A Compositional Cost Model for the λ-calculus

James Laird
Department of Computer Science
University of Bath

***Abstract*—We describe a (time) cost model for the (call-by-value) λ-calculus based on a natural presentation of its game semantics: the cost of computing a finite approximant to the denotation of a term (its evaluation tree) is the size of its smallest derivation in the semantics. This measure has an optimality property enabling compositional reasoning about cost bounds: for any term A, context $C[\_]$ and approximants a and c to the trees of A and $C[A]$, the cost of computing c from $C[A]$ is no more than the cost of computing a from A and c from $C[a]$.**

Although the natural semantics on which it is based is nondeterministic, our cost model is *reasonable*: we describe a deterministic algorithm for recognizing evaluation tree approximants which satisfies it (up to a constant factor overhead) on a Random Access Machine. This requires an implementation of the λ_v-calculus on the RAM which is *completely lazy*: compositionality of costs entails that work done to evaluate any part of a term cannot be duplicated. This is achieved by a novel implementation of graph reduction for nameless explicit substitutions, to which we compile the λ_v-calculus via a series of linear *cost reductions*.

I. Introduction and Related Work

Although they have evolved into different traditions, *semantics* and *algorithmics* share a core of compositional reasoning: the meaning of an expression is given by composing the meanings of its subexpressions; the cost of solving a problem (sequentially) is the sum of the costs of solving and combining its sub-problems. The aim of this research is to show that compositional reasoning, guided by semantics, can be extended to the cost-analysis of programs in high level languages: in this instance, the (call-by-value) λ-calculus.

Referential transparency makes compositional reasoning about functional programs particularly straightforward (this is often cited as a key benefit [1]). Their denotational semantics is correspondingly elegant. However, the costs of evaluating them are not compositional in general — neither in theory (β-reductions of λ-terms) nor in practice (run-time for evaluation in a typical functional programming language). Our goal is to define a model of computation for higher-order functions (based on computing their evaluation trees, which are syntactic representations of the innocent strategies of game semantics) and an implementation-independent measure of its (time) costs which is compositional, in a sense we now describe.

In general, denotational (i.e. compositionally defined) semantics of programming languages ignore the cost of computation, identifying terminating programs with the values to which they converge. In some cases, usage of resources such as time is captured as a computational side-effect — e.g. [2], [3]. We seek something more direct: a cost-measure on the evaluation of functional programs which allows the use of compositional reasoning to establish cost-bounds via simple principles such as the following:

The cost of evaluating $C[M]$ to V is no more than the cost of evaluating M to V and $C[U]$ to V.

much as compositionality of meaning can be used to reason about (e.g.) program equivalence. To be more precise, we say that a transitive binary relation $\mathcal{R}$ on the terms of a language $\mathcal{L}$ is (left) *compositional* if

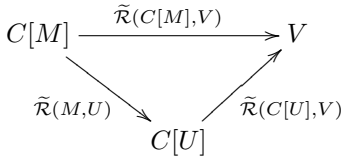
$$\mathcal{R}(M,U) \text{ and } \mathcal{R}(C[U],V) \text{ implies } \mathcal{R}(C[M],V).$$

A *cost model* for $\mathcal{R}$ is a function $\widetilde{R} : \mathcal{L} \times \mathcal{L} \to \mathbb{N}^\infty$ such that $\mathcal{R}(M,N)$ if and only if $\widetilde{\mathcal{R}}(M,N) < \infty$. $\widetilde{\mathcal{R}}$ is (left) compositional if

$$\widetilde{\mathcal{R}}(M,U) + \widetilde{\mathcal{R}}(C[U],V) \geq \widetilde{\mathcal{R}}(C[M],V).$$

The fundamental requirement of a cost model is to capture the time cost of computing its underlying relation (in an *implementation-independent* way: defining costs for high-level programs in terms of a low-level machine has all of the advantages of theft over honest toil). Framing this as a (semi)decision problem, and referring to the (weak) invariance thesis — *reasonable universal machines can simulate each other up to a polynomial overhead in time* [4] — we may say that a cost model for $\mathcal{R}$ is *reasonable* if there is a procedure which recognizes pairs of terms $(M,V) \in \mathcal{R}$ in time polynomial in $\widetilde{\mathcal{R}}(M,V)$ on any reasonable machine. By the invariance thesis it suffices to implement such a procedure on some reasonable machine: our algorithm for recognizing call-by-value evaluation trees runs on a Random Access Machine with a constant factor overhead.

As well as a potentially useful reasoning principle, compositionality of costs is an optimality property — a triangle inequality requiring the computation of $\mathcal{R}$ to follow the shortest path to a result:



Unsurprisingly, it may be easier to verify such a path (or find it non-deterministically) than to give an effective procedure which computes it. For example, a cost model for the reducibility relation in a term rewriting system is *unitary* if $\widetilde{\mathcal{R}}(M,N)$ is the length of some reduction path from M to N (where one exists). It is *optimal* if this is the shortest such path. A simple

978-1-6654-4895-6/21/$31.00

induction establishes that a unitary cost model is optimal if and only if it is compositional and for all M, $\widetilde{\mathcal{R}}(M, M) = 0$ and $\widetilde{\mathcal{R}}(M, N) = 1$ if $M \to N$ and $M \neq N$.

However, in the λ-calculus, there can be no effective method for finding optimal β-reduction paths (e.g. to normal form [5]). The relationship between optimality and compositionality of (unitary) costs is not so straightforward for β-normalization itself, but *standard* reduction strategies possess neither property, and for the same reason: *duplication of work*. For example, normal order reduction of

$$(\lambda x.x\ x)\ [\lambda y.y\ \lambda z.z]$$

duplicates $\lambda y.y\ \lambda z.z$: it would therefore be cheaper to reduce it first to $\lambda z.z$, and then normalize $(\lambda x.x\ x)\ [\lambda z.z]$ to $\lambda z.z$. More promising therefore is to consider reduction of *sharing graphs* for λ-terms, which eliminates duplication of work by allowing multiple β-redexes to be reduced in parallel, and for which Lamping's algorithm can be used to find optimal reduction paths effectively [6]. However, the cost model for the λ-calculus which counts shared β-reductions (proposed in [7][1]) cannot be reasonable [8], [9]: (in the case of optimal reduction it does not account for the additional "bookkeeping" work on the graphs required to correctly implement sharing). Results such as these confirm that claims about efficiency of evaluation require a reasonable cost model to support them by properly accounting for work done, since naïve measures such as the number of β-reductions performed may not do so. Even in cases where this is a reasonable cost model (e.g. normal order reduction), implementation requires considerable technical sophistication [10], [11], [12], [13].

In fact, functional programs are not generally evaluated by direct implementation of β-reduction, so we may look for compositional cost models satisfied by other forms of evaluation, which must therefore be efficient in this general sense. *Lazy evaluation* — evaluating arguments once, on demand, and sharing the result — is the general remedy for duplication of work by substitution, and thus a basis for implementing a compositional cost model. In its simplest form (call-by-need), $\lambda x.M\ N$ is evaluated by reducing M until x is called, then evaluating N (to a weak head normal form V) and sharing this value with any future calls to x. This avoids the work of repeatedly evaluating N each time x is called: the cost of lazily evaluating $\lambda x.M\ N$ is no more than the cost of evaluating N to V plus that of evaluating $\lambda x.M\ V$. So call-by-need evaluation should satisfy a compositional cost model with respect to *weak* contexts in which the hole is not in the scope of a λ-abstraction (indeed, call-by-need is optimal for weak reduction [14]).

This raises the question of how to formulate an implementation-independent cost model for lazy evaluation, given that the necessary sharing is not expressible in the λ-calculus itself. Hackett and Hutton [15] propose "clairvoyant call-by-value" as a state-free way to specify and reason about the costs of call-by-need as those of a non-deterministic evaluator which always makes the optimal choice between evaluating $\lambda x.M\ N$ by evaluating $M[\bot/x]$, if M does not call x, and evaluating N to a value V and then $M[V/x]$ if it does. To define a formal cost model for λ_v, we take a similar approach: *specifying* it in terms of the optimal cost of a non-deterministic, eager evaluation strategy which can discard unneeded subterms in favour of $\bot$, and *implementing* it with a deterministic, lazy evaluator. Compositionality with respect to all contexts requires that evaluation is strong (under λ-abstractions): our clairvoyant eager strategy performs *innermost* reduction rather than call-by-value, and the lazy evaluator shares between function bodies rather than weak head-normal forms. The main technical challenge is to achieve the latter.

Full laziness [16] is a refinement of laziness that does allow sharing under a λ-abstraction — of those subexpressions which are syntactically independent of it. In other words, it should satisfy a compositionality property: the cost of evaluating $C[M]$ is no more than that of evaluating M to U and then evaluating $C[U]$, *provided no free variable of M is bound by $C[\_]$*. However, a subexpression may be semantically independent of its context — and thus capable of being shared — without being syntactically independent of it: for example, fully lazy evaluation of

$$(\lambda w.w\ w)\ \lambda x.[(\lambda f.f\ x)\ \lambda y.\lambda z.z]$$

duplicates $\lambda f.(f\ x)\ \lambda y.\lambda z.z$, so its cost is greater than that of evaluating $\lambda f.(f\ x)\ \lambda y.\lambda z.z$ to $\lambda z.z$ and $(\lambda w.w\ w)\ \lambda x.[\lambda z.z]$ to $\lambda z.z$. Compositionality with respect to all contexts therefore requires sharing at a deeper, semantic level.

Complete laziness was proposed by Holst and Gomard [17] as a level of sharing giving the run-time evaluator the efficiency of *partial evaluation* — i.e. the ability to selectively evaluate subterms. In effect, this is equivalent to defining a completely lazy evaluator to be one which satisfies a compositional cost model. Indeed, formalizing complete laziness in this way should shed light upon a concept which is intriguing but not yet well understood. The explanations in [17] of how it is to be achieved are sketchy. Moreover, although both [17] and [18] (which proposes a more detailed semantics) present complete laziness as "fuller laziness", subsuming call-by-need and full laziness, their proposed semantics omits some sharing that is achieved by plain call-by-need[2]: sharing at the "deeper" level of function bodies does not imply sharing at the level of weak head normal forms. Thyer [19] describes an implementation of complete laziness which combines both, subsuming call-by-need (but not full laziness), at the price of a substantial increase in the complexity of the graph-manipulations required (substitutions of graphs into each other, implemented using memo-tables), making it difficult to give an implementation-independent characterization of its

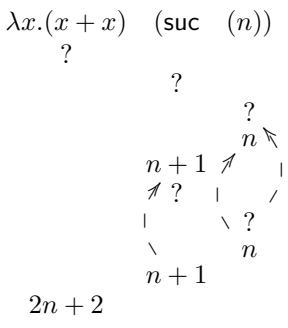
[1]This cost model includes the size of the term and its normal form but the resuts of [8], [9] still hold.

[2]The counterexample $(\lambda f.(f\ \lambda w.w))\ \lambda x.((\lambda y.y\ y)\ (x\ \lambda z.z))$ is given in [18] as an example for which completely lazy evaluation is not optimal. However, call-by-need does achieve optimal sharing in this case.

costs.[3] However, this implementation is shown in practice to *collapse a tower of interpreters*, which constitutes an empirical demonstration that its costs are compositional. Consider, for example, a program $\lambda x.M\,(\mathcal{I}\,N)$, where $\mathcal{I}$ is some higher-order program transformation such as an interpreter. The compositionality principle says that the cost overhead of doing the interpretation at run-time is no more than the cost of evaluating $\mathcal{I}\,N$ once to produce the finite amount of interpreted code consumed by M, however many times x is called. Indeed, a tower of interpreters $\lambda x.M\,(\mathcal{I}_1\,(\mathcal{I}_2\ldots(\mathcal{I}_n\,N)\ldots))$ should collapse to a constant overhead over running fully interpreted code, and this is confirmed by the experimental results in [19].

A. Lazy Implementation of Game Semantics

Our cost model is derived from a compositionally defined denotational model, and its implementation from an effective procedure for lazily computing denotations. Game semantics is a natural setting in which to apply this methodology: it captures low-level implementation details in an abstract, compositional setting, demonstrated by correspondences with more explicit models of computation such as the Krivine abstract machine [20]. The length of interactions between strategies can be used to to give a cost analysis of evaluating the programs which denote them in such an implementation [21]. However, this is not a compositional cost model[4]. The fundamental problem is the inefficiency of computing the composition of innocent strategies (the denotations of purely functional programs in Hyland-Ong games [23]) by playing them off against each other ("parallel composition plus hiding"): such plays will typically contain moves — and whole subsequences — which are copies of previous parts of the play because both players are constrained by innocence to play the same move in response to positions with the same view. For example, computing the (call-by-name) denotation of $\lambda x.(x+x)\,\mathsf{suc}(n)$ corresponds to computing the following sequence of moves :



(and hiding all but the first and last). The arrows show moves which are required by innocence to be copies of a previous move *and could therefore be shared*. This problem (duplication of work) is essentially the same as for β-reduction. However, by representing functional computation as a sequence of atomic interactions, game semantics allows the solution (sharing by graph reduction) to be implemented more simply, without bookkeeping overhead (as in combinator graph reduction [24], but preserving λ-calculus structure and respecting an implementation-independent cost model).

