $lloc_M(\lambda\overline{\xi_r}) = \bot$ and that the lambda-list at $@_r$ is of size $\sum_{r+1 \leq j \leq q} |@_j| - |\lambda\overline{\xi_j}|$. *Base case:* If $q = r$ (in which case $v = \epsilon$) then $\lambda\overline{y_2}$ and $\lambda\overline{\xi_q}$ share the same parent $@_q$, and $\lambda\overline{y_2}$ is the k_qth child of $@_q$.

Because x is an external variable we necessarily have $lloc_M(\lambda\overline{\xi_r}) = lloc_M(\lambda\overline{\xi_q}) = lloc_M(\lambda\overline{\xi_q}\ x) = \bot$.

The lambda list at $@_r$ is $\lambda_l(@_r) =$

Induction case: Suppose it's true for all $r < j \leq q$, then because of Eqn. 5 we have $k_{r+1} > |@_{r+1}| - |\lambda\overline{\xi_{r+1}}|$

...

We can now show that $lloc_M(\lambda\overline{y_2}) = \bot$. The subterm at $@_r$ is of the form

$$(\lambda\overline{\xi_r}.A_0) A_1 \ldots A_{k_r - 1}\ \lambda\overline{y_2} \ldots A_{|@_r|}$$

By assumption we have that $lloc_M(\lambda\overline{\eta}) = \bot$ therefore since $@_r$ occurs in the spine of M, by definition of $lloc_M$ we necessarily have $lloc_M(@_r) = \bot$, (and thus also $lloc_M(\lambda\overline{\xi_r}) = \bot$). Further the lambda-list at $\lambda\overline{\xi_r}$ has size

$$|\lambda_l(\lambda\overline{\xi_r}.A_0)| = \sum_{r+1 \leq j \leq q} |@_j| - |\lambda\overline{\xi_j}| \tag{7}$$

$$= k - |x| + |\lambda\overline{\xi_r}| - k_r \tag{8}$$

$$> |\lambda\overline{\xi_r}| - k_r \quad (by Eq. \mathbf{??}) \tag{9}$$

$$> \tag{10}$$

Hence by Property 4.5, we necessarily have $lloc_M(\lambda\overline{y_2}) = \bot$.

The sequence u_2 is shown to be a spinal descent by the same argument as in (B1) using the fact that $lloc_M(\lambda\overline{y_2}) = \bot$.

(C1) & (C2) □

Proposition 4.4. If M is in *quasi normal form* and t is a traversal in $\mathcal{T}rav^{\mathsf{norm}}(N)$ Then.

(a) $lloc_M(\alpha) = \bot$ for all external lambda node $\alpha \in N_\lambda$ occurring in t.

(b) t does not contain any internal variable node (*i.e.*, all the internal nodes are @ nodes);

(c) t is finite.

Proof. (a) and (b) are direct consequences of Proposition 4.3. (c) Let's consider the cases of the strand decomposition of Proposition 4.3 observe that in case (A), (B1), because u_1 and u_2 are spinal descents, the last node in the 'next' strand (x_1 or x_2) must be a descendant (in the computation tree representation of M) of the first node from the 'previous' strand ($\lambda\overline{\eta_1}$). In case (B2), since both u_1 and u_2 are spinal descents, there are also paths in the tree, therefore since $\lambda\overline{y_2}$ points to u_1 we also have that x_2 is a descendant of $\lambda\overline{\eta_1}$. In case (C1) and (C2), since u_1^+ and u_2 are spinal descents, x_2 is a descendant of $\lambda\overline{y_2}$, itself a descendant of $\lambda\overline{y_1}$.

Suppose t is infinite, then t is maximal. Thus if t_{max} was infinite, from the above strand decomposition we can construct an infinite path in the tree of M which contradicts the fact that M is a finitary term. □

Proposition 4.5 (Traversals implement linear reduction)**.** Let M be an untyped term.

(i) If $M \to_l N$ then $\mathcal{T}rav^{\mathsf{norm}}(M) \cong \mathcal{T}rav^{\mathsf{norm}}(N)$;

(ii) If M is in *qnf* and has beta-normal form N then $\pi(\mathcal{T}rav^{\mathsf{norm}}(M)) = \pi(\mathcal{T}rav^{\mathsf{norm}}(N))$.

- finish proof - explain where the relabelling done in π is used in the proof. Answer: during trivial reduction of the qnf, in (ii).

Proof sketch. (i) Proof sketch: The effect of leftmost linear reduction is essentially to inline the *lloc* variable by the argument of the generalized redex involving it. The sets of traversals $\mathcal{T}rav^{\mathsf{norm}}$ before and after reduction are identical up to some mapping between nodes of the term and its reduct.

(ii) By Theorem 5.2 a *qnf* trivially reduces to its beta-nf: there is a beta reduction sequence $M = M_1 \to_\beta M_2 \ldots \to_\beta M_q = N$, $q \geq 1$ where for each reduction step, M_{q+1} is obtained by reducing the leftmost standard redex of M_q of the form $(\lambda x.U)T$ such that U does not contain any redex. By Proposition 5.1(ii), each such redex is trivial: x does not occur freely in U and therefore no substitution is performed to reduce the redex.

We show that $\pi(\mathcal{T}rav^{\mathsf{norm}})$ is an invariant that is preserved by this reduction sequence: we prove by induction on q that $\pi(\mathcal{T}rav^{\mathsf{norm}}(M_q)) = \pi(\mathcal{T}rav^{\mathsf{norm}}(M))$. For $q = 1$ we have $M = M_q = N$ so the result holds trivially.

Let $q \geq 1$.

By Proposition 5.1, the *qnf* trivially reduces to its beta-nf. It's immediate to show □

Property 4.7. Let M be an untyped term.

(i) All traversals in $\mathcal{T}rav^{\mathsf{norm}}(M)$ are finite;

(ii) $\mathcal{T}rav^{\mathsf{norm}}(M)$ is finite.

Proof. (i) If M has a normal form then by Theorem 5.2 its linear reduction sequence terminates, and by Prop. 4.5(i) the set of traversals of M is isomorphic to the traversals of its quasi normal form. Hence if M has an infinite traversal then so does its *qnf*, which contradicts Proposition **??**.
(ii) Traversals rules defining $\mathcal{T}rav^{\mathsf{norm}}$ all have bounded non-determinism therefore by (i) $\mathcal{T}rav^{\mathsf{norm}}$ is finite. □

Proposition 4.6. Algorithm B terminates.

Proof. By Prop. 4.7 there is a finite number of traversals in $\mathcal{T}rav^{\mathsf{norm}}$ and each of them can be obtained with finetly many applications of travesal rules. □

Theorem 4.2 (Normalized Paths Characterization for ULC)**.** For any untyped term M with a normal form T we have the following isomorphism

$$\mathcal{T}rav^{\mathsf{norm}}(M)/\sim \;\cong\; \mathcal{P}aths(T)$$

Proof of Theorem 4.2. Let's first assume that M is in beta-normal form. Then its computation tree does not contain any @-node, therefore every node is external (*i.e.*, hereditarily enabled by the root). Hence the arity threshold of any justified sequences t ending with external variable x is precisely $|x|$, the arity of x. A trivial induction on the traversal rules thus show that $\mathcal{T}rav^{\mathsf{norm}}(M) = \mathcal{P}aths(M)$. Since all nodes are externals, the projection π and $\ulcorner\_\urcorner$ functions both coincides with the identity function thus $\ulcorner\pi(\mathcal{T}rav^{\mathsf{norm}}(M))\urcorner = \mathcal{T}rav^{\mathsf{norm}}(M) = \mathcal{P}aths(M)$.

Otherwise, since M has a beta-nf, by Theorem 5.2 its *leftmost linear reduction sequence* terminates and yields the *quasi normal form* (qnf). By Prop. 4.5(i) the set of traversals is preserved by *linear reduction* (up to an isomorphism) thus the set of traversals of the *qnf* is isomorphic to the set of traversals of M. By Prop. 4.5(ii), taking the projection π yields $\mathcal{T}rav^{\mathsf{norm}}(T)$. □

Soundness of the normalization procedure then follows from Theorem 4.2 and the fact that a term is uniquely characterized by its set of paths $\mathcal{P}aths$ (Property 2.1). The last step of the algorithm (assignment of labels to the tree nodes) is sound because by Property 2.1 justification pointers uniquely determine variables names.