λ_v-calculus (Section II)
⇓
CPS Interpretation (Section III)
⇓
Explicit Substitutions (Section IV)
⇓
Nameless Representation (Section V)
⇓
Graph Reduction (Section VI)
⇓
RAM Implementation (Section VII)

Fig. 1. Reduction steps for implementation of λ_v

How can the computation of a denotational semantics be captured operationally? One possibility is *normalization by evaluation* [25]: computing the denotation of a term as the element of a datatype in a high-level language, which can then be *reified* back to its normal form. Implementing this lazily allows infinitary normal forms (Böhm trees) to be computed [26], [27], and has been used to describe the composition of innocent strategies [28]. This is potentially a form of complete laziness, because higher-order functions are represented as shareable elements of lazy datatypes. The key difference to our approach is that it is an *interpretation* of one high-level language within another, rather than a *compilation* to run on a low-level (reasonable) machine. Its efficient implementation, and the analysis of costs, therefore depends on the "kindness of strangers" — the compiler for the target language.

We shall take an unmediated approach, computing their *evaluation trees* directly from λ-terms. These are a generalized form of Böhm tree, and the syntactic counterparts to innocent strategies [23]: defining a compositional interpretation which assigns to each term M its evaluation tree thus amounts to a form of game semantics [29]. Doing so effectively — computing *with* Böhm trees [30] — gives an operational semantics for functional programs which subsumes evaluation to head-normal form by computing as much of the tree as is required. Evaluation trees are most simply represented as limits (ideals) of their finite approximants: we give a compositional natural semantics for the call-by-value λ-calculus in the form of a system for deriving judgments of the form "a is an approximant to (the evaluation tree of) A". The main challenge lies in giving an effective semi-decision procedure for this relation with cost linear in the size of derivations. This is achieved by a sequence of (linear-time) reductions to an implementation on a Random Access Machine (Figure 1), using variants of several well-known compilation techniques and in particular, a graph representation which allows completely lazy computation of trees/strategies.

[3]Completely lazy implementation fits well with call-by-value, as we show: this may seem paradoxical but illustrates the distinction between a semantics (such as call-by-name or call-by-value) and an evaluation technique for computing it, such as call-by-need or complete laziness.

[4]See recent work characterizing its (in)efficiency [22]

II. A Compositional Cost Model for λ_v

In this section we describe a natural ("big-step") evaluation tree semantics for the call-by-value λ-calculus and prove compositionality of its intrinsic cost model. We will present this semantics via an *approximation relation* on λ_v-terms.

Definition 2.1: An approximation relation on a set $\mathcal{S}$ is a binary relation $\propto$ on $\mathcal{S}$ such that:

- If $a \propto A$ for some $A \in \mathcal{S}$ then $a \propto a$.
- $a, b \propto A$ if and only if $\exists c \propto A$ such that $a, b \propto c$.

In other words, $\propto$ restricts to a preorder on the set $\{a \in \mathcal{S} \mid \exists A. a \propto A\}$ of *approximants* for $\propto$[5], and for each $A \in \mathcal{S}$ the set $\{a \in \mathcal{S} \mid a \propto A\}$ of approximants to A is an ideal of this preorder.

This is general enough to describe evaluation trees for objects such as graphs (see Section 6), but typically we will define an evaluation tree semantics for a programming language (with a distinguished constant $\bot$) by giving an approximation relation on its set of terms: the approximants correspond to the finite trees, and $a \propto A$ if the evaluation tree of A may be pruned to a by replacing some of its subtrees with $\bot$. The most direct way to define such a relation is via a natural presentation — i.e. an inference system (in the sense of Aczel [31]) for deriving judgements of the form $a \propto A$, consisting of a set of rules for which these judgments are the premisses and conclusions and the approximation relation is the least fixed point.

To simplify the presentation of the semantics of the call-by-value λ-calculus we adopt a slightly elaborated syntax making the flow of control more explicit. (A type of *administrative normal form* [32].

Definition 2.2: The terms of (partial) λ_v are given by the grammar: $M ::= \bot \mid V \mid V\ V \mid \mathsf{let}\ x = M\ \mathsf{in}\ M$
where values V are given by the grammar: $V ::= x \mid \lambda x.M$. So we may express $M\,N$ as either $\mathsf{let}\, x = M\, \mathsf{in}\, \mathsf{let}\, y = N\, \mathsf{in}\, (x\ y)$ or $\mathsf{let}\, y = N\, \mathsf{in}\, \mathsf{let}\, x = M\, \mathsf{in}\, (x\ y)$ — terms which may have distinct evaluation trees.

The approximation relation $\propto_v$ on λ_v-terms is the least fixed point of the rules in Table I. Here, and elsewhere, the substitution $a[c/x]$ of c for free occurrences of x in a is considered well-defined only if no free variable of c becomes bound — the rules for the approximation relation allow for explicit renaming of bound variables to avoid such capture. An example derivation of $\lambda x.\lambda y.\bot \propto_v \lambda x.\mathsf{let}\ z = \lambda u.\lambda v.\bot\ x\ \mathsf{in}\ z$ is given in Figure 2.

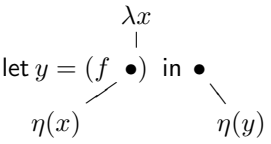
Implicit in the derivation rules there is a "clairvoyant" (in the sense of [15]) algorithm for computing the approximants to a term compositionally (from the inside out) and semideterministically — the rule required to infer $a \propto_v A$ is determined by the outermost constructors of A, except where $a = \bot$.

It is straightforward to show that approximants are given by the grammar: $a ::= \bot \mid \lambda x.a \mid \mathsf{let}\ x = (y\ \lambda z.a)\ \mathsf{in}\ a$
and that if a and b are approximants then $a \propto_v b$ if and only $a \sqsubseteq_\bot b$ up to renaming of bound variables, where $\sqsubseteq_\bot$ is the least precongruence on terms such that $\bot \sqsubseteq_\bot M$ for all M. Thus $\propto_v$ is a preorder on the set of approximants, and the equivalence relation it induces on approximants ($a \propto_v b$ and $b \propto_v a$) is α-equivalence in the usual sense. Using this characterization of finite approximants, together with the fact that substitution is $\sqsubseteq_\bot$-monotone — i.e. if $a \sqsubseteq_\bot a''$ and $v \sqsubseteq_\bot v''$ then $a[v/x] \sqsubseteq_\bot a''[v''/x]$ — we may prove the following lemma by induction on derivation size:

Lemma 2.3: If $a \propto_v A$ and $a' \propto_v A'$, where $A, A' \sqsubseteq_\bot A''$ then there exists $a'' \propto_v A''$ such that $a, a' \propto_v a''$.

Hence $\propto_v$ is a well-defined approximation relation.

This notion of evaluation tree for λ_v identifies strictly more terms than the "call-by-value Böhm trees" defined in [33]: specifically, any variable $f \neq x$ with $\lambda x.f\ x$ (η_v-equivalence), and $f\ V$ with $\mathsf{let}\ y = (f\ V)\ \mathsf{in}\ y$. So, for example, the evaluation tree of f is the infinite tree $\eta(f) =$:



Both of these equations are sound with respect to the denotational semantics of λ_v (in a closed Freyd category with a reflexive object $D \cong D \Rightarrow D$). Indeed, the (α-equivalence classes of) closed approximants are order-isomorphic (via a proof of definability) to the finite innocent strategies on such an object in a category of games, defined in [34] as an "intensionally fully abstract" model of the call-by-value λ-calculus, and thus their ideals — evaluation trees — correspond to innocent strategies: (the node at the end of each branch of the tree is the response by the strategy to the "view" corresponding to the branch, either with a *question* ($\mathsf{let}\ x = y\ \lambda z.\_\ \mathsf{in}\ \_$) with a pointer ($y$) back to its binder or an *answer* ($\lambda z.\_$). Our natural semantics may be understood as an alternative (but still compositionally defined) presentation of this model, similar to the presentation of the HO games model of PCF in [35].

A. A Compositional Cost Model

A cost model for an approximation relation $\propto$ on a set $\mathcal{S}$ is a map $\widetilde{\propto} : \mathcal{S} \times \mathcal{S} \to \mathbb{N}^\infty$ such that $\propto(a, A) < \infty$ iff $a \propto A$. We assign to $\propto_v$ its *intrinsic cost model*:

Definition 2.4: If $a \propto_v A$ is derivable, let $\widetilde{\propto_v}(a, A)$ be the size[6] of its minimal derivation. Otherwise, $\widetilde{\propto_v}(a, A) = \infty$.

This is monotone in its first argument — $a \propto_v b$ implies $\widetilde{\propto_v}(a, A) \leq \widetilde{\propto_v}(b, A)$. It is also compositional:

Proposition 2.5: For any terms a, b, B and context $C[\_]$, $\widetilde{\propto_v}(a, C[B]) \leq \widetilde{\propto_v}(b, B) + \widetilde{\propto_v}(a, C[b])$.

PROOF: This is evident if either $\widetilde{\propto_v}(a, C[b]) = \infty$ or $\widetilde{\propto_v}(b, B) = \infty$, so assume that $a \propto_v C[b]$ and $b \propto_v B$ and proceed by structural induction on $C[\_]$.

If $C[\_]$ is the empty context then $\widetilde{\propto_v}(a, C[B]) \leq \widetilde{\propto_v}(b, B) \leq \widetilde{\propto_v}(a, C[b]) + \widetilde{\propto_v}(b, B)$ by monotonicity of $\widetilde{\propto_v}(\_, B)$.

[5]We will follow a convention of using lower case for approximants and upper case for general terms, graphs etc.

[6]Some rules (e.g. the axiom $\bot \alpha_v M$) need not be counted — but this is at most a constant factor of derivation size.

$$\frac{}{\bot \propto_v M} \qquad \frac{a \propto_v M}{\lambda y.a[y/x] \propto_v \lambda x.M}\, y \notin \mathrm{FV}(a) \qquad \frac{a \propto_v \lambda y.x\ y}{a \propto_v x}\, y \neq x \qquad \frac{a \propto_v V \quad b \propto_v M \quad c \propto_v b[a/x]}{c \propto_v \mathsf{let}\ x{=}V\ \mathsf{in}\ M} \qquad \frac{a \propto_v \mathsf{let}\ y{=}M\ \mathsf{in\ let}\ x{=}M'\ \mathsf{in}\ N}{a \propto_v \mathsf{let}\ x{=}\mathsf{let}\ y{=}M\ \mathsf{in}\ M'\ \mathsf{in}\ N}$$

$$\frac{a \propto_v \mathsf{let}\ x{=}U\ V\ \mathsf{in}\ x}{a \propto_v U\ V} \qquad \frac{a \propto_v V \quad b \propto_v M}{\mathsf{let}\ y{=}f\ a\ \mathsf{in}\ b[y/x] \propto_v \mathsf{let}\ x{=}f\ V\ \mathsf{in}\ M}\, y \notin \mathrm{FV}(b) \qquad \frac{a \propto_v M \quad b \propto_v V \quad c \propto_v N \quad d \propto_v \mathsf{let}\ x{=}a[b/y]\ \mathsf{in}\ c}{d \propto_v \mathsf{let}\ x{=}\lambda y.M\ V\ \mathsf{in}\ N}$$

TABLE I
EVALUATION TREE SEMANTICS FOR λ_v

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\bot \propto_v \bot}{\lambda v.\bot \propto_v \lambda v.\bot} \quad \dfrac{\bot \propto_v x\ w}{\lambda w.\bot \propto_v \lambda w.x\ w}}{\lambda w.\bot \propto_v x} \quad \dfrac{\bot \propto_v z\ y}{\lambda y.\bot \propto_v \lambda y.z\ y}}{\lambda y.\bot \propto_v z} \quad \dfrac{\bot \propto_v \bot}{\lambda v.\bot \propto_v \lambda v.\bot} \quad \dfrac{\bot \propto_v \bot}{\lambda y.\bot \propto_v \lambda y.\bot} \quad \dfrac{\bot \propto_v \bot}{\lambda y.\bot \propto_v \lambda y.\bot}}{\lambda y.\bot \propto_v \mathsf{let}\ z = \lambda v.\bot\ \mathsf{in}\ \lambda y.\bot}}{\lambda y.\bot \propto_v \mathsf{let}\ z = \lambda u.\lambda v.\bot\ x\ \mathsf{in}\ z}}{\lambda x.\lambda y.\bot \propto_v \lambda x.\mathsf{let}\ z = \lambda u.\lambda v.\bot\ x\ \mathsf{in}\ z}$$

Fig. 2. Derivation of $\lambda x.\lambda y.\bot \propto_v \lambda x.\mathsf{let}\ z = \lambda u.\lambda v.\bot\ x\ \mathsf{in}\ z$

For the induction step there is a range of similar cases depending on the structure of $C[\_]$, we give a typical one: suppose $C[\_] \equiv \mathsf{let}\ x = \lambda y.C'[\_]\ \mathsf{in}\ N$ (and $a \neq \bot$, since $\widetilde{\propto_v}(\bot, C[M]) = 1 \leq \widetilde{\propto_v}(b, B)$ for all b).