Remark 4.5. The infinitary counterpart of the characterization theorem for lambda terms with no normal form (not shown in this paper) can be stated as:

$$\bigcup_{k\geq 0} \mathcal{T}rav^{k}(M)/\sim) = \mathcal{P}aths(BT(M))$$

where $\mathcal{T}rav^{k}(M)$ denotes the set of traversals of length k at most and $\mathcal{P}aths(BT(M))$ denotes the set of *possibly infinite* maximal paths in the Böem tree $BT(M)$ of M (the infinitary computation tree of the term obtained by reducing beta-redexes ad-infinitum [2]).[1]

4.5.1 Reading out the normal form

For completeness, we include Algorithm C which gives a more detailed description of the normalization procedure with the steps involved in reconstructing the term tree from the set of paths, and in particular how to assign names to the ghost variables.

[1]See also [8, 11] for a formal definition of infinitary computation trees.

PROGRAM - Structural rules

$$\frac{r \text{ is the root of } CT(M)}{\models r} \qquad (\mathsf{Root}_{\mathsf{norm}})$$

$$\frac{\models t\cdot @ \qquad @ \vdash_0 \alpha}{\begin{tikzcd}[column sep=tiny] \models t\cdot & @ & \cdot & \alpha \arrow[ll, bend right, "0"'] \end{tikzcd} \qquad (\alpha \in N_\lambda)} \qquad (\mathsf{App}_{\mathsf{norm}})$$

$$\frac{\models t\cdot\alpha \qquad \alpha\in N_\lambda \qquad \mathsf{Ch}(\alpha)\in N_@}{\models t\cdot\alpha\cdot\mathsf{Ch}(\alpha) \qquad \mathsf{Ch}(\alpha) \text{ has no pointer}} \qquad (\mathsf{Lam}^{@}_{\mathsf{norm}})$$

$$\frac{\models u\cdot\beta\cdot v\cdot\alpha \quad \alpha\in N_\lambda \quad \mathsf{Ch}(\alpha)\in N_{\mathsf{var}} \quad \beta\vdash_i \mathsf{Ch}(\alpha), i\geq 1 \quad \beta \text{ visible at } \alpha}{\begin{tikzcd}[column sep=tiny] \models u\cdot & \beta & \cdot v\cdot\alpha\cdot & \mathsf{Ch}(\alpha) \arrow[ll, bend right, "i"'] \end{tikzcd}} \qquad (\mathsf{Lam}^{\mathsf{var}}_{\mathsf{norm}})$$

$$\frac{\begin{tikzcd}[column sep=tiny] \models t\cdot & \alpha & \cdot & m & \ldots & \lambda\!\!\lambda \arrow[ll, bend right, "i"'] \end{tikzcd}}{\begin{tikzcd}[column sep=tiny] \models t\cdot & \alpha & \cdot & m & \ldots & \lambda\!\!\lambda \arrow[ll, bend right, "i"'] & \cdot & \theta \arrow[llllll, bend right, "|\alpha|+i-|m|"'] \end{tikzcd} \qquad (\alpha\in N^{\lambda\!\!\lambda}_\lambda, m\in N^\theta_{\mathsf{var}})} \qquad (\mathsf{Lam}^{\lambda\!\!\lambda}_{\mathsf{norm}})$$

PROGRAM - Copy-cat rules

$$\frac{\begin{tikzcd}[column sep=tiny] \models t\cdot m\cdot & \alpha & \ldots & n \arrow[ll, bend right, "i"'] \end{tikzcd} \qquad n\in N^\theta_{\mathsf{var}}\setminus N^{\vdash^*} \qquad m\vdash_i\beta \qquad i>0}{\begin{tikzcd}[column sep=tiny] \models t\cdot & m & \cdot & \alpha & \ldots & n \arrow[ll, bend right, "i"'] & \cdot & \beta \arrow[llllll, bend right, "i"'] \end{tikzcd} \qquad (m\in N^\theta_{\mathsf{var}}\cup N_@, \alpha,\beta\in N^{\lambda\!\!\lambda}_\lambda)} \qquad (\mathsf{Var}_{\mathsf{norm}})$$

DATA - Input-variable rules

$$\frac{\models t\cdot n \qquad n\in N^\theta_{\mathsf{var}}\cap N^{\vdash^*} \qquad n\vdash_i\alpha \qquad 1\leq i\leq \mathsf{arth}(t)}{\begin{tikzcd}[column sep=tiny] \models t\cdot & n & \cdot & \alpha \arrow[ll, bend right, "i"'] \end{tikzcd} \qquad (\alpha\in N_\lambda)} \qquad (\mathsf{IVar}_{\mathsf{norm}})$$

where t, u, v range over (subsequences of) justified sequences of nodes.

Table 2: Normalizing traversals $\mathcal{T}rav^{\mathsf{norm}}$ of the untyped lambda calculus (ULC).

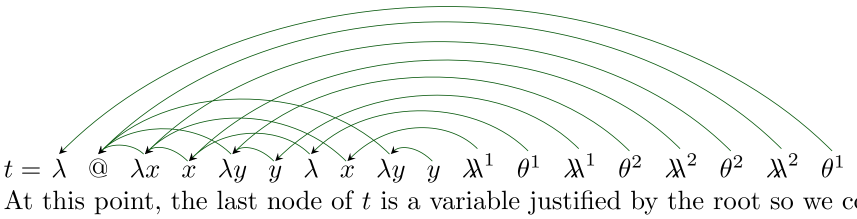
Algorithm C Normalization by traversals for the ULC with term read-out

```
Input: An untyped term $M$ that has a normal form.
Output: $CT(\beta_{\mathsf{nf}}(M))$, the tree of the normal form of $M$.
  $\mathcal{T}rav^{\mathsf{norm}} \leftarrow$ Enumerate all maximal normalizing traversals using rules of Table 2
  $C \leftarrow \{\ulcorner \pi(t) \urcorner \mid t \in \mathcal{T}rav^{\mathsf{norm}}\}$
  Let $(P, L)$ be the labelled-tree induced by $C$:
        • Paths $P \subseteq \mathbb{N}^*$ are given by the justified sequences in $C$: child index is 1
          for variable occurrences, and is given by the justification label for lambda
          nodes;
        • Label function $L : \mathbb{N}^* \to \tilde{N} \times P \times \mathbb{N}$ defined by $L(n) = (l, b, i)$ where
          $l$ is the label of the corresponding occurrence in $C$, $b$ is the path to the
          binder node (given by the justification pointer in $C$), and $i$ is the label of
          the justification pointer in $C$.
  for all node $n \in P$ (by depth-first enumeration) do
    Let $(l, b, i) = L(n)$
    if $o \in N_{\mathsf{var}} \cup N_{\lambda} \cup N_{@}$ then
      $L'(n) \leftarrow l$ {It's either @, some variable name $x$, or $\lambda x_1 \ldots x_m$ for $m \geq 0$}
    else if $o = \lambda\!\!\lambda$ then
      Let $\alpha$ be a fresh name {e.g., the path from the root to $n$}
      $id(n) \leftarrow \alpha$
      $arity \leftarrow \max\{i \mid L(v) = (\theta, n, i) \text{ for some } v \in P\}$
      $L'(n) \leftarrow \lambda \alpha_1 \ldots \alpha_{arity}$
    else if $o = \theta$ then
      $\alpha \leftarrow id(b)$ {Get the base name previously assigned to the binder}
      $L'(n) \leftarrow \alpha_i$ {Any two variable occurrences sharing same binder and justification label will
      have the same name.}
    end if
  end for
  return The labelled-tree $(P, L')$.
```

4.6 ULC Normalization Examples

In this section we demonstrate on various walk-through examples, how the beta-normal form of a term can be calculated by traversing the term using the normalizing traversals of Table 2.