Since $a \propto_v C[b]$ has a smallest derivation, there exist $a' \propto_v \lambda y.C'[b]$ and $c \propto_v N$ such that $a \propto_v c[a'/x]$ and $\widetilde{\propto_v}(a, C[b])$ $= \widetilde{\propto_v}(a', \lambda y.C'[b]) + \widetilde{\propto_v}(c, N) + \widetilde{\propto_v}(a, c[a'/x]) + 1 \qquad (*)$

We claim (†) that $\widetilde{\propto_v}(a, C[B])$
$\leq \widetilde{\propto_v}(a', \lambda y.C'[B]) + \widetilde{\propto_v}(c, N) + \widetilde{\propto_v}(a, c[a'/x]) + 1. \leq$
$\widetilde{\propto_v}(b, B) + \widetilde{\propto_v}(a', \lambda y.C'[b]) + \propto (c, N) + \widetilde{\propto_v}(a, c[a'/x]) + 1$
by induction hypothesis applied to $\lambda y.C'[\_]$
$= \widetilde{\propto_v}(b, B) + \widetilde{\propto_v}(a, C[b])$ by $(*)$, as required.

To justify the claim (†): if $\widetilde{\propto_v}(a', \lambda y.C'[B]) = \infty$ then it is immediate. Otherwise, $a' \propto_v \lambda y.C'[B]$, and so there is a derivation of $a \propto_v C[B]$ of size $\widetilde{\propto_v}(a', \lambda y.C'[B]) + \widetilde{\propto_v}(c, N) + \widetilde{\propto_v}(a, c[a'/x]) + 1$ and $\widetilde{\propto_v}(a, C[B])$ is less than this by definition. □

We now give an example of the application of compositionality to analyse a sequence of terms $A_1, A_2, \ldots$ for which the cost of proving termination of A_n is linear in n in our cost model, but requires exponentially many (leftmost) outermost reductions. This example is adapted from a call-by-name setting [7] where it is also shown that the cost of evaluating each A_n using either combinators or supercombinators [36] is exponential in n (nor does full laziness give any speedup [18]).

Definition 2.6: $A_0 \triangleq \lambda y.\bot$ and for each $n \in \omega$, $A_{n+1} \triangleq$ $\mathsf{let}\ w = \lambda x.A_n\ \mathsf{in\ let}\ z = w\ x\ \mathsf{in}\ w\ z$.
Writing $M \overset{k}{\longrightarrow\!\!\!\!\!\rightarrow} M'$ if there is a normal order reduction sequence of length k from M to M' consisting of β_v-reductions (let $x = V\ \mathsf{in}\ N \to N[V/x]$ and $\lambda x.N\ V \to N[V/x]$):

Proposition 2.7: $A_n \overset{2^{n+2}-4}{\longrightarrow\!\!\!\!\!\rightarrow} \lambda y.\bot$.

PROOF: This follows from the fact that for any value V, $\lambda x.A_n\ V \overset{2^{n+2}-3}{\longrightarrow\!\!\!\!\!\rightarrow} \lambda y.\bot$, which we prove by induction on n.
At $n = 0$, $\lambda x.\lambda y.\bot\ V \overset{1}{\longrightarrow\!\!\!\!\!\rightarrow} \lambda y.\bot$. For the induction step:
$\lambda x.A_{n+1}\ V \overset{1}{\longrightarrow\!\!\!\!\!\rightarrow} \mathsf{let}\ w = \lambda x.A_n\ \mathsf{in\ let}\ z = w\ V\ \mathsf{in}\ w\ z$
$\overset{1}{\longrightarrow\!\!\!\!\!\rightarrow} \mathsf{let}\ z = (\lambda x.A_n\ V)\ \mathsf{in}\ (\lambda x.A_n\ z)$
$\overset{2^{n+2}-3}{\longrightarrow\!\!\!\!\!\rightarrow} \mathsf{let}\ z = \lambda y.\bot\ \mathsf{in}\ (\lambda x.A_n\ z)$ by induction hypothesis
$\overset{1}{\longrightarrow\!\!\!\!\!\rightarrow} \lambda x.A_n\ \lambda y.\bot \overset{2^{n+2}-3}{\longrightarrow\!\!\!\!\!\rightarrow} \lambda y.\bot$ by induction hypothesis.
Hence $\lambda x.A_{n+1}\ V \overset{2^{n+3}-3}{\longrightarrow\!\!\!\!\!\rightarrow} \lambda y.\bot$ as required. □

The proof that the size of the derivation of $\lambda y.\bot \propto_v A_n$ is linear in n is a simple application of compositionality.

Proposition 2.8: $\widetilde{\propto_v}(A_0, A_n) \leq n.\widetilde{\propto_v}(A_0, A_1)$ for $n > 0$.

PROOF: By induction on n: at $n = 1$ this holds by definition. For the induction step, $\widetilde{\propto_v}(A_0, A_{n+1})$
$\triangleq \widetilde{\propto_v}(A_0, \mathsf{let}\ w = \lambda x.A_n\ \mathsf{in\ let}\ z = w\ x\ \mathsf{in}\ w\ z)$
$\leq \widetilde{\propto_v}(A_0, A_n) + \widetilde{\propto_v}(A_0, \mathsf{let}\ w = \lambda x.A_0\ \mathsf{in\ let}\ z = w\ x\ \mathsf{in}\ w\ z)$
by compositionality
$\leq n.\widetilde{\propto_v}(A_0, A_1) + \widetilde{\propto_v}(A_0, A_1)$ by induction hypothesis
$= (n+1).\widetilde{\propto_v}(A_0, A_1)$ as required. □

We leave it as an exercise to find a bound for $\widetilde{\propto_v}(A_0, A_1)$ using the compositionality property.

III. COST REDUCTIONS

It remains to establish that $\widetilde{\propto_v}$ is reasonable by implementing it on a reasonable machine. We will show that there is a constant K and a Random Access Machine which takes a (linear time computable) compilation of pairs (a, A) as its initial states and accepts within $K.\widetilde{\propto_v}(a, A)$ steps if $a \propto_v A$ and either rejects or diverges otherwise.

To bridge the gap between the high-level features of λ_v and their low-level implementation on the RAM, we adopt the standard compilation strategy of breaking it down into a sequence of translations into progressively lower-level intermediate languages. We give an evaluation tree cost model for each language, and show that each compilation step is a cost reduction in the following sense.

Definition 3.1: A (reasonable) *cost reduction* between cost-models $\widetilde{\propto_1} : \mathcal{L}_1 \times \mathcal{L}_1 \to \mathbb{N}^\infty$ and $\widetilde{\propto_2} : \mathcal{L}_2 \times \mathcal{L}_2 \to \mathbb{N}^\infty$ is a function $f : \mathcal{L}_1 \to \mathcal{L}_2$ (computable in time polynomial in input size) such that for some polynomial ϕ, $\widetilde{\propto_2}(f(a), f(A)) \leq^\infty \phi(\widetilde{\propto_1}(a, A))$ for all $a, A \in L_1$, where $x \leq^\infty y$ if $x, y \in \mathbb{N}$ and $x \leq y$, or $x = y = \infty$. We may write this as $\widetilde{\propto_2} \lesssim f(\widetilde{\propto_1})$.

Evidently, if $\widetilde{\propto_2} \lesssim f(\widetilde{\propto_1})$ and $\widetilde{\propto_2}$ is reasonable, then so is $\widetilde{\propto_1}$. The sequence of cost reductions from λ_v to its implementation is shown in Figure 1. Each is in fact linear — in terms of both compilation time and run-time overhead — and thus maintains a constant factor overhead. The key step is *determinization* of the algorithm for computing approximants by reduction to a graph representation in which we can define a deterministic call-by-need graph reduction strategy with the same cost as clairvoyant innermost evaluation. The other reductions prepare for this transformation by breaking down the high-level features of λ_v — simplification of control flow by CPS transform, replacement of meta-level with explicit substitutions, and the elimination of name binding by conversion into a nameless representation using de Bruijn levels.

A. Continuation-passing-style Interpretation

The first example of a cost reduction, and the first step in compiling λ_v into its RAM implementation, is a familiar one in the context of compilation — continuation-passing-style (CPS) translation into a calculus (λ_{CPS}) which has a simpler semantics because its functions never return a value. We define a (compositional) evaluation tree semantics for λ_{CPS} and a Plotkin-style CPS interpretation of λ_v which is a linear-time cost reduction with respect to their intrinsic cost models. In game semantics terms, this corresponds to embedding into a category of continuations with the one-move game as answer object (or dropping the distinction between questions and answers).

Each term of λ_{CPS} is (exclusively) either a non-returning *computation* or a *value*: a function taking an argument consisting of a pair of values and returning a computation.

Definition 3.2: The sets of computations and values of λ_{CPS} are given by the grammars:

$M ::= \bot \mid V\,\langle V, V\rangle \qquad\qquad V ::= \lambda x.M \mid x.t$

where x ranges over a set of variables and t over the set $\{l, r\}$. So each variable represents a pair of values, and is invoked with a tag for left or right projection. We write $M[\langle V_l, V_r\rangle/x]$ for the substitution which replaces each free occurrence of $x.l$ in M with V_l and each free occurrence of $x.r$ with V_r, provided this is non-capturing.

The natural evaluation tree semantics for λ_{CPS} is given by an approximation relation ($\propto_{\mathsf{c}}$) on the set of computations and values — the least fixed point of the big-step derivation rules for the judgement $a \propto_{\mathsf{c}} A$ given in Table II.

It is straightforward to show that the sets of (computation and value) λ_{CPS} approximants are generated by the grammars:

$a ::= \bot \mid x.t\,\langle v, v\rangle \qquad\qquad v ::= \lambda x.a$

The closed evaluation trees of λ_{CPS} are therefore binary trees in which each edge is labelled with a λ-abstraction and each node with a left/right tag and a pointer (its head variable) back to some earlier edge where it was bound. They correspond to innocent strategies for the HO arena in which players take turns playing either a right or left tag with a pointer to a previous move by the other player (the least fixed point $D = \neg(D \times D)$, where $\neg D$ flips the roles of the two players and adds an initial move). The evaluation tree presentation of this semantics (and thus its cost model) is compositional (the proof follows that of Proposition 2.5), and we shall show that it is reasonable. Thus CPS interpretation from any language into λ_{CPS} is the basis of a compositional cost model, providing it is compositional (i.e. $\overline{C[M]} = \overline{C}[\overline{M}]$) and $\sqsubseteq_\perp$-monotone.

$$\frac{}{\bot \propto_{\mathsf{c}} M} \qquad \frac{a \propto_{\mathsf{c}} M}{\lambda y.a[y/x] \propto_{\mathsf{c}} \lambda x.M}\, y \notin \mathrm{FV}(a)$$

$$\frac{a_l \propto_{\mathsf{c}} y.l \quad a_r \propto_{\mathsf{c}} y.r}{\lambda y.x.t\,\langle a_l, a_r\rangle \propto_{\mathsf{c}} x.t} \qquad \frac{a_l \propto_{\mathsf{c}} V_l \quad a_r \propto_{\mathsf{c}} V_r}{x.t\,\langle a_l, a_r\rangle \propto_{\mathsf{c}} x.t\,\langle V_l, V_r\rangle}$$

$$\frac{a \propto_{\mathsf{c}} M \quad b_l \propto_{\mathsf{c}} V_l \quad b_r \propto_{\mathsf{c}} V_r \quad c \propto_{\mathsf{c}} a[\langle b_l, b_r\rangle/x]}{c \propto_{\mathsf{c}} \lambda x.M\,\langle V_l, V_r\rangle}$$

TABLE II
NATURAL EVALUATION TREE SEMANTICS FOR λ_{CPS}

The compilation of λ_v into λ_{CPS} is a variant of Plotkin's call-by-value CPS translation [37]. We define mutually inductive translations sending each value V of λ_v to a value V^* of λ_{CPS}, and each term M of λ_v to a context $\overline{M}[\bullet]$ of λ_{CPS} (a computation with one value-shaped hole, or none).

$x^* = x.l \qquad\qquad \overline{\bot} = \bot$

$(\lambda x.M)^* = \lambda x.\overline{M}[x.r] \qquad \overline{V\,U} = V^*\,\langle U^*, \bullet\rangle$

$\overline{V} = \bullet\,\langle V^*, \lambda y.\bot\rangle \qquad \overline{\mathsf{let}\ y = M\ \mathsf{in}\ N} = \overline{M}[\lambda y.\overline{N}[\bullet]]$

E.g. (compare Figure 2) $(\lambda x.\mathsf{let}\ z = \lambda u.\lambda v.\bot\ x\ \mathsf{in}\ z)^* = \lambda x.\lambda u.u.r\,\langle \lambda v.\bot, \lambda y.\bot\rangle\,\langle x.l, \lambda z.x.r\,\langle z_l, \lambda y.\bot\rangle\rangle$.

Observe that if v is a λ_v-approximant value then v^* is a λ_{CPS} approximant, and if a is a λ_v-approximant term then $\overline{a}[x.r]$ is a λ_{CPS} approximant. There is a constant factor increase in the size of translated terms (e.g. from $\lambda x.M$ to $\bullet\,\langle \lambda x.\overline{M}[x.r], \lambda y.\bot\rangle$ but while other CPS translations can introduce administrative reductions with an inflationary effect on evaluation cost, the natural semantics for λ_v and λ_{CPS} track each other through CPS interpretation(by design) — we show by induction on derivation that:

Lemma 3.3: $\widetilde{\propto_{\mathsf{c}}}(\overline{a}[\kappa.r], \overline{M}[\kappa.r]) \leq^{\infty} 4.\widetilde{\propto_v}(a, M)$ for terms a, M, and $\widetilde{\propto_{\mathsf{c}}}(b^*, V^*) \leq^{\infty} 4.\widetilde{\propto_v}(b, V)$ for values b, V.
where κ is an unbound value. Thus:

Proposition 3.4: $\widetilde{\propto_{\mathsf{c}}} \lesssim \widetilde{\widetilde{\propto_v}}$.

IV. EXPLICIT SUBSTITUTIONS

Our next cost reduction is a compilation of λ_{CPS} to a CPS calculus with *explicit substitutions* ($\lambda\sigma_{\mathsf{CPS}}$). An explicit treatment of the substitution operation [38] is a well-established step towards implementation of λ-calculi. It also enables a correspondingly more exact reflection of evaluation cost — see Remark 4.3. We will use explicit substitutions to implement a form of *linear head reduction*, which has been used to analyse the dynamics of game semantics interaction [20].

Definition 4.1: Terms (computations and values, respectively) of $\lambda\sigma_{\mathsf{CPS}}$ are given by the grammar:

$M ::= \bot \mid x.t\,\langle V, V\rangle \mid M[x := \langle V, V\rangle]$

$V ::= \underline{x.t} \mid \lambda x.M$

where x ranges over the set of variables and t over $\{l, r\}$.
We will define a "small-step" evaluation tree semantics of $\lambda\sigma_{\mathsf{CPS}}$ (bringing us closer to implementation on an abstract machine)

$$
\begin{array}{rcl}
x.t\,\langle \lambda z.M_l, \lambda z.M_r\rangle[x := \langle \lambda w.N_l, \lambda w.N_r\rangle] & \to & N_t[w := \langle \lambda z.M_l[x := \langle \lambda w.N_l, \lambda w.N_r\rangle], \lambda z.M_r[x := \langle \lambda w.N_l, \lambda w.N_r\rangle]\rangle] \\
y.t\,\langle \lambda z.M_l, \lambda z.M_r\rangle[x := \langle V_l, V_r\rangle] & \to & y.t\,\langle \lambda z.M_l[x := \langle V_l, V_r\rangle], \lambda z.M_r[x := \langle V_l, V_r\rangle]\rangle \quad (x \neq y) \\
\underline{x.t} & \to & \lambda y.x.t\,\langle \underline{y.l}, \underline{y.r}\rangle \quad (y \neq x)
\end{array}
$$

Fig. 3. Reduction rules for Explicit Substitutions

— its approximation relation is derived from the rewriting rules for explicit substitutions given in Figure 3 (which are well-defined only where reduction is non-capturing, i.e. z must not be free in N_l, N_r). Note that terms of $\lambda\sigma_{\mathsf{CPS}}$ do not contain β-redexes. Whereas substitution of a λ-abstraction for a variable applied to an argument creates a β-redex, the reduction rule for its explicit substitution implicitly elides the reduction of this redex, creating another explicit substitution.

The approximation relation $\propto_{\lambda\sigma}$ on $\lambda\sigma_{\mathsf{CPS}}$ is defined by rewriting terms to their approximants, which are given explicitly as the sets generated by the grammars:

$a ::= \bot \mid x.t\,\langle v, v\rangle \qquad v ::= \lambda x.a$

So the sets of approximants (and hence evaluation trees)for $\propto_{\mathsf{c}}$ and $\propto_{\lambda\sigma}$ are the same.

To define a compositional semantics, approximation is computed using *innermost* reductions: the above rules are applied only to subterms which do not contain other redexes. On its own, this is insufficient to reach all approximants (a subterm which is discarded by outermost reduction may contain infinitely unfolding innermost reductions). So terms are reduced up to the preorder $\sqsubseteq^{\alpha}_{\bot}$ (the least precongruence on terms such that $\bot \sqsubseteq_{\bot} M$ for all computation terms M, closed under α-equivalence) rather than just α-equivalence.

Definition 4.2: We write $A \Rightarrow B$ if there exists $C[\_], A', B'$ such that $C[A'] \sqsubseteq^{\alpha}_{\bot} A$ and $B \equiv C[B']$ where $A' \to B'$ is an innermost reduction — i.e. if $A' \equiv C'[A'']$ such that $A'' \to B''$ for some B'' then $C'[\_] \equiv [\_]$.

$a \propto_{\sigma} A$ if a is an approximant and there is an innermost reduction sequence $A \Rightarrow^{*} A'$ such that $a \sqsubseteq^{\alpha}_{\bot} A'$.

Let $\widehat{\propto_{\lambda\sigma}}$ be the unitary cost model for $\propto_{\lambda\sigma}$ — i.e. $\widehat{\propto_{\lambda\sigma}}(a, A)$ is the length of the shortest innermost reduction sequence from A to A' such that $a \sqsubseteq^{\alpha}_{\bot} A'$. We take as our intrinsic cost model for $\propto_{\lambda\sigma}$ this "internal cost" of reducing A, plus the size (number of constructors) of the approximant a — $\widetilde{\propto_{\lambda\sigma}}(a, A) = \widehat{\propto_{\lambda\sigma}}(a, A) + |a|$

Remark 4.3: The unitary cost model for $\lambda\sigma_{\mathsf{CPS}}$ satisfies a stronger form of the compositionality property: for some approximant it is an equality.

Proposition 4.4: For any terms a, B and context $C[\_]$ there exists b such that $\widehat{\propto_{\lambda\sigma}}(a, C[b]) + \widehat{\propto_{\lambda\sigma}}(b, B) \leq \widehat{\propto_{\lambda\sigma}}(a, C[B])$.

PROOF: By induction on $\widehat{\propto_{\lambda\sigma}}(a, C[B])$. At the base case B is an approximant so let $b \triangleq B$. Otherwise $C[B] \Rightarrow A$, where $\widehat{\propto_{\lambda\sigma}}(a, A) = \widehat{\propto_{\lambda\sigma}}(a, C[B]) - 1$, so there exists $C'[\_] \sqsubseteq_{\bot} C[\_]$ and $B' \sqsubseteq_{\bot} B$ such that $C'[B'] \to A$. Since this is an innermost reduction, either $A \equiv C'[B'']$, where $B' \to B''$ — and by induction hypothesis there exists b such that $\widehat{\propto_{\lambda\sigma}}(a, C'[b]) + \widehat{\propto_{\lambda\sigma}}(b, B'') \leq \widehat{\propto_{\lambda\sigma}}(a, C'[B''])$ — or $A \equiv C''[B']$, where $C'[\_] \to C''[\_]$ — so by inductive hypothesis there exists b such that $\widehat{\propto_{\lambda\sigma}}(a, C''[b]) + \widehat{\propto_{\lambda\sigma}}(b, B') \leq \widehat{\propto_{\lambda\sigma}}(a, C''[B'])$. In either case, $\widehat{\propto_{\lambda\sigma}}(a, C[b]) + \widehat{\propto_{\lambda\sigma}}(b, B) \leq \widehat{\propto_{\lambda\sigma}}(a, C[B])$ as required. □

The further reductions to the intrinsic RAM cost model will preserve this this property, which is is a counterpart for (and implies) continuity of the evaluation tree semantics (originally observed for Böhm trees by Wadsworth [39] and Hyland [40]) — i.e. if $a \propto_{\lambda\sigma} C[B]$ there exists a finite approximant $b \propto_{\lambda\sigma} B$ such that $a \propto_{\lambda\sigma} C[b]$.

The cost reduction $(\_)$ from λ_{CPS} to $\lambda\sigma_{\mathsf{CPS}}$ simply contracts all β-redexes to explicit substitutions: we define by induction:

$$
\underline{\bot} \triangleq \bot \quad \underline{x.t} \triangleq \underline{x.t} \quad \underline{\lambda x.M} \triangleq \lambda x.\underline{M}
$$
$$
\underline{\lambda x.M\,\langle V_l, V_r\rangle} \triangleq \underline{M}[x := \langle \underline{V_l}, \underline{V_r}\rangle] \quad \underline{x.t\langle V_l, V_r\rangle} \triangleq x.t\langle \underline{V_l}, \underline{V_r}\rangle
$$

E.g. (our running example): $\underline{(\lambda x.\mathsf{let}\ z = \lambda u.\lambda v.\bot\ x\ \mathsf{in}\ z)^*}$
$= \lambda x.\lambda u.u.r\ \langle \lambda v.\bot, \lambda y.\bot\rangle\ \langle \underline{x.l}, \lambda z.x.r\ \langle z_l, \lambda k.\bot\rangle\rangle$
$= \overline{\lambda x.u.r\ \langle \lambda v.\bot, \lambda y.\bot\rangle[u := \langle \underline{x.l}, \lambda z.x.r\ \langle \underline{z.l}, \lambda y.\bot\rangle\rangle]}$.

To prove that $(\_)$ is a linear cost reduction, we note that $\underline{a} = a$ for all approximants, and show (by induction on derivation):

Lemma 4.5: For any approximants a, b, c_l, c_r, $\widehat{\propto_{\lambda\sigma}}(a, b[x := \langle c_l, c_r\rangle]) \leq^{\infty} \widehat{\propto_{\mathsf{c}}}(a, b[\langle c_l, c_r\rangle/x])$.

It is then straightforward to show that for all λ_{CPS} terms a, A, $\widehat{\propto_{\lambda\sigma}}(\underline{a}, \underline{A}) \leq^{\infty} \widehat{\propto_{\mathsf{c}}}(a, A)$. Since $|a| \leq \widetilde{\propto_{\mathsf{c}}}(a, A)$ for all approximants a (by a straightforward induction) we have:

Proposition 4.6: $\widetilde{\propto_{\lambda\sigma}} \lesssim \widetilde{\propto_{\mathsf{c}}}$.

V. Nameless Representation

The next cost reduction step simplifies the binding structure of $\lambda\sigma_{\mathsf{CPS}}$ by converting terms to a "nameless" representation, using natural number indices in place of variable names.These are instances of *de Bruijn levels* and *reverse de Bruijn indices* [41]: informally, the de Bruijn level of a subterm is the number of distinct free variables it may contain and the reverse de Bruijn index of a variable is the level of (the minimal subterm containing) its binder.