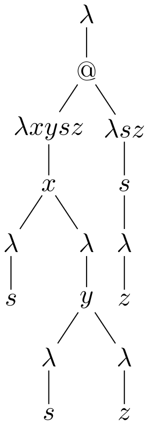
Example 4.1 (Baby example)**.** Take $M = (\lambda x.xx)(\lambda y.y)$. Repeatedly applying the structural traversal rules yields the following traversal:


$t = \lambda \ @ \ \lambda x \ x \ \lambda y \ y \ \lambda \ x \ \lambda y \ y \ \lambda\!\!\lambda^1 \ \theta^1 \ \lambda\!\!\lambda^1 \ \theta^2 \ \lambda\!\!\lambda^2 \ \theta^2 \ \lambda\!\!\lambda^2 \ \theta^1$

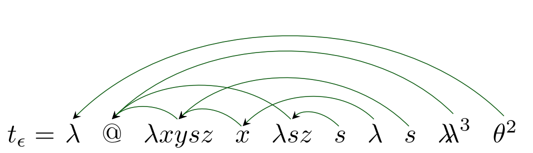
At this point, the last node of t is a variable justified by the root so we could apply rule (lVar) to get further game-semantic traversals. However, the traversal's arity threshold is $arth(t) = 0$

therefore there is no more *normalizing* traversal to explore: t is a maximal normalizing traversal. The P-view core of the traversal is $\ulcorner\pi(t)\urcorner = \lambda\ \theta^1$ thus the beta-normal form of M is, up to α-conversion $\lambda y.y$.

Example 4.2 (Church increment: "add 1")**.** Consider the Church numerals written $k \equiv \lambda sz.s^k z$ for $k \geq 0$. We consider a term M representing the increment function that adds 1 to the input integer: Mk reduces to $(k{+}1)$ for all k. Such term can be defined as $M \equiv \mathsf{add}\ 1$ where $\mathsf{add} \equiv \lambda xysz.x\, s(y\, s\, z)$. Note that M is not beta-normal. The computation tree of M is:



- Applying as many deterministic rules as possible gives the following traversal at which point the Opponent must make a choice:

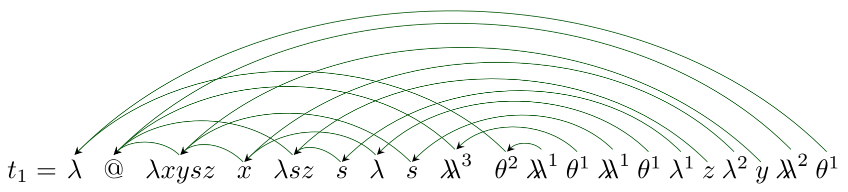


$t_\epsilon = \lambda\ \ @\ \ \lambda xysz\ \ x\ \ \lambda sz\ \ s\ \ \lambda\ \ s\ \ \lambda\!\!\lambda^3\ \ \theta^2$

(For readability we indicate the link label in exponent of the source node when representing traversals.) The core of t_ϵ is $\pi(t_\epsilon) = \lambda \cdot \theta^2$ and therefore $\ulcorner\pi(t_\epsilon)\urcorner = \lambda\cdot\theta^2$ is a path in $\beta_{\mathsf{nf}}(M)$. This means that $\beta_{\mathsf{nf}}(M)$ must be of the form $\lambda ys \ldots \cdot yN_1 \ldots\ldots N_q$ for some fresh variable y and s and $q \geq 0$.

- In order to determine what each argument N_k is in the final normal form, we eta-expand by applying rule (lVar) for each possible argument index $k \geq 1$ and then continue applying the traversal rules.

For $k = 1$ we get the traversal:



$t_1 = \lambda\ \ @\ \ \lambda xysz\ \ x\ \ \lambda sz\ \ s\ \ \lambda\ \ s\ \ \lambda\!\!\lambda^3\ \ \theta^2\ \lambda\!\!\lambda^1\ \theta^1\ \lambda\!\!\lambda^1\ \theta^1\ \lambda^1\ z\ \lambda^2\ y\ \lambda\!\!\lambda^2\ \theta^1$

The P-view of the traversal core is $\ulcorner\pi(t_1)\urcorner = \lambda \cdot \theta^2 \cdot \lambda\!\!\lambda^1 \cdot \theta^1$ which means that the normal form is of the form $\lambda ys \ldots \cdot s(yR_1 \ldots R_{q_2})N_2 \ldots N_q$ for some terms $R_1, \ldots R_{q_2}$, and $q, q_2 \geq 0$.

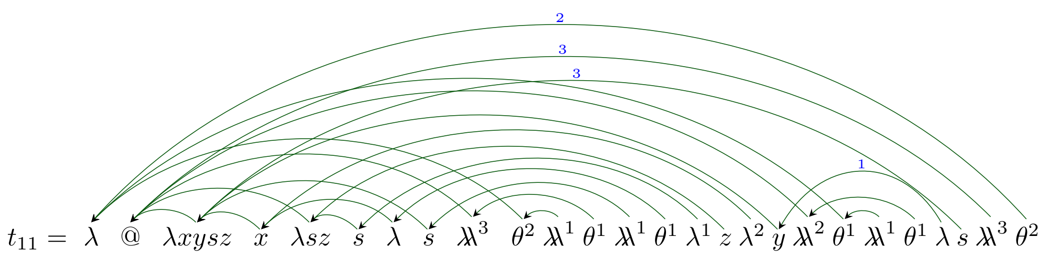
- What should be the value of q? In other words, how many more k do we need to look at? The answer: we need to keep iterating on k until the point where applying the traversal rules will only produce ghost variables and ghost lambda-nodes! Because there is a finite number of nodes in the computation tree, the variable node arities are bounded. Therefore for high enough index k, the eta-expansion case from rule (Var) will never be met: after applying rule (IVar) on t_ϵ, all subsequent extensions of the traversal will be constructed using repeated applications of rule (Lam^θ) or the eta-expanded case of rule (Var). The upper-bound q for k is precisely given by the *arity threshold* of the traversal t_ϵ as defined in 4.4. Here we have

$$\begin{aligned}
\mathsf{arth}(t_\epsilon) &= \max\{|s^2|, \\
&\qquad |s^1| + (|s^2| - |\lambda|), \\
&\qquad |x| + (|s^1| - |\lambda sz|) + (|s^2| - |\lambda|)\} \\
&= \max\{0, \\
&\qquad 1 + (0-0), \\
&\qquad 2 + (1-2) + (0-0)\} \\
&= 1
\end{aligned}$$

  Thus we don't need to look at higher value of k at t_ϵ, we thus have: $\beta_{\mathsf{nf}}(M) = \lambda ys \ldots \cdot s(yR_1 \ldots R_{q_2})$.

- The arity threshold of t_1 is $\mathsf{arth}(t_1) = |z| + |y| - |\lambda^2| = 0 + 2 - 0 = 2$ hence $\beta_{\mathsf{nf}}(M)$ is of the form $\lambda ys \ldots \cdot s(yR_1R_2)$.

- Let's eta-expand using rule (IVar) for varying value of child index $1 \leq k_2 \leq q_2 = 2$. We first look at the case $k_2 = 1$. The traversal obtained is:

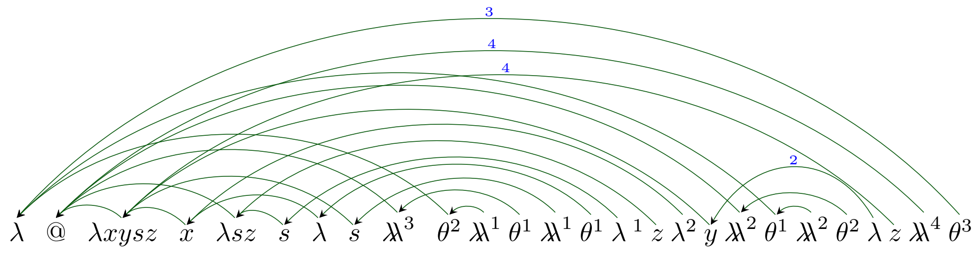


$$t_{11} = \lambda \;\; @ \;\; \lambda xysz \;\; x \;\; \lambda sz \;\; s \;\; \lambda \;\; s \;\; \lambda\!\!\lambda^3 \;\; \theta^2 \; \lambda\!\!\lambda^1 \; \theta^1 \; \lambda\!\!\lambda^1 \; \theta^1 \; \lambda^1 \; z \; \lambda^2 \; y \; \lambda\!\!\lambda^2 \; \theta^1 \; \lambda\!\!\lambda^1 \; \theta^1 \; \lambda \; s \; \lambda\!\!\lambda^3 \; \theta^2$$

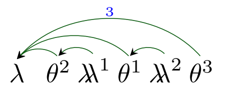
  Thus $\ulcorner\pi(t_{11})\urcorner = \lambda \;\; \theta^2 \; \lambda\!\!\lambda^1 \; \theta^1 \; \lambda\!\!\lambda^1 \; \theta^2$

  Hence $\beta_{\mathsf{nf}}(M)$ is of the form $\lambda ys \ldots \cdot s \; (y \; s \; R_2)$.