Our nameless representation makes all global binding information locally available by decorating subterms with their de Bruijn levels. We can then delete the λ-abstractions themselves, and represent variable occurrences as their reverse de Bruijn indices — so all variable occurrences with the same binder have the same index. Moreover, the re-indexing of variables in substituted terms (required when their binder has changed levels) can be implemented as part of the explicit substitution itself, rather than requiring a separate shift operation. (Note that the sum of the de Bruijn index (number of λs between the occurrence and its binder) and reverse de Bruijn index of a variable occurrence is equal to its de Bruijn level, so decorating subterms with their levels allows either indices or reverse indices to be used. The latter is more comprehensible.)

$$j.t\,\langle A_l, A_r\rangle^m[i := \langle B_l, B_r\rangle]^n \to \begin{cases} B_t[i+n+1-m := \langle A_l[i := \langle B_l, B_r\rangle]^{n+1}, A_r[i := \langle B_l, B_r\rangle]^{n+1}\rangle]^n & \text{if } j = i \\ j.t\,\langle A_l[i := \langle B_l, B_r\rangle]^{n+1}, A_r[i := \langle B_l, B_r\rangle]^{n+1}\rangle^n & \text{if } j < i \\ (j+n-m).t\,\langle A_l[i := \langle B_l, B_r\rangle]^{n+1}, A_r[i := \langle B_l, B_r\rangle]^{n+1}\rangle^n & \text{if } j > i \end{cases}$$

$$\underline{i.t}^n \to i.t\,\langle \underline{n.l}^{n+1}, \underline{n.r}^{n+1}\rangle^{n+1}$$

Fig. 4. Reduction rules for nameless explicit substitutions

$$\begin{array}{c} 1.r\,\langle\bot,\bot\rangle^2[1 := \langle \underline{0.l}^2, 0.r\,\langle \underline{1.l}^2, \bot\rangle^2\rangle]^1 \sqsubseteq_\bot 1.r\,\langle\bot,\bot\rangle^2[1 := \langle\bot, 0.r\,\langle\bot,\bot\rangle^2\rangle]^1 \\ \downarrow \\ 0.r\langle\bot,\bot\rangle^2[1 := \langle\bot[1 := \langle\bot, 0.r\langle\bot,\bot\rangle^2\rangle]^1, \bot[1 := \langle\bot, 0.r\langle\bot,\bot\rangle^2\rangle]^1\rangle]^1 \sqsubseteq_\bot 0.r\,\langle\bot,\bot\rangle^2[1 := \langle\bot,\bot\rangle]^1 \\ \downarrow \\ 0.r\,\langle\bot[1 := \langle\bot,\bot\rangle]^2, \bot[1 := \langle\bot,\bot\rangle]^2\rangle^1 \sqsubseteq_\bot 0.r\,\langle\bot,\bot\rangle^1. \end{array}$$

Fig. 5. Derivation of $\lceil\underline{(\lambda x.\lambda y.\bot)^*}\rceil \propto_\sigma \lceil\underline{(\lambda x.\mathsf{let}\ z = (\lambda u.\lambda v.\bot)\ x\ \mathsf{in}\ z)^*}\rceil$ by innermost reduction sequence.

$$y.t\langle\lambda z.M_l, \lambda z.M_r\rangle^{\Gamma,x,\Sigma}[x := \langle\lambda w.N_l, \lambda w.N_r\rangle^{\Gamma,\Delta}] \to \begin{cases} N_t^{\Gamma,\Delta,w}[w := \langle\lambda z.M_l^{\Gamma,x,\Sigma,z}[x := \langle\lambda w.N_l, \lambda w.N_r\rangle^{\Gamma,\Delta}], \lambda z.M_r^{\Gamma,x,\Sigma,z}[x := \langle\lambda w.N_l, \lambda w.N_r\rangle^{\Gamma,\Delta}]\rangle^{\Gamma,\Delta,\Sigma}] & \text{if } x = y \\ y.t\,\langle\lambda z.M_l^{\Gamma,x,\Sigma,z}[x := \langle\lambda w.N_l, \lambda w.N_r\rangle^{\Gamma,\Delta}], \lambda z.M_r^{\Gamma,x,\Sigma,z}[x := \langle\lambda w.N_l, \lambda w.N_r\rangle^{\Gamma,\Delta}]\rangle & \text{otherwise} \end{cases}$$

Fig. 6. Reduction Rules for Context-Decorated Explicit Substitutions

Definition 5.1: Terms of σ_{CPS} — the nameless CPS-calculus with explicit substitutions — are given by the grammar:
$A ::= \bot \mid \underline{i.t}^n \mid i.t\,\langle A, A\rangle^n \mid A[i := \langle A, A\rangle]^n$.
where the $n \in \mathbb{N}$ (superscripts) are de Bruijn levels, the $i \in \mathbb{N}$ are reverse de Bruijn indices and $t \in \{l, r\}$ are left/right tags. The reduction rules for σ_{CPS} terms are given in Figure 4. Note that the rules for reducing an explicit substitution $j.t\,\langle A_l, A_r\rangle^m[i := \langle B_l, B_r\rangle]^n$ with $j \neq i$ are different depending on whether $j < i$ or $j > i$ — i.e. whether j is bound below or above the level where the explicit substitution for i was created: in the latter case, j is shifted by the difference in levels $(m - n)$ to "make room" for variables free in B_l, B_r but not A_l, A_r. As for $\lambda\sigma_{\mathsf{CPS}}$, the evaluation tree semantics is defined by innermost reduction up to the preorder $\sqsubseteq_\bot$.

Definition 5.2: Approximants are given by the grammar:
$a ::= \bot \mid i.t\,\langle a, a\rangle^n$.
$a \propto_\sigma A$ if a is an approximant and $A \Rightarrow^* A'$ such that $a \sqsubseteq_\bot A'$. The intrinsic cost $\widetilde{\propto_\sigma}(a, A)$ is the sum of the size of a with the length of the shortest such innermost reduction sequence.

A $\lambda\sigma_{\mathsf{CPS}}$ term-in-context is compiled to its nameless representation by decorating subterms with their levels, replacing variables with reverse de Bruijn indices and erasing λ-abstractions.

Definition 5.3: Writing $|\Gamma|$ for the length of the Γ, the nameless representation of a $\lambda\sigma_{\mathsf{CPS}}$ term-in-context $\Gamma \vdash A$ is the term $\lceil A\rceil_\Gamma$, where:

$$\lceil\bot\rceil_\Gamma = \bot \quad \lceil\lambda x.M\rceil_\Gamma = \lceil M\rceil_{\Gamma,x} \quad \lceil\underline{x.t}\rceil_{\Gamma,x,\Gamma'} = \underline{|\Gamma|.t}^{|\Gamma,x\Gamma'|}$$

$$\lceil\lambda w.x.t\,\langle V_l, V_r\rangle\rceil_{\Gamma,x,\Gamma'} = |\Gamma|.t\,\langle\lceil V_l\rceil_{\Gamma,x,\Gamma'}, \lceil V_r\rceil_{\Gamma,x,\Gamma'}\rangle^{|\Gamma,x,\Gamma'|}$$

$$\lceil M[y := \langle V_l, V_r\rangle]\rceil_\Gamma = \lceil M\rceil_{\Gamma,y}[|\Gamma, y| := \langle\lceil V_l\rceil_\Gamma, \lceil V_r\rceil_\Gamma\rangle]^{|\Gamma|}$$

Figure 5 implements the example of Figure 2, showing that $\lceil\underline{(\lambda x.\lambda y.\bot)^*}\rceil \propto_\sigma \lceil\underline{(\lambda x.\mathsf{let}\ z = (\lambda u.\lambda v.\bot)\ x\ \mathsf{in}\ z)^*}\rceil$.

$\lceil\_\rceil$ is not a homomorphism between the two reduction systems: $M \to M'$ does not imply $\lceil M\rceil_\Gamma \to \lceil M'\rceil_\Gamma$ when $M \equiv x_j.t\,\langle U_l, U_r\rangle[x_i := \langle V_l, V_r\rangle]$ and $i > j$. The problem is that $\lambda\sigma_{\mathsf{CPS}}$ does not keep track of the contexts of subterms, as σ_{CPS} does via the level decorations. To prove that $\lceil\_\rceil$ is nevertheless a sound cost reduction, we define in Table III a version of $\lambda\sigma_{\mathsf{CPS}}$ in which explicit substitutions are decorated with their contexts. The rule for reducing decorated $\lambda\sigma_{\mathsf{CPS}}$ terms (Figure 6) preserves well-formedness of terms in context (i.e. subject reduction: if $\Gamma \vdash A$ and $A \to A'$ then $\Gamma \vdash A'$).
Extending nameless representation to decorated terms:
$\lceil M^{\Gamma,x,\Gamma'}[x := \langle V_l, V_r\rangle^{\Gamma,\Delta,\Gamma'}]\rceil_{\Gamma,\Delta,\Gamma'}$
$= \lceil M\rceil_{\Gamma,x,\Gamma'}[|\Gamma| := \langle\lceil V_l\rceil_{\Gamma,\Delta}, \lceil V_r\rceil_{\Gamma,\Delta}\rangle]^{|\Gamma,\Delta,\Gamma'|}$.

$$\frac{}{\Gamma\vdash\bot} \qquad \frac{}{\Gamma,x,\Gamma'\vdash x.t} \qquad \frac{\Gamma,x,\Gamma'\vdash V_l \quad \Gamma,x,\Gamma'\vdash V_r}{\Gamma,x,\Gamma'\vdash x.t\,\langle V_l, V_r\rangle}$$

$$\frac{\Gamma,x\vdash M}{\Gamma\vdash\lambda x.M} \qquad \frac{\Gamma,x,\Gamma'\vdash M \quad \Gamma,\Delta\vdash V_l \quad \Gamma,\Delta\vdash V_r}{\Gamma,\Delta,\Gamma'\vdash M^{\Gamma,x,\Gamma'}[x:=\langle V_l, V_r\rangle^{\Gamma,\Delta}]}$$

TABLE III
CONTEXT-DECORATED $\lambda\sigma_{\mathsf{CPS}}$

Let $\|\_\|$ be the operation erasing all decorating contexts. By decorating each explicit substitution with the largest available context (as in $\lceil\_\rceil$), we show:

Lemma 5.4: For any $\lambda\sigma_{\mathsf{CPS}}$ term-in-context $\Gamma \vdash A$ there is a decorated term-in-context $\Gamma \vdash \lfloor A\rfloor_\Gamma$ such that $\|\lfloor A\rfloor_\Gamma\| = A$ and $\lceil\lfloor A\rfloor_\Gamma\rceil_\Gamma = \lceil A\rceil_\Gamma$.
Moreover, the reduction rules may be decorated as follows:

Lemma 5.5: For any well-formed decorated term $\Gamma \vdash A$:

1) $\|A\| \to B$ if and only if there exists a decorated term $\Gamma \vdash A'$ such that $A \to A'$ and $\|A'\| = B$.
2) $\|A\| \sqsupseteq^\alpha_\bot B$ if and only if there exists a decorated term $\Gamma \vdash A'$ such that $A \sqsupseteq^\alpha_\bot A'$ and $\|A'\| = B$.
3) $\lceil A\rceil_\Gamma \to B$ if and only if there exists a decorated term

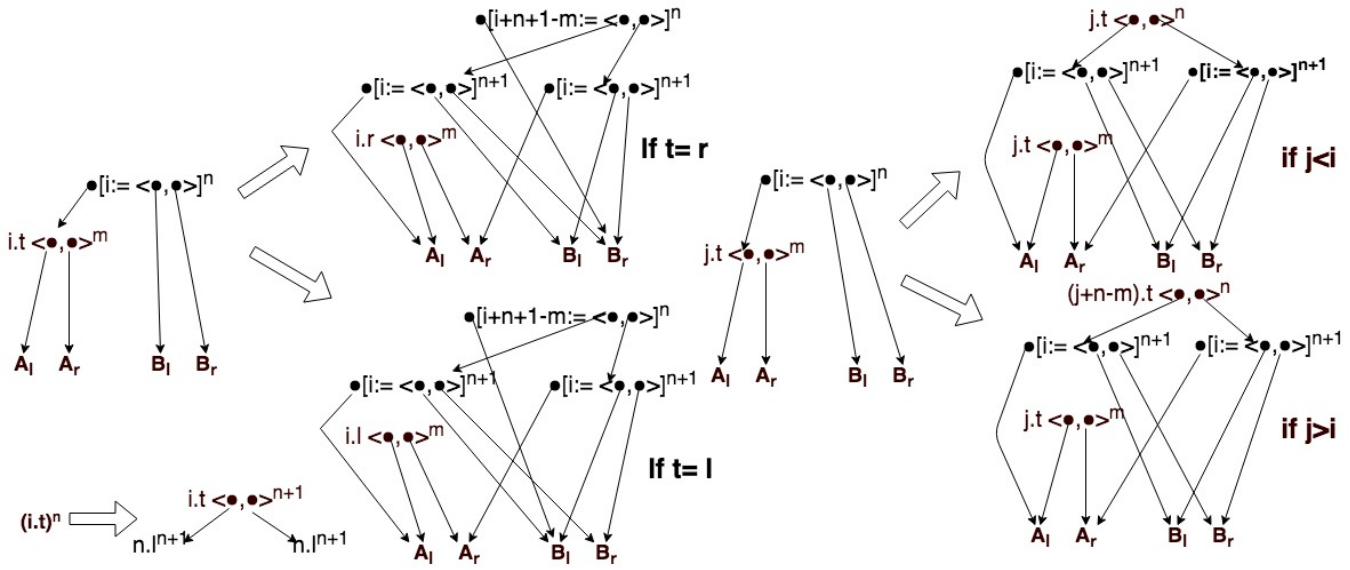


Fig. 7. Reduction Rules for Heap Graphs

A' such that $A \to A'$ and $\lceil A' \rceil_\Gamma = B$.

4) If $A \sqsupseteq_\perp^\alpha A'$ then $\lceil A \rceil_\Gamma \sqsupseteq_\perp \lceil A' \rceil_\Gamma$. If $\lceil A \rceil_\Gamma \sqsupseteq_\perp B$ then $A \Rightarrow^* A' \sqsupseteq_\perp^\alpha A''$ for some decorated A', A'' such that $\lceil A'' \rceil_\Gamma = B$.

(4) is not a logical equivalence because e.g. $\underline{i.t}^n \sqsupseteq_\perp \perp$ but $\underline{x.t} \to \lambda y.\underline{x.t}\langle y_l, y_r\rangle \sqsupseteq_\perp \lambda y.\perp$.

Proposition 5.6: For any $\lambda\sigma_{\text{CPS}}$ terms-in-context $\Gamma \vdash a, A$ $\widehat{\propto_\sigma}(\lceil a \rceil_\Gamma, \lceil A \rceil_\Gamma) = \widehat{\propto_{\lambda\sigma}}(a, A)$.

PROOF: By Lemmas 5.4 and 5.5, if $A \stackrel{n}{\Rightarrow} A' \sqsupseteq_\perp a$ then there is a decorated inner reduction sequence $\lfloor A \rfloor_\Gamma \stackrel{n}{\Rightarrow} A''$ for some A'' such that $\|A''\| = A'$ and hence $a \sqsubseteq_\perp^\alpha A''$. Then $\lceil A \rceil_\Gamma = \lceil \lfloor A \rfloor_\Gamma \rceil_\Gamma \stackrel{n}{\Rightarrow} \lceil A'' \rceil_\Gamma \sqsupseteq_\perp \lceil a \rceil_\Gamma$.
Conversely, if $\lceil A \rceil_\Gamma = \lceil \lfloor A \rfloor_\Gamma \rceil_\Gamma \Rightarrow^* A' \sqsupseteq_\perp \lceil a \rceil_\Gamma$ then $\lfloor A \rfloor_\Gamma \Rightarrow^* B$ for some B such that $B \sqsupseteq_\perp^\propto a$, so $A = \|\lfloor A \rfloor_\Gamma\| \Rightarrow^* \|B\| \sqsupseteq_\perp^\alpha a$ as required. □

Since compilation to nameless form maintains the size of approximants it is therefore a cost-reduction from $\widetilde{\propto_{\lambda\sigma}}$ to $\widetilde{\propto_\sigma}$.

Proposition 5.7: $\widetilde{\propto_\sigma} \lesssim \lceil \widetilde{\propto_{\lambda\sigma}} \rceil$.

VI. Graph Reduction of Nameless Terms

The next cost reduction step transforms nameless terms to a graph representation and implements explicit substitutions by graph rewriting. This allows for deterministic, lazy evaluation by sharing, which is equivalent in cost to clairvoyant eager reduction because innermost reduction sequences can be reordered to call-by-need without duplicating work. We will first describe term graphs and their rewriting rules diagrammatically, and then give a formal characterization for them as equivalence classes of states in a nominal transition system of abstract memory heaps.

Definition 6.1: A heap graph is a σ_{CPS} term graph [42] — a directed, acyclic graph with a specified root node, in which each node is labelled with one of the σ_{CPS} term constructors: $\perp \mid \underline{i.t}^n \mid i.t\ \langle \bullet, \bullet \rangle^n \mid \bullet\ [i \mapsto \langle \bullet, \bullet \rangle]^n$
(We call these $\perp$, variable, substitution and value nodes, respectively.) Each node may have any number of incoming edges, and has one outgoing edge from each $\bullet$ (i.e. the node has a sequence of outgoing edges of the same length as the arity of the label, which is the number of $\bullet$s occurring in it.) The term rewriting rules for σ_{CPS} determine graph rewriting rules, which are given in Figure 7. Every reduction replaces exactly one node (the uppermost) with a graph, corresponding to an update of the original node to the root of the replacement graph and the creation of the rest of its nodes. The original node to be updated and the new nodes created must be variable or substitution nodes (collectively called computation nodes) — i.e. value or $\perp$-nodes are never rewritten or created). So graph reduction of a term corresponds to a process of rewriting computation nodes to values, in the process creating new computation nodes and thus unfolding a graph containing the evaluation tree.

The key property of this graph reduction system, allowing a cost-invariant determinization of the computation of the approximation relation, is *strong confluence* (the following proof may be formalized in the nominal setting below).

Proposition 6.2: If $\mathcal{G} \to \mathcal{G}_1$ and $\mathcal{G} \to \mathcal{G}_2$ then either $\mathcal{G}_1 = \mathcal{G}_2$ or there exists $\mathcal{G}'$ such that $\mathcal{G}_1 \to \mathcal{G}'$ and $\mathcal{G}_2 \to \mathcal{G}'$.

PROOF: In any graph there is at most one possible reduction for each computation node, and this can depend only on a value node. Hence any two available reductions are to different computation nodes and may be applied in any order. □

We formalise the heap graph reduction system by representing graphs syntactically as equivalence classes of states in a nominal (labelled) transition system of memory-states (heaps).

If $H(X) = j.t\langle A_l, A_r\rangle^m$ and $Z_l, Z_r \notin \mathsf{dom}(H)$ then $H\{Y = X[i := \langle B_l, B_r\rangle]^n\} \xrightarrow{Y}$

$$\begin{cases} H\{Y = B_t[i+m+2-n := \langle Z_l, Z_r\rangle]^n; Z_l = A_l[i := \langle B_l, B_r\rangle]^{n+1}; Z_r = A_r[i := \langle B_l, B_r\rangle]^{n+1}\} & \text{if } i = j, \\ H\{Y = j.t\langle Z_l, Z_r\rangle^n; Z_l = A_l[i := \langle B_l, B_r\rangle]^{n+1}; Z_r = A_r[i := \langle B_l, B_r\rangle]^{n+1}\} & \text{if } i < j, \\ H\{Y = (j+m-n).t\langle Z_l, Z_r\rangle^n; Z_l = A_l[i := \langle B_l, B_r\rangle]^{n+1}; Z_r = A_r[i := \langle B_l, B_r\rangle]^{n+1}\} & \text{if } i > j. \end{cases}$$

$H\{Y = \underline{i.t}^n\} \xrightarrow{Y} H\{Y = i.t\,\langle Z_l, Z_r\rangle^{n+1}; Z_l = \underline{n.l}^{n+1}; Z_r = \underline{n.r}^{n+1}\}$

Fig. 8. Transition Rules for Heap Reduction

This leads naturally to an implementation of graph rewriting by updating the heap. To represent creation of nodes (allocation of fresh locations) rigorously while avoiding the specifics of memory management, we work with labelled transitions over *nominal sets* [43], [44].That is, we assume a fixed, countably infinite set $\mathcal{A}$ of *atoms* $(X, Y, Z, \ldots)$ — these can be understood as the names of graph nodes, or locations in the store, as appropriate. Let G be the group of permutations on $\mathcal{A}$: a nominal set $\mathcal{S}$ is an action of G on a set $|\mathcal{S}|$ such that the *support* of each $s \in |\mathcal{S}|$, $\sup(s) = \bigcap\{\mathcal{A}' \subseteq \mathcal{A} \mid (\forall a \in \mathcal{A}'.\pi(a) = a) \implies \pi(s) = s\}$ is finite. We write $s \sim s'$ for permutation equivalence (i.e. $s \sim s$ if there exists π such that $s' = \pi(s)$

Definition 6.3: Let $\mathcal{C}$ be the nominal set of location contents or node labels — that is:
$\{\bot\} \cup \{\underline{i.t}^n \mid i, n \in \mathbb{N}, t \in \{l, r\}\}$
$\cup \{i.t\,\langle Y_l, Y_r\rangle^n \mid i, n \in \mathbb{N}, t \in \{l, r\}, Y_l, Y_r \in \mathcal{A}\}$
$\cup \{Y[i := \langle Z_l, Z_r\rangle]^n \mid i, n \in \mathbb{N}, Y, Z_l, Z_r \in \mathcal{A}\}$
— with the pointwise action of G. A *heap* is an element of the nominal set of finite partial functions $H : \mathcal{A} \rightharpoonup \mathcal{C}$ (with G-action $\pi(H)(X) = \pi(H(\pi^{-1}(X)))$) which satisfies:

- No dangling pointers: $\sup(H) = \mathsf{dom}(H)$.
- Acyclicity: The transitive closure of the relation $\ll_H$ such that $X \ll_H Y$ iff $Y \in \sup(H(X))$ is irreflexive.

The nominal set of *rooted heaps* consists of pairs (H, X) of a heap and a location $X \in \mathsf{dom}(H)$ (which may have incoming edges) with the pointwise G-action. We may thus define a heap-graph to be a permutation-equivalence class $[H, X]$ of rooted heaps. The σ_{CPS} graph rewriting rules are implemented as a transition relation between heaps in which each reduction is labelled with the name of the node updated.

Definition 6.4: The *states* and *actions* of the nominal LTS of heaps are the nominal sets of heaps and location names, and the *transition relation* is given by the rules in Figure 8. This is an equivariant relation — $H \xrightarrow{Y} H'$ implies $\pi(H) \xrightarrow{\pi(Y)} \pi(H')$. We ignore the labels where they are not relevant, and so define heap-graph reduction by $[H, X] \to [H', X]$ if $H \to H'$.

The approximation relation $\propto_{\mathcal{H}}$ on heap graphs is defined via the equivariant operation $[\![\_]\!]$ which converts a rooted heap (and thus the corresponding graph) to a term — i.e. $[\![H, X]\!] =$

$$\begin{array}{ll} i.t\,\langle [\![H, Y_l]\!], [\![H, Y_r]\!]\rangle^n & \text{if } H(X) = i.t\,\langle Y_l, Y_r\rangle^n \\ [\![H, Y]\!][i := \langle [\![H, Z_l]\!], [\![H, Z_r]\!]\rangle]^n & \text{if } H(X) = Y[i := \langle Z_l, Z_r\rangle]^n \\ H(X) & \text{otherwise.} \end{array}$$

Definition 6.5 (Heap Approximation): $[h, X] \propto_{\mathcal{H}} [H, Y]$ if $H \longrightarrow\!\!\!\!\!\rightarrow H'$ such that $[\![h, X]\!] \sqsubseteq_\bot [\![H', Y]\!]$.

The intrinsic cost model $\widetilde{\propto_{\mathcal{H}}}([h, X], [H, Y])$ is the length of the shortest such reduction sequence, plus the size of $[\![h, X]\!]$.

A. Cost Reduction from nameless terms to heaps

A term of σ_{CPS} is compiled to a rooted heap by labelling each node of its syntax tree with a (distinct) location name. The set of all such labellings defines an interpretation of terms as heap-graphs which sends each term to its own syntax tree.

Definition 6.6: For each nameless term A let $(\!|A|\!)$ be the equivalence class of rooted heaps defined as follows:

- $(\!|\bot|\!) \triangleq [\{X = \bot\}, X]$ and $(\!|\underline{i.t}^n|\!) \triangleq [\{X = \underline{i.t}^n\}, X]$
- $(\!|i.t\,\langle A_l, A_r\rangle^n|\!) \triangleq [\{X = i.t\,\langle Y_l, Y_r\rangle^n; H_l; H_r\}, X]$ where $(H_l, Y_l) \in (\!|A_l|\!), (H_r, Y_r) \in (\!|A_r, Y_r|\!)$,
- $(\!|A[i := \langle B_l, B_r\rangle]^n|\!) \triangleq$ $[\{X = Y[i := \langle Z_l, Z_r\rangle]^n; H_Y; H_l; H_r\}, X]$ where $(H_Y, Y) \in (\!|A|\!), (H_l, Z_l) \in (\!|B_l|\!), (H_r, Z_r) \in (\!|B_r|\!)_r$.

It is straightforward to show that $[\![(\!|A|\!)]\!] = A$. Given a scheme for assigning distinct locations to each node, of which we omit details, there is a (linear time) compilation sending each term A to some rooted heap $(H, X) \in (\!|A|\!)$. To show that this is a cost reduction, we use the following property of inner reduction up to $\sqsubseteq_\bot$ in σ_{CPS}.

$$\text{If } A \Rightarrow A' \text{ then } A \to A'' \text{ such that } A' \sqsubseteq_\bot A'' \quad (\ddagger)$$

together with the fact that if $[\![H, X]\!] \to A$ by innermost reduction then $H \to H'$ such that $[\![H', X]\!] = A$.

Lemma 6.7: If $\widetilde{\propto_\sigma}(a, [\![H, X]\!]) = n$ then $\widetilde{\propto_{\mathcal{H}}}((\!|a|\!), [H, X]) \leq n$.