- Now let's extend t_1 with (IVar^λ) for $k_2 = 2$. We get the traversal:



$t_{12} = \lambda \ @ \ \lambda xysz \ x \ \lambda sz \ s \ \lambda \ s \ \lambda\!\!\lambda^3 \ \theta^2 \ \lambda\!\!\lambda^1 \ \theta^1 \ \lambda\!\!\lambda^1 \ \theta^1 \ \lambda^1 \ z \ \lambda^2 \ y \ \lambda\!\!\lambda^2 \ \theta^1 \ \lambda\!\!\lambda^2 \ \theta^2 \ \lambda \ z \ \lambda\!\!\lambda^4 \ \theta^3$

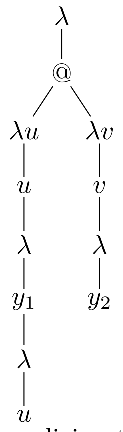


Thus $\ulcorner\pi(t_{12})\urcorner = \lambda \ \theta^2 \ \lambda\!\!\lambda^1 \ \theta^1 \ \lambda\!\!\lambda^2 \ \theta^3$

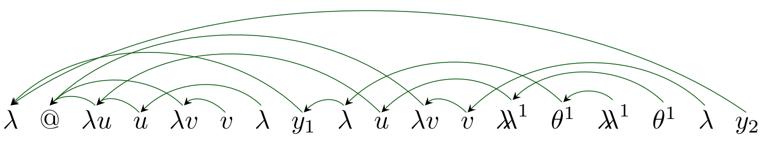
Hence $\beta_{\mathsf{nf}}(M) = \lambda ysz \cdot s \ (y \ s \ z)$.

Example 4.3 ("Missing operand" example by Neil Jones)**.** This small example illustrates how on-the-fly eta-expansion helps resolve the "missing argument" problem faced when using the traversal rules of STLC. Take $M = (\lambda u.u \ (y_1 \ u))(\lambda v.v \ y_2)$.

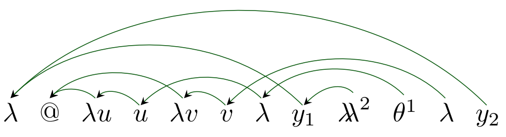
The computation tree is:



The only two maximal normalizing traversals are:



- $t_1 = \lambda \ @ \ \lambda u \ u \ \lambda v \ v \ \lambda \ y_1 \ \lambda \ u \ \lambda v \ v \ \lambda\!\!\lambda^1 \ \theta^1 \ \lambda\!\!\lambda^1 \ \theta^1 \ \lambda \ y_2$



- $t_2 = \lambda \ @ \ \lambda u \ u \ \lambda v \ v \ \lambda \ y_1 \ \lambda\!\!\lambda^2 \ \theta^1 \ \lambda \ y_2$

Using the STLC traversals, one can traverse t_1 all the way to variable v at which point one get stuck due to the lack of operand applied to the last occurrence of u (the occurrence at the bottom of the left branch in the tree). Using ULC rule, that operands gets created on-the-fly through eta-expansion and is represented by ghost lambda node $\lambda\!\!\lambda^1$.

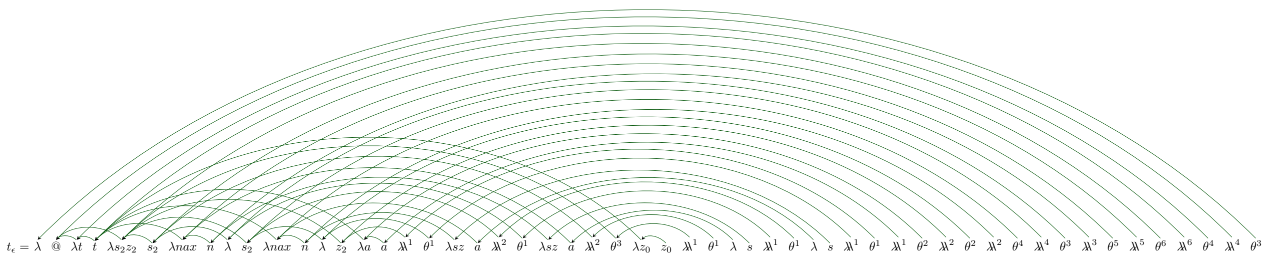

$t_\epsilon = \lambda$ @ λt t $\lambda s_2 z_2$ s_2 λnax n λ s_2 λnax n λ z_2 λa a $\lambda\!\!\lambda^1$ θ^1 λsz a $\lambda\!\!\lambda^2$ θ^1 λsz a $\lambda\!\!\lambda^2$ θ^3 λz_0 z_0 $\lambda\!\!\lambda^1$ θ^1 λ s $\lambda\!\!\lambda^1$ θ^1 λ s $\lambda\!\!\lambda^1$ θ^1 $\lambda\!\!\lambda^1$ θ^2 $\lambda\!\!\lambda^2$ θ^2 $\lambda\!\!\lambda^2$ θ^4 $\lambda\!\!\lambda^4$ θ^3 $\lambda\!\!\lambda^3$ θ^5 $\lambda\!\!\lambda^5$ θ^6 $\lambda\!\!\lambda^6$ θ^4 $\lambda\!\!\lambda^4$ θ^3

Table 3: Traversal t_ϵ of *varity* 2

The two core P-views give the two maximal paths in the beta-normal form of M:

- $\ulcorner \pi(t_1) \urcorner = \lambda \quad y_1 \quad \lambda \quad \theta^1 \quad \lambda\!\!\lambda^1 \quad y2$

- $\ulcorner \pi(t_2) \urcorner = \lambda \quad y_1 \quad \lambda\!\!\lambda^2 \quad y_2$

Thus: $\beta_{\mathsf{nf}}(M) = \lambda x.y_1(\lambda z.z\ y_2)y_2$.

Example 4.4 (Neil Jones' "*varity* 2" example)**.** Take $M = \mathit{varity}\ \mathit{two}$ where

$$\begin{aligned} \mathit{varity} &\equiv \lambda t.t(\lambda nax.n(\lambda sz.as(xsz)))(\lambda a.a)(\lambda z_0.z_0) \\ \mathit{two} &\equiv \lambda s_2 z_2.s_2(s_2\ z_2) \end{aligned}$$

In order to compute the set of normalizing traversals, we apply the traversal rules inductively until an input variable is reached, at which point we eta-expand with rule (IVar) for all possible values of k ranging from 1 to the arity threshold of the traversal. The first traversal obtained is denoted t_ϵ, and for every sequence of integer $s \in \mathbb{N}^*$, the traversal $t_{s\cdot k}$ represents the maximal traversal obtained after extending t_s using one application of the eta-expanded subcase of rule (IVar) with link-label $k \in \mathbb{N}$. This process yields the set of traversals $\{t_\epsilon, t_{11}, t_{12}, t_{121}, t_{122}\}$. Table 4.4 represents t_ϵ and Table 4.4 represents the three maximal traversals t_{11}, t_{121}, t_{122}.