PROOF: By induction on n. If $\widetilde{\propto_\sigma}(a, [\![H, X]\!]) = 0$ then $[\![(\!|a|\!)]\!] = a \sqsubseteq_\bot [\![H, X]\!]$ as required. For the induction case, if $\widetilde{\propto_\sigma}(a, [\![H, X]\!]) = n + 1$ then there exists A such that $[\![H, X]\!] \Rightarrow A$ and $\widetilde{\propto_\sigma}(a, A) = n$. By $(\ddagger)$, $[\![H, X]\!] \to A'$ for some A' such that $A \sqsubseteq_\bot A'$ and thus $\widetilde{\propto_\sigma}(a, A') \leq n$ and $H \to H'$ such that $[\![H', X]\!] = A'$.
By induction hypothesis, $\widetilde{\propto_{\mathcal{H}}}((\!|a|\!), [H', X]) \leq n$, and so $\widetilde{\propto_{\mathcal{H}}}((\!|a|\!), [H, X]) \leq n + 1$ as required. □

To prove that $(\!|a|\!) \propto_{\mathcal{H}} (\!|A|\!)$ implies $a \propto_\sigma A$ we need to show that *any* heap reduction sequence on a term graph corresponds to an *innermost* reduction sequence on the term. Observe that if $X \not\ll_H^* Y$ then reduction of Y cannot depend on evaluation of X and so can be performed first — i.e. if $H_1 \xrightarrow{X} H_2 \xrightarrow{Y} H_3$ then $H_1 \xrightarrow{Y} H_2' \xrightarrow{X} H_3'$, where $H_3' \sim H_3'$. Using this principle we can reorder any reduction sequence to a "bottom up" sequence in which no node is reduced before any of its descendants in the graph.

Lemma 6.8: If $H \overset{n}{\twoheadrightarrow} H'$ then there is a sequence $H = H_1 \overset{X_1}{\to} \ldots \overset{X_n}{\to} H_n \sim H'$ such that $X_i \ll^*_{H_i} X_j$ implies $i \leq j$.

Lemma 6.9: If $(\!|a|\!) \propto_{\mathcal{H}} (\!|A|\!)$ then $a \propto_\sigma A$.

PROOF: Supposing that $(\!|a|\!) \propto_{\mathcal{H}} (\!|A|\!)$, there exists $(H_1, X) \in (\!|A|\!)$ and a reduction sequence $H_1 \to \ldots \to H_n$ such that $(\!|a|\!) \sqsubseteq_\perp [\![H_n, X]\!]$ — by Lemma 6.8 we may assume that this is a bottom up sequence. For each $1 \leq i < n$, we define a term A_i such that $A_1 = A$, either $A_i = A_{i+1}$ or $A_i \Rightarrow A_{i+1}$ and $a \sqsubseteq_\perp A_n$, so that $a \propto_\sigma A$ as required.

For each heap H_i define H'_i by $H'_i(X) = \perp$ if $H_i(X)$ is a computation node and $H_n(X)$ is not a value, and $H'_i(X) = H_i(X)$ otherwise. Then either $[\![H'_i, X]\!] = [\![H'_{i+1}, X]\!] = A_{i+1}$ (if the node reduced in H_i is not below X) or $[\![H'_i, X]\!] \to A_{i+1}$ for some A_i such that $[\![H_{i+1}, X]\!] \sqsubseteq_\perp A_{i+1}$ and this is an innermost reduction: if it updates node Y then any computation node $Z \ll_{H_i} Y$ cannot be updated in any later reduction, so $H_n(Z) = H_i(Z)$ and hence $H'_i(Z) = \perp$. □

Thus we have shown:

Proposition 6.10: $\widetilde{\propto_{\mathcal{H}}} \lesssim (\!|\widetilde{\propto_\sigma}|\!)$.

VII. Recognizing Heap Approximation

In this section we define an abstract machine which implements the cost model $\widetilde{\propto_{\mathcal{H}}}$ up to a constant factor overhead. Specifically, given rooted heaps (h, X),(H, Y) the machine accepts in no more than $3.\widetilde{\propto_{\mathcal{H}}}([h, X], [H, Y])$ steps if $[h, X] \propto_{\mathcal{H}} [H, Y]$ and rejects or fails to terminate otherwise.

First, we describe the call-by-need reduction strategy on heaps that the machine will use to evaluate a given node, and show that this strategy always finds a minimal length reduction sequence which achieves this.

Definition 7.1: For a heap H, let $\prec_H$ be the relation on $\mathsf{dom}(H)$: $X \prec_H Y$ if $H(X) = Y[i := \langle Z_l, Z_r \rangle]^m$ for some i, Z_l, Z_r, m. By acyclicity, its transitive closure $\prec^+_H$ is a strict order. A heap reduction $H \overset{Y}{\to} H'$ is *needed* by a computation node $X \in \mathsf{dom}(H)$ if $X = Y$ or $X \prec^+_H Y$.

For a given computation node $X \in \mathsf{dom}(H)$ there is at most one heap reduction for H which is needed by X. This yields an unambiguous reduction strategy for evaluating X: perform the heap reductions needed by X until either X is a value node (success), X is a substitution node but has no needed reduction (failure), or there is an infinite sequence of needed reductions (non-termination).

Definition 7.2: An *evaluation sequence* for $\mathcal{S} \subseteq \mathsf{dom}(H)$ is a reduction sequence $H \longrightarrow\!\!\!\!\!\rightarrow H'$ such that every $X \in \mathcal{S}$ is a value node in H'. It is a *call-by-need sequence* for $\mathcal{S}$ if every transition in the sequence is needed by some $X \in \mathcal{S}$.

Clearly, a call-by-need sequence for a single node X is unique (up to $\sim$), if it exists. Using the following lemma (which allows any available reduction in a sequence to be brought forward), any sequence evaluating X can be reordered to start with a call-by-need sequence for X.

Lemma 7.3: Suppose $H_1 \to H_2 \to \ldots \to H_n$ and $H_1 \overset{X}{\to} H'_2$ where $H_1(X) \neq H_n(X)$. Then there is a reduction sequence $H'_2 \to H'_3 \to \ldots \to H'_n$ such that $H'_n \sim H_n$.

PROOF: By induction on n. We either have $H_1 \overset{X}{\to} H_2$ — in which case we take $H'_i = H_i$ for each $i \leq n$ — or else $H_1 \overset{X}{\to} H_2$ for some $Y \neq X$. Then by strong confluence, $H_2 \overset{X}{\to} H'_3$ and $H'_2 \overset{Y}{\to} H''_3$ such that $H'_3 \sim H''_3$. By induction hypothesis there is a reduction sequence $H'_3 \to H'_4 \to \ldots \to H'_n$ such that $H_n \sim H'_n$, and thus a reduction sequence $H''_3 \to H''_4 \to \ldots \to H''_n$ such that $H''_n \sim H_n$, satisfying the induction hypothesis. □

Proposition 7.4: For any evaluation sequence $H \overset{n}{\longrightarrow\!\!\!\!\!\rightarrow} H'$ for X there is a call-by-need sequence $H \overset{k}{\longrightarrow\!\!\!\!\!\rightarrow} H''$ for X, for some H'', k such that $H'' \overset{n-k}{\longrightarrow\!\!\!\!\!\rightarrow} H'''$ where $H' \sim H'''$.

Next, we define an abstract machine implementing the key subroutine of our semi-decision procedure for heap approximation — given a rooted heap (H, X), this executes the call-by-need sequence of H for X if it exists, by storing the locations upon which it depends on a stack, which is initialized with X. The instruction cycle of the machine checks the stack — if it is empty, the machine halts (accepts), otherwise it reads the contents of the location at the top of the stack:

- If this node has a reduction in the heap, it is performed.
- If it is a value, the location is popped from the stack.
- If it is a substitution node $Y[i := \langle Z_l, Z_r \rangle]^m$ where Y is, a computation node then Y is pushed onto the stack.

If one of these applies the cycle is repeated, otherwise the machine rejects.

Definition 7.5: The root evaluation machine is the nominal automaton in which *states* are pairs $(H; S)$ of a heap and a stack (finite, non-repeating sequence) of locations in $\mathsf{dom}(H)$ (with the pointwise G-action). The *final states* are those of the form $(H; \varepsilon)$, and its *transition relation* is:

$$(H; S, X) \to \begin{cases} (H; S) & \text{if } H(X) \text{ is a value} \\ (H'; S, X) & \text{if } H \overset{X}{\to} H' \\ (H; S, X, Y) & \text{if } X \prec_H Y \text{ and } H \overset{X}{\not\to} H' \end{cases}$$

This relation is nominally deterministic: $(H; S) \to (H'; S')$ and $(H; S) \to (H''; S'')$ implies $(H'; S') \sim (H''; S'')$.

Lemma 7.6: Let S be a stack of computation nodes in H:

1) If there is a call-by-need sequence $H \overset{k}{\longrightarrow\!\!\!\!\!\rightarrow} H'$ for S then $(H; S) \overset{\leq 3k - |S|}{\longrightarrow\!\!\!\!\!\rightarrow} (H'; \varepsilon)$.
2) If $(H; S) \longrightarrow\!\!\!\!\!\rightarrow (H'; \varepsilon)$ then there is a call-by-need sequence $H \longrightarrow\!\!\!\!\!\rightarrow H'$ for S.

PROOF: We show e.g. (1) by induction on $3k - |S|$ (which must be non-negative). If this is zero then $S = \varepsilon$, so we are done. For the induction case, $S \neq \varepsilon$ so let $S = S', X$.

Suppose $H \overset{X}{\to} H''$, so that $(H; S) \to (H''; S)$. Either $H''(X)$ is a computation node, in which case there is a call-by-need sequence for S from H'' to H' of length $k - 1$, and so by induction hypothesis $(H''; S) \overset{\leq 3k-|S|-3}{\longrightarrow\!\!\!\!\!\rightarrow} (H'; \varepsilon)$ and $(H; S) \overset{\leq 3k-|S|}{\longrightarrow\!\!\!\!\!\rightarrow} (H'; S)$ as required, or else $H''(X)$ is a value node, in which case $(H''; S) \to (H''; S')$ and by induction hypothesis $(H''; S') \overset{\leq 3k-|S|-2}{\longrightarrow\!\!\!\!\!\rightarrow} (H'; \varepsilon)$ and so $(H''; S) \overset{\leq 3k-|S|}{\longrightarrow\!\!\!\!\!\rightarrow} (H'; S)$ as required.

$$
\begin{array}{lcl}
(h;H;\varepsilon;S,(X,X')) & \to & \begin{cases} (h;H;\varepsilon;S) & \text{if } h(X)=\bot \\ (h;H;\varepsilon;S,(Y_r,Y_r'),(Y_l,Y_l')) & \text{if } h(X)=j.t\langle Y_l,Y_r\rangle^n \text{ and } H(X)=j.t\langle Y_l',Y_r'\rangle^n \\ (h;H;X';\mathcal{S},(X,X')) & \text{if } h(X)=j.t\langle Y_l,Y_r\rangle^n \text{ and } H(X) \text{ is a computation node} \end{cases} \\
(h;H;S_1,X;S_2) & \to & (h;H';S_1';S_2) \quad \text{if } (H;S_1,X)\to(H';S_1')
\end{array}
$$

Fig. 9. Transitions of the Heap Approximation Machine

Otherwise, $H(X) = Y[i := \langle Z_l, Z_r\rangle]^n$ for some Y, and the call-by-need sequence for S, X is also a call-by-need sequence for S, X, Y. Then $(H;S,X) \to (H;S,X,Y)$, and by induction hypothesis $(H;S,X,Y) \overset{\leq 3k-|S|-1}{\longrightarrow\!\!\!\!\!\rightarrow} (H';\varepsilon)$ and so $(H;S,X) \overset{\leq 3k-|S|}{\longrightarrow\!\!\!\!\!\rightarrow} (H;\varepsilon)$ as required. □

A. The Heap Approximation Machine

We now define an abstract machine which recognizes the relation $\propto_{\mathcal{H}}$ between rooted heaps in fewer than $3.\widetilde{\propto_{\mathcal{H}}}$ steps by traversing h (in pre-order), evaluating the corresponding node of H with the root evaluation machine and comparing them. The states of the machine are tuples $(h;H;S_1;S_2)$ consisting of heaps h, H, a stack S_1 for the root-evaluation machine, and a stack S_2 of locations $X_1, X_1', \ldots, X_n, X_n'$ for which the machine must check that $[h, X_i] \propto_{\mathcal{H}} [H, X_i']$ for each $i \leq n$. Its transition relation is given in Figure 9.

To determine whether $[h, X] \propto_{\mathcal{H}} [H, X']$, the machine is initialized with the state $(h;H;\varepsilon;X,X')$. It executes the following instruction cycle:

1) Halt (accept) if both stacks are empty — i.e. the final states are those of the form $(h;H;\varepsilon;\varepsilon)$.
2) Otherwise, pop two addresses X, X' from S_2.
3) If $h(X) = \bot$ then return to 1.
4) If $h(X)$ is a value node $j.t\langle Y_l, Y_r\rangle^n$ then evaluate $H(X')$ using the root evaluation machine (if necessary). Then if $H(X') = j.t\langle Y_l', Y_r'\rangle^n$, push Y_r, Y_r', Y_l, Y_l' onto the stack and return to 1. Otherwise, reject.