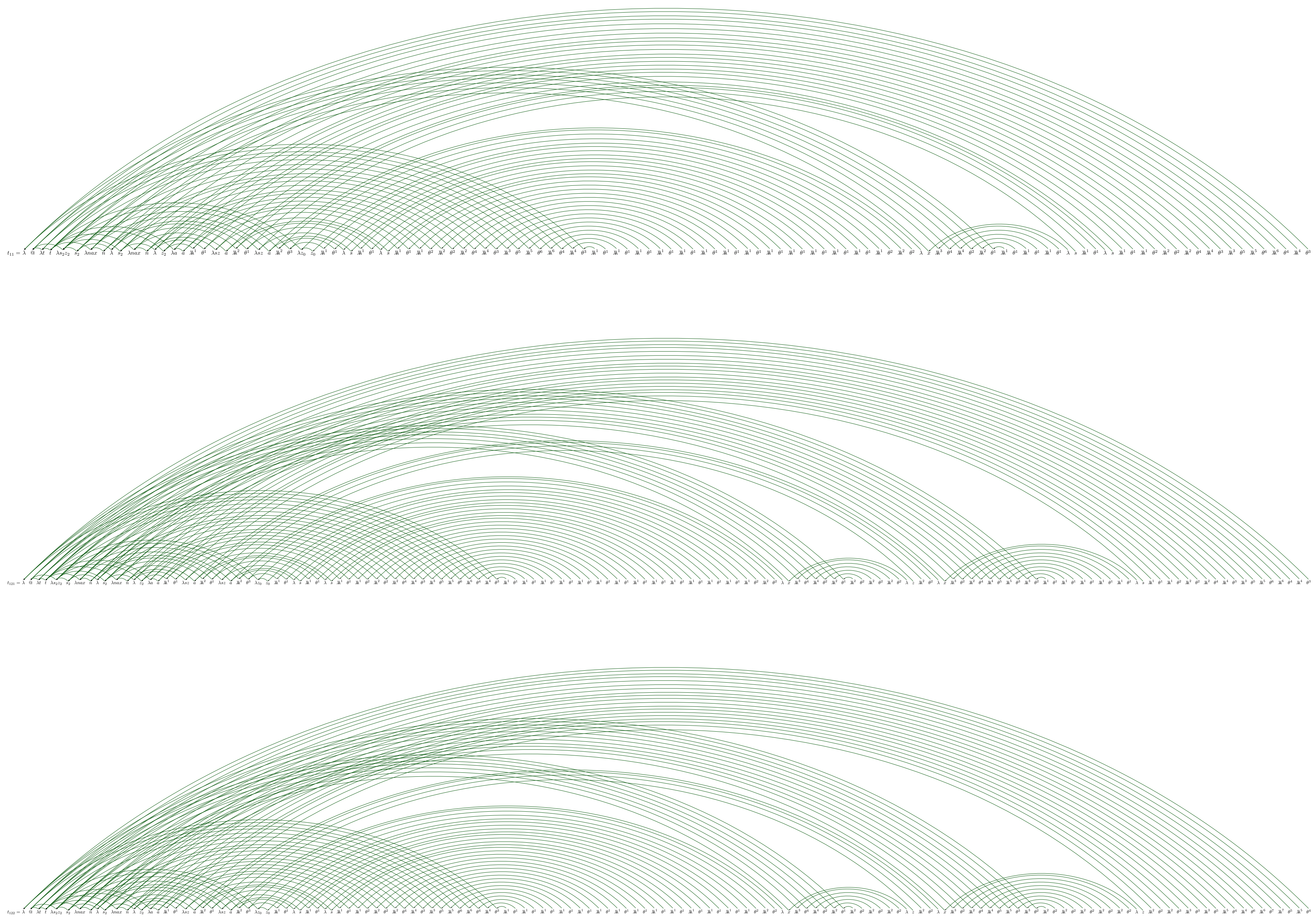


Table 4: Maximal traversals of *varity* 2

Keeping only the maximal traversals gives us the following set of maximal normalizing traversals of M:

$$\mathcal{T}rav^{\mathsf{norm}}_{max}(M) = \{t_{11}, t_{121}, t_{122}\}$$

Hence the set of maximal paths in the beta-normal form of M is given by the traversal core P-views:

$$\begin{tikzcd}
\ulcorner\pi(t_{11})\urcorner = \lambda & \theta^3 \arrow[l, bend right] & \lambda\!\!\lambda^1 \arrow[l, bend right] & \theta^1 \arrow[lll, bend right] & \lambda\!\!\lambda^1 \arrow[l, bend right] & \theta^3 \arrow[lllll, bend right]
\end{tikzcd}$$

$$\begin{tikzcd}
\ulcorner\pi(t_{121})\urcorner = \lambda & \theta^3 \arrow[l, bend right] & \lambda\!\!\lambda^1 \arrow[l, bend right] & \theta^1 \arrow[lll, bend right] & \lambda\!\!\lambda^2 \arrow[l, bend right] & \theta^2 \arrow[lllll, bend right] & \lambda\!\!\lambda^1 \arrow[l, bend right] & \theta^3 \arrow[lllllll, bend right]
\end{tikzcd}$$

$$\begin{tikzcd}
\ulcorner\pi(t_{122})\urcorner = \lambda & \theta^3 \arrow[l, bend right] & \lambda\!\!\lambda^1 \arrow[l, bend right] & \theta^1 \arrow[lll, bend right] & \lambda\!\!\lambda^2 \arrow[l, bend right] & \theta^2 \arrow[lllll, bend right] & \lambda\!\!\lambda^2 \arrow[l, bend right] & \theta^4 \arrow[lllllll, bend right]
\end{tikzcd}$$

Therefore the beta-normal from of *varity* 2 is:

$$\begin{aligned}
\beta_{\mathsf{nf}}(\mathit{varity}\ 2) &= \lambda x_1 x_2 x_3 x_4 . x_3 (x_1 x_3 (x_2 x_3 x_4)) \\
&\equiv_\alpha \lambda xysz.s(xs(ysz))
\end{aligned}$$

5 Leftmost linear reduction

5.1 Background: Lambda calculus and reduction

We recall from standard results of the lambda calculus. A ***redex*** is a sub-term of the form $(\lambda x.M)N$. Reducing redex $(\lambda x.M)N$, or also *firing* the redex, means substituting all free occurrences of x in M by the term N using capture avoiding substitution (the bound variable is renamed afresh when recursively substituting under a lambda).

A term is said to be in ***normal form*** if it does not contain any redex. A term is in ***head normal form*** if it can be written $\lambda x_1 \ldots x_n . y A_1 \ldots A_m$ for $n, m \geq 0$. If a term is not in head normal form then its ***head beta-redex*** is the left-most redex, otherwise the term does not have any head beta-redex.

For any reduction relation $\rightarrow$ between terms we will write $\rightarrow^*$ to denote its reflexive transitive closure.

The ***head reduction***, denoted $\rightarrow_h$, fires the head redex of a term if it exists. It can be shown that $\rightarrow_h^*$ yields the head-normal form. The ***normal-order reduction strategy***, also called leftmost-outermost reduction strategy, performs head reduction until reaching the head-normal form and then recursively applies head reduction on each operand of the head variable. This reduction strategy can be shown to yield the normal form if it exists.

5.2 Head-linear reduction

We now introduce the head-linear reduction of Danos-Regnier [9].

In the standard lambda calculus, a redex is necessarily formed by the outermost lambda in operator position of an application: if the operator consists of consecutive lambda abstractions (*e.g.*, as in $(\lambda x \lambda y.M)A_1A_2$) then the outermost lambda (*e.g.*, λx) is the one that will form the redex. The notion of redex can be generalized to allow evaluation of arguments in any order. In particular, one can allow any of the consecutive λ-abstractions to form a redex (*e.g.*, the abstraction λy and corresponding argument A_2 would be a valid redex). This is formalized by the notion of *generalized redex* (a generalized version of the *prime redex* of Danos-Regnier):

Definition 5.1 (Generalized redex)**.** The set of generalized redexes of a term M, written $gr(M)$, is a set of pairs $(\lambda x, A)$ where λx is some abstraction in M and A is a subterm called the argument of λx. The head λ list of M, written $\lambda_l(M)$ is a list of lambda abstractions of M. They are defined by induction:

$$\begin{array}{rlrll}
gr(v) &= \emptyset & \lambda_l(v) &= \emptyset & \\
gr(\lambda x.U) &= gr(U) & \lambda_l(\lambda x.U) &= \lambda x \cdot \lambda_l(U) & \\
gr(UV) &= \{(\lambda x, V)\} \cup gr(U) \cup gr(V) & \lambda_l(UV) &= l & \text{if } \lambda_l(U) = \lambda x \cdot l \\
gr(UV) &= gr(U) \cup gr(V) & \lambda_l(UV) &= \epsilon & \text{if } \lambda_l(U) = \epsilon.
\end{array}$$

where v ranges over variable occurrences, x ranges over variable names, U, V range over subterms of M, and ϵ denotes the empty list.

Example 5.1. For any term M, N, A_1, A_2 we have (i) $gr((\lambda x \lambda y.M)A_1A_2) = \{(\lambda x, A_1), (\lambda y, A_2)\}$ and (ii) $gr((\lambda z.(\lambda x \lambda y.M)N)A_1A_2) = \{(\lambda z, A_1), (\lambda x, N), (\lambda y, A_2)\}$.

Note: In order to define head-linear reduction, one needs to consider specific *occurrences* of variables and sub-terms in a given term. In particular, let's emphasize that when denoting a generalized redex by a pair $(\lambda x, A)$, the component λx and A denote specific *occurrences* of λx and subterm A.

We say that a variable occurrence is ***involved*** in the generalized redex $(\lambda x, A)$ if the variable occurrence is bound by λx. A variable occurrence can therefore be involved in at most one generalized redex. We define the ***linear substitution*** of x for A as the term obtained by performing capture-avoiding substitution of that single occurrence of x by A. (Compare this to the standard substitution that applies to *every* occurrence of x in M.) When performing such substitution we say that we ***linearly fire*** the generalized redex $(\lambda x, A)$ for that occurrence of x.

We define the ***head variable occurrence*** of a term as the left-most variable occurring in the term (*i.e.*, the first variable found by depth-first traversal of the term tree.) If the head variable occurrence of a term is involved in a generalized redex then we call that redex the ***head-linear redex***. A term that does not have a head-linear redex is said to be in ***quasi head-normal form***.

The ***head-linear reduction*** $\rightarrow_{hl}$ is defined as the reduction that linearly fires the head linear redex, if it exists. It can be shown that the reflexive transitive closure $\rightarrow_{hl}^*$ yields the quasi head-normal form.

Soundness The set of ***spine subterms*** is defined by induction: a term is a spine subterm of itself; the spine subterms of UV and $\lambda x.U$ are those of U. A ***prime redex*** (ala Danos-Regner) is a generalized redex $(\lambda x, A)$ such that the operator of the redex ($\lambda x.U$ for some U) is a spine subterm.

Danos-Reigner showed the following result:

Theorem 5.1 (Soundness and completeness of head-linear reduction [9]).

- If $T \rightarrow^*_{hl} T$ then T and T' are β-equivalent.
- If T is in quasi-head normal form and has n prime redexes then the head reduction of T leads to a head normal form in exactly n steps.
- If T is any term, the head linear reduction of T terminates iff the head reduction of T terminates.

5.2.1 Leftmost linear reduction strategy

As the name indicates, the head-linear reduction is a linear version of the *head* reduction. It thus only yields the quasi *head*-normal form, not the normal form, and therefore is not complete for normalization.

In the standard lambda calculus, normal-order reduction strategy is obtained by repeatedly applying head reduction to get to the head-normal form, and then continuously applying the head-reduction on each argument of the head variable. This ultimately yields the normal form of the term if it exists.

We now define the linear counterpart of the normal-order reduction: Informally, the ***leftmost linear reduction strategy*** is the strategy that performs head-linear reduction if possible, and otherwise, if the term is in quasi-head normal form, continuously (and recursively) applies the head-linear reduction on each argument (from left to right) of the head variable occurrence. Formally:

Definition 5.2 (Leftmost linear reduction). Given a term M, we write $\mathcal{V}(M)$ for the set of variable occurrences in M and $\mathcal{V}^{\perp}(M)$ for $\{\perp\} + \mathcal{V}(M)$ where the bottom element $\perp$ represents the 'undefined' occurrence. We introduce the partial order $\sqsubseteq$ on $\mathcal{V}^{\perp}(M)$ defined by: for all $x, y \in \mathcal{V}^{\perp}(M)$, $x \sqsubseteq y$ if and only if $x = y$ or $x = \perp$.

We define the partial function $lloc_M$ from subterms of M to variable occurrences by induction on the subterms of M:

$$lloc_M(v) = \begin{cases} v & \text{if } v \text{ is involved in a generalized redex in } M, \\ \perp & \text{otherwise.} \end{cases}$$

$$lloc_M(\lambda x.U) = lloc_M(U)$$

$$lloc_M(UV) = \begin{cases} lloc_M(U) & \text{if } lloc_M(U) \neq \perp, \\ lloc_M(V) & \text{if } lloc_M(U) = \perp \text{ and } \lambda_l(U) = \epsilon, \\ \perp & \text{if } lloc_M(U) = \perp \text{ and } \lambda_l(U) \neq \epsilon. \end{cases}$$

where v ranges over variable occurrences in M, and for any subterm N, $lloc_M(N) = \perp$ denotes that $lloc$ is undefined at N.

The ***leftmost linear variable occurrence*** of M is defined as $lloc_M(M)$ if it exists and the generalized redex it's involved in is called the ***lloc redex***, otherwise M is said to be in ***quasi normal form***, or *qnf* for short. The ***leftmost linear reduction*** strategy, written $\rightarrow_l$, is defined as the strategy that linearly fires the generalized redex involving the leftmost linear variable occurrence.

Leftmost linear reduction proceeds by first locating the left-most variable occurrence and, if it is involved in a redex, linearly fires it. In comparison, the traditional left-most outermost reduction first locates the leftmost redex and then fires it by substituting all the variables involved in it.

Example 5.2. Take $M = (\lambda x.xxN)z$. We have $M \rightarrow_{hl} (\lambda x.zxN)z$ which is in quasi head normal form because the head variable z is not involved in any generalized redex. But since the left-most occurrence x is involved in the redex $(\lambda x, z)$ the leftmost linear reduction gives $(\lambda x.zxN)z \rightarrow_l (\lambda x.zzN)z$.

Property 5.1. Some immediate properties:

(i) If N is a subterm of M then $lloc_N(N) \sqsubseteq lloc_M(N)$;

(ii) If M is beta-normal then $lloc_M(M) = \bot$;

(iii) If UV is in quasi normal form then so is U;

(iv) If $\lambda x.U$ is in quasi normal form then so is U;

(v) If M has a *lloc* redex then M has a redex;

(vi) Suppose M has at least one redex. If all the redexes are trivial then M is in *qnf*, otherwise M has a *lloc* redex.

Proof. (i) Immediate from the fact that generalized redexes of N are also generalized redexes of M. (ii) If $lloc_M(M) \neq \bot$ then M must contain a generalized redex and therefore must also contain a standard redex hence M is not beta-normal. (iii) If UV is in quasi normal form then $lloc_{UV}(UV) = \bot$ so by definition of *lloc* we must have $lloc_{UV}(U) = \bot$. By (i) since U is a subterm of UV this implies $lloc_{UV}(U) = \bot$. (iv) We have $\bot = lloc_{\lambda x.U}(\lambda x.U) = lloc_{\lambda x.U}(U) \sqsupseteq lloc_U(U)$. (v) and (vi) are immediate from the defintion of *lloc* redex. ☐

We say that a standard redex $(\lambda x.N)A$ is *trivial* if x does not occur freely in N in which case the redex can be *trivially reduced* to just N.

Proposition 5.1 (Trivial redexes in quasi normal forms)**.** Let M be a term in quasi normal form and suppose that M is not in beta-normal form. Then:

(i) If x is a variable occurrence in M involved in a generalized redex of M then x occurs in operand position in a subterm of the form:

$$(\lambda z.U)A_1 \ldots A_q(\ldots x \ldots) \qquad q \geq 0$$

(ii) Let $(\lambda x.N)A$ be the leftmost standard redex occurring in M such that N does not contain any redex. Then x does not occur freely in N.