Lemma 7.7: Suppose $H \overset{k}{\longrightarrow\!\!\!\!\!\rightarrow} H'$ such that $[\![h, X_i]\!] \sqsubseteq_\bot [\![H', X_i']\!]$ for $1 \leq i \leq n$. Then $(h;H;\varepsilon;X_1,X_1',\ldots,X_n,X_n')$ is accepted within $\sum_{i\leq n} |[\![h, X_i]\!]| + 3k$ steps.

PROOF: By induction on $\sum_{i\leq n} |[\![h, X_i]\!]|$. At the base case, the stack is empty and the machine accepts. For the induction step, let $S = X_1, X_1', \ldots, X_{n-1}, X_{n-1}'$. If $h(X_n) = \bot$ then $(h;H;\varepsilon;S,X_n,X_n') \to (h;H;\varepsilon;S)$. By induction hypothesis $(h;H;\varepsilon;S)$ is accepted within $\sum_{i\leq n} |[\![h, X_i]\!]| + 3k - 1$ steps.

Otherwise, $h(X_n) = j.t\langle Y_l, Y_r\rangle^n$ for some Y_l, Y_r such that $[\![h, X_n]\!] = j.t\langle [\![h, Y_l]\!], [\![h, Y_r]\!]\rangle \sqsubseteq_\bot [\![H', X_n']\!]$ and so $H'(X_n') = j.t\langle Y_l', Y_r'\rangle^n$ for some Y_l', Y_r' such that $[\![h, Y_l]\!] \sqsubseteq_\bot [\![H', Y_l']\!]$ and $[\![h, Y_r]\!] \sqsubseteq_\bot [\![H', Y_r']\!]$. By Proposition 7.4 there is a call-by-need sequence $H \overset{l}{\longrightarrow\!\!\!\!\!\rightarrow} H''$ for X_n' such that $H'' \overset{k-l}{\longrightarrow\!\!\!\!\!\rightarrow} H'''$ with $H''' \sim H'$. By Lemma 7.6, $(h;H;\varepsilon;S,X_n,X_n') \overset{3l}{\longrightarrow\!\!\!\!\!\rightarrow} (h;H'';\varepsilon;S,X_n,X_n') \to (h;H'';\varepsilon;S,Y_r,Y_r',Y_l,Y_l')$. By induction hypothesis this is accepted within $\sum_{i\leq n-1} |[\![h, X_i]\!]| + |[\![h, Y_l)]\!]| + |[\![h, Y_r)]\!]| + 3(k-l)$ steps. Hence $(h;H;\varepsilon;S,X_n,X_n')$ is accepted within $\sum_{i\leq n} |[\![h, X_i]\!]| + 3k$ steps as required. □

Lemma 7.8: If $(h;H;\varepsilon;X_1,X_1',\ldots,X_n,X_n')$ is accepted then $H \longrightarrow\!\!\!\!\!\rightarrow H'$ such that $[\![h, X_i]\!] \sqsubseteq_\bot [\![H', X_i']\!]$ for $1 \leq i \leq n$. (The proof is similar to Lemma 7.7.) So we have shown that:

Proposition 7.9: The heap approximation machine implements the cost model $3.\widetilde{\propto_{\mathcal{H}}}$.

It remains to observe that heap approximation may be implemented on a Random Access Machine with a constant factor overhead. In other words, there is a RAM program, a constant K and a (linear time) function r from the states of the heap approximation machine to RAM states (finite partial functions from $\mathbb{N}$ to $\mathbb{N}$) such that if $(h;H;S_1,S_2)$ is accepted within n steps, then $r(h;H;S_1,S_2)$ is accepted within $K.n$ steps by the RAM, and if $(h;H;S_1,S_2)$ is not accepted, then $r(h;H;S_1,S_2)$ is not accepted. Without being specific about the instruction-set of the RAM, we note that it is sufficient to partition the address space of the RAM into:

- Two disjoint heaps, in which each address stores the contents of a location as a 5-tuple of integer values (e.g. $X[i := \langle Y_l, Y_r\rangle]^m$ as (X, i, Y_l, Y_r, m)) tagged according to which kind of node they represent — with an operation allocating a new address (using a register storing a heap pointer) requiring a constant number of RAM operations.
- Two stacks, storing integer addresses.
- a finite number of fixed registers with operations to read from, and write results to, any field of any register in the heaps, or the top of the stacks (pop and push) within a bounded number of RAM steps.

and give constant time encodings of copy, increment, addition and subtraction and conditional jump operations, reading from and writing to the registers. (The full expressiveness of an arithmetic RAM [4] is not required as the operands are linearly bounded in the initial state and reduction length.) Each step of heap-reduction, root evaluation, and heap approximation thus requires a bounded number of RAM steps, and so:

Theorem 7.10: $\widetilde{\propto_v}$ may be implemented on a Random Access Machine, up to constant factor overhead.

VIII. CONCLUSIONS

Using semantic insights, we have defined and implemented a cost model for computing call-by-value evaluation trees. This comes with a compositionality principle for establishing upper cost bounds: an approach to reasoning about lower bounds is suggested by Remark 4.3 but requires a broader framework of results. For example, we may establish that our cost model is *invariant* by giving a Turing machine implementation in λ_v

which respects it (see e.g. [11]). This makes significant use of the polynomial overhead allowed by the invariance thesis.

The potential benefits of our cost model are of two kinds: it allows compositional reasoning about costs, based on the simple principle of assuming a clairvoyant, eager evaluator, and those costs satisfy a theoretical efficiency property. To reap these benefits with a practical implementation will require consideration of other factors.

- Space efficiency will require garbage collection. Sharing is itself a time/memory tradeoff with a risk of space leaks?
- Optimization — despite its theoretical efficiency, our algorithm is capable of optimizations such as implementing let using pointers, recursion using cyclic heaps, or control structures using linear substitutions.
- Parallelization — multiple processors may compute separate nodes of one or more trees while sharing access to the same heap.
- Types — adding typing information allows evaluation trees to take more varied forms, with a corresponding increase in complexity of the graph reduction rules.
- Architecture — the assumption that accessing all parts of the store has constant-bounded cost is architecture-dependent. How far the Von Neumann bottleneck affects complete laziness remains to be seen.

These considerations in turn suggest further theoretical problems such as formulating and establishing compositional cost models for for space resources or parallel computation.

References

[1] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, pp. 98–107, 1989.

[2] D. Ghica, "Slot games: A quantitative model of computation," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005, pp. 85 – 97.

[3] J. Laird, G. Manzonetto, G. McCusker, and M. Pagani, "Weighted relational models of typed lambda-calculi," in *Proceedings of LICS '13*, 2013.

[4] P. van Emde Boas, "Machine models and simulations," in *Handbook of Theoretical Computer Science (vol. A)*. MIT Press, 1991, pp. 1–69.

[5] H. P. Barendregt, *The Lambda Calculus, its Syntax and Semantics, revised edition*. North-Holland, 1984.

[6] J. Lamping, "An algorithm for optimal lambda calculus reduction," in *Proceedings of POPL '90*, 1990, pp. 16–30.

[7] G. Frandsen and C. Sturtivant, "What is an efficient implementation of the λ-calculus," in *Proceedings of FPCA '91*, 1991, pp. 289–312.

[8] J. Lawall and H. Mairson, "Optimality and inefficiency: what isn't a cost model of the lambda-calculus," *ACM Sigplan Notices*, vol. 31, 1996.

[9] A. Asperti and H. Mairson, "Parallel beta reduction is not elementary recursive," p. 49–80, 2001.

[10] U. D. Lago and S. Martini, "The weak lambda calculus as a reasonable machine," *Theor. Comput. Sci.*, vol. 398(1-3), pp. 32 – 50, 2008.

[11] ——, "An invariant cost model for the lambda calculus," in *Proceedings of CiE 2006*, ser. LNCS, no. 3988. Springer, 2006, pp. 105–114.

[12] B. Accattoli and U. D. Lago, "On the invariance of the unitary cost model for head reduction," in *Proceedings of RTA '12*. Springer, 2012, pp. 22–37.

[13] ——, "Beta reduction is invariant, indeed," in *Proceedings of CSL-LICS '14*, 2014.

[14] T. Balabonski, "Weak optimality, and the meaning of sharing," in *Proceedings of ICFP '13*, 2013.

[15] J. Hackett and G. Hutton, "Call-by-need is clairvoyant call-by-value," in *Proceedings of ICFP '19*, 2019.

[16] C. Wadsworth, "Semantics and pragmatics of the lambda-calculus," Ph.D. dissertation, Oxford University, 1971.

[17] C. Holst and D. Gomard, "Partial evaluation is fuller laziness," in *Proceedings of PEPM' '91*, 1991, pp. 223 – 233.

[18] F.-R. Sinot, "Complete laziness: A natural semantics," in *Proceedings of WRS '07*, ser. ENTCS, no. 204, 2008.

[19] M. J. Thyer, "Lazy specialization," Ph.D. dissertation, University of York, 1999.

[20] V. Danos, H. Herbelin and L. Regnier, "Games semantics and abstract machines," in *Proceedings of the Eleventh International Symposium on Logic In Computer Science, LICS '96*, 1996.

[21] P. Clairambault, "Estimation of the length of interactions in arena game semantics," in *Proceedings of FoSSaCS*, ser. LNCS, no. 6604. Springer, 2011, pp. 335–349.

[22] B. Accattoli, U. D. Lago, and G. Vanoni, "The (in)efficiency of interaction," in *Proceedings of POPL '21*, 2021.

[23] J. M. E. Hyland and C.-H. L. Ong, "On full abstraction for PCF: I, II and III," *Information and Computation*, vol. 163, pp. 285–408, 2000.

[24] D. Turner, "A new implementation technique for functional languages," *Software practice and experience*, vol. 9, pp. 31–49, 1979.

[25] U. Berger and H. Schwichtenberg, "An inverse of the evaluation functional for typed λ-calculus," in *Proceedings of LICS '91*, 1991.

[26] K. Aehlig and F. Joachimski, "Operational aspects of untyped normalisation by evaluation," *Mathematical Sructures in Computer Science*, vol. 14, pp. 57–611, 2004.

[27] A. Filinski and H. Rohde, "Denotational aspects of untyped normalisation by evaluation," *RAIRO - Theoretical Informatics and Applications*, vol. 29, pp. 423–453, 2005.

[28] P. Clairambault and P. Dybjer, "Game semantics and normalization by evaluation," in *Proceedings of FoSSaCS '15*, ser. LNS, no. 9034, 2015, pp. 56–70.

[29] P.-L. Curien, "Abstract Böhm trees," *Mathematical Structures in Computer Science*, vol. 8, pp. 559–591, 1998.

[30] R. David, "Computing with Böhm trees," *Fundamenta Informaticae*, vol. 45, no. 1, 2001.

[31] P.Aczel, "An introduction to inductive definitions," in *Handbook of Mathematical Logic*, J. Barwise, Ed. Amsterdam: North-Holland, 1977, p. 739–782.

[32] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen, "The essence of compiling with continuations," *ACM SiGPLAN Notices*, vol. 28, 1993.

[33] E. Kerinec, G. Manzonetto, and M. Pagani, "Revisiting call-by-value Böhm trees ˜ in light of their Taylor expansion," *Logical Methods in Computer Science*, vol. 16, pp. 1–26, 2020.

[34] G. McCusker, "Games and full abstraction for a functional metalanguage with recursive types," Ph.D. dissertation, Imperial College London, 1996, cambridge University Press.

[35] R. Amadio and P.-L. Curien, *Domains and Lambda-Calculi*. Cambridge University Press, 1998.

[36] R. S. Bird, "A formal development of an efficient supercombinator compiler," *Science of Computer Programming*, vol. 8, pp. 113–137, 1987.

[37] G. Plotkin, "Call-by-name, call-by-value and the λ-calculus," *Theoretical Computer Science*, vol. 1, pp. 125 – 159, 1975.

[38] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lèvy, "Explicit substitutions," in *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*. ACM, 1990, pp. 31–46.

[39] C. Wadsworth, "The relationship between computational and denotational properties for Scott's D_∞-models of the λ-calculus," *SIAM Journal on Computing*, vol. 5, no. 3, pp. 488–521, 1976.

[40] M. Hyland, "A syntactic characterization of the equality in some models for the untyped lambda-calculus," *Journal of the LMS*, vol. 12, pp. 361–370, 1976.

[41] N. de Bruijn, "Lambda-calculus notation with nameless dummies," *Indagationes Mathematicae*, vol. 34, pp. 381–392, 1972.

[42] H. Barendregt, M. van Eekelen, J. Glauert, R. Kennaway, R. Plasmeijer, and R. Sleep, "Term graph rewriting," in *Proc. Parallel Architectures and Languages Europe*, ser. LNCS, no. 259. Springer-Verlag, 1987, p. 141–158.

[43] A. M. Pitts, *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

[44] J. Parrow, J. Borgström, L.-H. Eriksson, R. Gutkovas, and T. Weber, "Modal Logics for Nominal Transition Systems," in *26th International Conference on Concurrency Theory (CONCUR 2015)*, vol. 42, 2015, pp. 198–211.