Proof. (i) By induction on M. Suppose $M = v$ for some variable v then the result trivially holds since M has no generalized redex. Suppose $M = \lambda x.U$ then by Property 5.1(iv) U is in *qnf* and we conclude by the induction hypothesis. Suppose $M = UV$. If x occurs in U then we conclude by the induction hypothesis (U is in *qnf* by Property 5.1(iii)). If x occurs in V then if V is in *qnf* we conclude by the induction hypothesis. Otherwise by definition of *lloc* we necessarily have $\lambda_l(U) \neq \epsilon$. Hence U is either an abstraction or a redex with possibly multiple applied arguments. Consequently, so is UV which proves (i).

(ii) By induction on M. Suppose $M = v$ for some variable v then it trivially holds since M has no redex. Suppose $M = \lambda x.U$ then the result follows by the induction hypothesis since the redexes of M are those of U.
Suppose $M = UV$. Let's consider the three sub-cases depending on where the redex is located in UV:
   (1) The redex $(\lambda x.N)A$ is in U. Since M is in quasi normal form, by Property 5.1(iii) so is U, we can thus conclude by the induction hypothesis.
   (2) The redex $(\lambda x.N)A$ is in V. If $\lambda_l(U) = \epsilon$ then by definition of *lloc* we have $lloc_M(UV) = lloc_M(V) = \bot$ so we conclude by the induction hypothesis. If $\lambda_l(U) \neq \epsilon$, then either U contains a redex or U is an abstraction. But U cannot be a redex since by assumption N does not contain any redex, and U being abstraction would make $M = UV$ itself a redex which would contradict the fact that redex $(\lambda x.N)A$ in V is the leftmost redex in M.
   (3) $U = \lambda x.N$ and $V = A$. By assumption UV is in *qnf* therefore by Property 5.1(iii) and (iv), N is in *qnf*. Suppose x occurs freely in N then by the induction hypothesis (i), it must appear in operand position in consecutive applications to some lambda abstraction. But this contradicts the assumption that N does not contain any redex.

□

Theorem 5.2 (Soundness and completeness of leftmost linear reduction)**.** We have:

(i) If $M \to_l N$ then M and N are beta-equivalent.

(ii) If M is in *quasi normal form* and contains n standard redexes then there is an evaluation strategy that reduces M to its beta-normal form in exactly n (trivial) beta-reductions.

(iii) If M has a beta-normal form then it has a *quasi normal form.* (*i.e.*, its left-most head linear reduction terminates).

Proof. (i) TODO (ii) The idea consists in reducing the redex 'inside-out' from left to right. By Proposition 5.1(ii), each reduction is trivial and it is easy to see that *lloc* is preserved by trivial reduction, therefore each term in the reduction sequence is in *qnf*. The last term in the sequence has no redex therefore it's in beta-normal form.

Proof of (i) is a variation of Danos-Regnier proof for the same result for head-linear reduction.

(iii) TODO: Hopefully the proof of (ii) should not involve state machine/stack/environments, ... □

Definition 5.3 (Leftmost redex sequence)**.** The leftmost redex sequence $LRS(M)$ of a term M is a sequence of the generalized redexes of M defined by induction as follows:

$$\begin{aligned}
LRS(v) &= \emptyset \\
LRS(\lambda x.U) &= LRS(U) \\
LRS(UV) &= \begin{cases} (\lambda x, V) \cdot LRS(U) \cdot LRS(V) & \text{if } \lambda_l(U) = \lambda x \cdot l \text{ and } lloc_M(U) = \bot \\ (\lambda x, V) \cdot LRS(U) & \text{if } \lambda_l(U) = \lambda x \cdot l \text{ and } lloc_M(U) \neq \bot \\ LRS(U) & \text{if } \lambda_l(U) = \epsilon \text{ and } lloc_M(U) \neq \bot \\ LRS(U) \cdot LRS(V) & \text{if } \lambda_l(U) = \epsilon \text{ and } lloc_M(U) = \bot \end{cases}
\end{aligned}$$

where v ranges over variable occurrences and U, V over subterms of M.

Example 5.3. Take $M = (\lambda x.z((\lambda wy.y)x))U$ for any term U. Then M is in quasi-head normal form with one prime spine redex (x, U). Head reduction gives $M \to_h z((\lambda wy.y)U)$.

M is also in quasi normal form. Indeed $lloc_M(M) = lloc_M(\lambda wy.y)$ which is undefined. The leftmost redex sequence of M is $LRS(M) = (\lambda x, U) \cdot (\lambda w, x)$. Reducing the term with two head reduction gives $M \to_h z((\lambda wy.y)x) \to_h z(\lambda y.y)$ which is in normal form.

6 Further directions

6.1 Higher-order collapsible pushdown automata

In [11], a construction[2] is defined that converts a higher-order recursion scheme into an equivalent collapsible higher-order pushdown automaton (in the sense that they generate the same infinite tree). The automaton proceeds by calculating traversals over the tree representation of the higher-order recursion scheme. In that setting, the recursion scheme represents a closed term (no free variable) of order-0, and the generated structure is an order-0 value (a tree).

Conceivably, there must be some kind of tree-generating automaton that can be used to calculate the set of traversals of an untyped lambda term as defined in this paper. The generated tree would be an order-0 tree representation of the beta-normal of a term: each terminal symbol of the automaton would correspond to a "token" of the lambda term: either a lambda node (represented by non-terminal of type $o- > o$), or a variable node (represented by a non-terminal of type $o^k - > o$ where $k \geq 0$ is the number of operand applied to the variable in the beta normal form of the term.

In the untyped lambda calculus, though, there is no notion of order, therefore the type-based definition of higher-order pushdown automata, where each non-terminals has a simple-type, would not be appropriate here. The definition of HOCPDA would thus need to be relaxed to allow non-terminals of any type. The construction of the HOPCPDA itself would follow the same lines as [11] except that η-long normal expansion is not needed.

6.2 Connection with compilation of ULC to LLL

In [3], Daniil Berezun and Neil Jones define a different version of traversals with notable differences: no ghost lambda nodes, two types of justification pointers: binding pointers and control pointers. Although very similar, the relationship between their notion of traversals and the one defined in this paper is still not formally established. Danos and Regnier's Linear Head reduction paper [9] appears to be the best avenue to establish the connection.

6.3 Game semantics connection

In the lines of [7] where the connection between the theory of traversals and Game Semantics was formalized for typed languages including the simply-typed lambda calculus, PCF and Idealized Algol, a natural question to ask is whether there is a similar connection between the ULC traversals defined in this paper on game models of the untyped lambda calculus. We conjecture that the Game Semantic correspondence can be generalized to the game model of the untyped lambda calculs introduced in Andrew Ker's thesis [10] in which lambda-terms are denoted by *effectively almost everywhere copycat* strategy (Conjecture 3.1).

[2]This CPDA construction from HORS is implemented in the HOG tool.)

6.4 Head linear reduction

Traversals have emerged as a constructive method to represent Relationship between Game Semantics and

In 2004, Danos and Regnier [9] establish the connection between Game Semantics and the notion of *head-linear reduction*.

Head linear reduction is a non-standard reduction strategy for the lambda calculus and shed some light studies the connection with Game Semantics.

A Notes for Neil and Daniil

Here are some thoughts on comparing the following three approaches based on the conversations Neil and I had during his visit to MSR: (i) Berezun-Jones (BJ) traversals, (ii) Danos-Regnier's Pointer Abstract Machine (PAM) state runs, (iii) Blum's game-semantic-inspired imaginary traversals for ULC introduced in this note.

A.0.1 Normalization

- Danos-Regnier: PAM implements head-linear reduction. This reduction does not itself normalize the term, it only yields the head-normal form of the term! The paper does not clearly indicate any method to normalize the term based solely on head-linear reduction. In particular in the PAM transition of Section 5.2, step 2, if the hoc variable is free then the machine stops: no transition applies.
- Berezun-Jones: Although not explicitly explained in the paper. Daniil and Neil have a "read-out procedure"' to reconstruct the term from the traversal. The procedure is based on rewriting rules applied to the traversals.
- Blum: Normalization is performed by exploring all possible paths in the beta-normal form of the term. The algorithm generates one maximal traversal per path in the tree representation of the term. The term can then be uniquely recreated (up to variable naming) from those paths.

A.0.2 Traversal/Run structure

- Danos-Regnier: State run. Each state has three components: one pointer, a reference to a variable node, a reference to a subterm.
- Berezun-Jones: Traversals of 'tokens' with two pointers (control and binding pointers). A tokens is a reference to a node in the AST of the lambda term.
- Blum: Traversals are sequences of node with one back pointer. Nodes can either be structural nodes from the 'compact' AST of the term (where consecutive lambda nodes and application nodes are bundled together) or imaginary 'ghost' nodes. Each node occurrence except the first one has exactly one justification pointer.

A.0.3 Binding pointers – PAM pointers – P-view

Neil-Daniil's binding pointers represent the "head λ-list" notion in Danos-Regnier paper (page 3): it gives the list of lambda nodes encountered from the variable all the way to the root.

- Danos-Regnier: The De Bruijn index of a variable is used to find the binder in the head λ-list. If index is k (by Lemma 11) the binder can be retrieved by following k pointers in the PAM states. The notion of "subterm chain" of a variable correspond to the notion of P-view in Game Semantics.
- Berezun-Jones: The De Bruijn index is used to determine how many binding (green) pointers to follow to reach the occurrence of the binder in the traversal.
- Blum: Traversals are game-semantics inspired (ala Ong) and in particular verify the alternation property. To that end, consecutive binders are bulked together in a single binder node. Given a variable occurrence, its 'bulk' binder is located by reading the P-view backward by k' iterations where k' is statically determined as the distance between the variable occurrence and its binder in the computation tree. The label associated with the link determines which exact variable in the 'bulk lambda' binds the variable.

A.0.4 Control pointers – PAM –

A.0.5 Retrieving application operands: Control pointers – PAM's redex argument

- Danos-Regnier: the operands of a 'prime redex' is statically defined from the term's expression (the 'argument of a hoc variable', in Section 5.1)
- Berezun-Jones: control pointers are used to associate operators to the corresponding operand in an application: for instance if $@_1(\lambda x.M)N$ is a redex of the tree then any occurrence of λx will point to some previous occurrence of $@_1$ in the traversal. The token representing the operand N can then be statically determined from the AST of the tree. Control pointers always point to @ tokens.
- Blum: There are two cases:
  - Structural operands: Same as Ong's traversals: pretty much the same as Berezun-Jones with the only technical difference that consecutive lambda and applications nodes are 'bulked' together. In particular, 'bulk lambda' nodes point to their parent node (either a variable or an application node @). If the variable x of the redex is the ith bound variable in the bulk lambda, then the operand is obtained by fetching the ith child of the bulk lambda's parent node.
  - Eta-expanded operands: When an operand is missing, a ghost lambda node is first introduced with rule (Var) (corresponding to the variable obtained after eta-expanding the operator). Then through repeated application of rule ($\mathsf{Lam}^{\lambda\!\!\lambda}$) and ($\mathsf{Var}$), either a structural node gets eventually materialized, which then represents the operand of the application; or the traversal reaches a ghost variable hereditarily justified by the root. In the latter case, a new variable gets created through eta-expansion to represent the missing operand.

Note: the ghost materialization loop implemented by rule ($\mathsf{Lam}^{\lambda\!\!\lambda}$) and ($\mathsf{Var}$) proceeds by calculating the quantity "$|\alpha| + i - |m|$" at each iteration which is the exact same quantity $(r - a + l)$ used in step 2. (b) of the PAM transition [9], the "price to pay for having no stacks or environments".

A.0.6 Flag + Control pointers – PAM's "price to pay" – Ghost nodes

- Berezun-Jones: Semantics 2, 3, 4 and 5 from section 3 of [3] involve a 'flag' boolean parameter that indicates the context of the application (either weak or strong evaluation).
- Danos-Regnier: The 'flag' must somehow be present in the iterative loop in step 2(b) of the PAM transition–the "price to pay for having no stacks or environments" (Danos-Regnier).
- Blum: The 'flag' must be somehow be hidden in the iterative ghost materialization algorithm implemented by the (Var) and ($\mathsf{Lam}^{\lambda\!\!\lambda}$) rules. (The ghost nodes being the equivalent of the "price to pay" from Danos-Regnier's paper.

A.0.7 Converting other traversals to the imaginary ULC traversals

From Berezun-Jones

1. Remove single application token @ that are immediately followed by a variable.
2. Merge consecutive application nodes in the traversal, preserve all control pointers pointing to them. Assign link label 0 if the source of the pointer is in operator position of the @, and $i > 0$ if is in operand position of the ith merged @.

   Remove all binding and control pointers starting from the @ tokens themselves.
3. Merge consecutive lambda tokens $\lambda x_1, \ldots, \lambda x_n$, $n > 1$ in the traversals into single lambda nodes $\lambda x_1 \ldots x_n$. Binding pointers:
   - For any binding pointer from some variable token x in the traversal, pointing to one of the merged lambda nodes, replace the link as follows: if x has deBrujin index i then follow i binding pointer in the original sequence. This will lead to some lambda token λx. Make the link points instead to the merged lambda node containing that λx. Assigned link label i where i is the index of the bound variable in the merged list of $x_1 \ldots x_n$.
   - Remove all the binding pointers from either lambda or @ tokens that are pointing to any of the merged lambda nodes.
   - Remove all binding pointers from the merged lambda nodes to other nodes.

   Control pointers: preserve only the control pointer associated with the first lambda in the sequence of lambda tokens.
4. Add dummy lambdas λ and ghost nodes: for each pair (x, y) of consecutive variable tokens x and y, processing the traversal from left to right, reconstruct the "missing" dummy and ghost nodes between x and y by applying rules ($\mathsf{Lam}^{\mathsf{norm}}$) and ($\mathsf{Var}^{\mathsf{norm}}$).

   IT REMAINS TO PROVE THAT: by applying those rules repeatedly starting from x, we necessarily get back to variable nodes y with the expected justification pointer. The proof will involve formalizing the relationship between (i) flag and control pointers (ii) ghost nodes.

From Danos-Regnier Take a substitution sequence of a PAM: $((z_0, A_1), (z_1, A_2) \ldots$. (See my handwritten notes.)

A.0.8 Correctness

Blum The soundness and termination proof is mainly based on first principles. Not a game semantic argument.

Proof idea: "traversals implement left-most linear reduction". The proof sketch:

- Show that 'leftmost linear reduction' yields the 'quasi normal form'. (Theorem 5.2)
- Observe that if M is beta-normal then (trivially) its set of traversal cores P-views is precisely the set of paths in the tree representation of M
- If M reduces to N using the leftmost linear reduction then the traversal cores P-views of N are isomorphic to the traversals core P-views of M.
- Reducing a trivial redex preserves (modulo some traversal prefix) the set of traversals. The interesting part of the proof will be to show how the ghost nodes gradually "materialize" after each step of the linear reduction, and how the arity threshold gives the right bound for limiting eta-expansion.

From Danos-Regnier Proof based on environments.

References

[1] S. Abramsky. Algorithmic game semantics: a tutorial introduction. In *Proceedings of 2001 Marktoberdorf International Summer School*, 2001.

[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2013.

[3] D. Berezun and N. Jones. Compiling, untyped lambda calculus to lower-level code by game semantics and partial evaluation. *Galop*, 2017.

[4] W. Blum. A tool for constructing structures generated by higher-order recursion schemes and collapsible pushdown automata. 2007.

[5] W. Blum. A concrete presentation of game semantics. *Galop*, 2008.

[6] W. Blum. Local computation of beta-reduction - a concrete presentation of game semantics. *Technical Report - Oxford University Computing Laboratory*, 2008.

[7] W. Blum. *The Safe Lambda Calculus.* PhD thesis, University of Oxford, 2009. https://ora.ox.ac.uk/objects/uuid:537d45e0-01ac-4645-8aba-ce284ca02673.

[8] W. Blum. Type homogeneity is not a restriction for safe recursion schemes. *CoRR*, abs/1701.02118, 2017.

[9] V. Danos and L. Regnier. Head linear reduction. submitted for publication, 2004.

[10] A. D. Ker. Innocent game models of untyped lambda-calculus. *DPhil Thesis, Oxford University*, 2000.

[11] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *Logic in Computer Science, 2006 21st Annual IEEE Symposium on*, pages 81–90, 12-15 Aug. 2006.

[12] C. L. Ong. Normalisation by traversals. *CoRR*, abs/1511.02629, 2015.