A Fresh Inductive Approach to Useful Call-by-Value

Pablo Barenbaum[1,2][0009−0003−2494−3345], Delia Kesner*[3][0000−0003−4254−3129], and Mariana Milicich[3][0009−0007−5722−2607]

[1] Universidad Nacional de Quilmes (CONICET), Argentina
[2] Instituto de Ciencias de la Computación, Universidad de Buenos Aires, Argentina
pbarenbaum@dc.uba.ar
[3] Université Paris Cité, CNRS, IRIF, France
{kesner,milicich}@irif.fr

Abstract. *Useful evaluation* is an optimised evaluation mechanism for functional programming languages, introduced by Accattoli and Dal Lago. The key to useful evaluation is to represent programs with *sharing* and to implement substitution of terms only when this contributes to the progress of the computation. Initially defined in the framework of call-by-name, useful evaluation has since been extended to call-by-value. The definitions of usefulness in the literature are complex and lack inductive structure, which makes it challenging to (formally) reason about them.
In this work, we define useful call-by-value evaluation inductively, proceeding in two stages. First, we refine the well-known *Value Substitution Calculus*, so the substitution operation becomes *linear*, yielding the LCBV calculus. The two calculi are observationally equivalent. We then further refine LCBV by restricting linear substitution only when it contributes to the progress of the computation, yielding the UCBV strategy. This new substitution notion is sensitive to the surrounding evaluation context, so it is non-trivial to capture it inductively. Moreover, we formally show that the resulting UCBV is a sound and complete implementation of LCBV, optimised to implement useful evaluation.
As a further contribution, we show that the UCBV strategy can be implemented by an existing lower-level abstract machine called GLAMoUr with polynomial overhead in time. This entails, as a corollary, that UCBV is *time-invariant*, *i.e.*, that the number of reduction steps to normal form in UCBV can be used as a measure of time complexity.
Our UCBV strategy is part of the preliminary work required to develop semantic interpretations of useful evaluation, for which its inductive formulation is more suitable than the (non-inductive) existing ones.

Keywords: evaluation strategies · call-by-value · useful evaluation

1 Introduction

The λ-calculus is the foundational model behind functional programming languages and most proof assistants. Implementing them in an *efficient* way requires

* Corresponding author

designing an evaluation mechanism that closely adheres to the operational semantics, while optimising the key operations for resource efficiency, both for time and space. This creates a significant gap between the original model and its optimised version, making it crucial to guarantee that both notions have the same *observable behaviour*.

In the λ-calculus, evaluation strategies vary, with *call-by-name* (CBN) and *call-by-value* (CBV) being two prominent examples. In both cases, efficient implementations rely on term representations with *sharing*. For example, implementations typically use environments to bind a variable to a shared subexpression, rather than textually substituting the variable by many copies of the subexpression. Formally, λ-terms with sharing can be represented with *explicit substitutions*[4] (ESs). An ES binding a variable x to an expression s is written $[x\backslash s]$, and a term t affected by such an ES is written $t[x\backslash s]$ (which some authors note "$\mathsf{let}\ x = s\ \mathsf{in}\ t$"). The expression $t[x\backslash s]$ means that all the free occurrences of x in t are bound to s, and thus s is *shared*. While the β-reduction rule of the λ-calculus is based on the meta-level substitution operation $t\{x := s\}$ that replaces all the free occurrences of x in t at once, some calculi with ESs allow replacing a *single* occurrence of a variable, such as $(x\,x)[x\backslash t] \to (t\,x)[x\backslash t]$. This finer-grained substitution operation is called **linear substitution**.

Accattoli and Dal Lago have proposed an optimised evaluation mechanism, called **useful evaluation** [6], which represents terms with ESs so that the copy of shared subterms is restricted to avoid *size explosion*. Useful evaluation has in particular been extended to *open CBV* [2,4], *i.e.*, CBV over terms that may contain free variables. This mechanism relies on two key *usefulness principles* that we call `UP1` and `UP2`. Specifically, these two principles ensure that an occurrence of a variable may be linearly substituted by an expression only if this contributes to creating a *redex*[5]. Principle `UP1` (*Sharing Structures*) ensures that terms headed by a free variable, or more precisely *structures*, such as $x\,y$, always remain shared in ESs, and are never copied. Principle `UP2` (*Substituting Abstractions for Progress*) ensures that an occurrence of a variable bound to an abstraction may be substituted only if the variable is applied to an argument. See Section 2 for more details.

An Inductive Approach to Useful Evaluation. It is well-known that there are two alternative ways to specify evaluation strategies. A strategy can be presented *inductively*, by providing *computation rules*, that specify how terms interact locally, and *congruence rules*, that specify when and how evaluation should focus on a subterm. Regarding congruence rules, another point of view is to rely on *evaluation contexts*, to specify the strategy at top-level.[6]

[4] See [14] for a survey on ESs.

[5] **Red**ucible **ex**pression.

[6] For example, CBN evaluation may be specified with a computation rule for β-reduction, declaring that $(\lambda x.\, t)\, s \to_{\mathsf{CBN}} t\{x := s\}$, and a congruence rule to allow evaluation to proceed on the left of an application, declaring that $t \to_{\mathsf{CBN}} t'$ implies $t\,s \to_{\mathsf{CBN}} t'\,s$. An alternative way to specify CBN evaluation is with a single top-level

It is usually easy to switch between these two points of view, because typical strategies only rely on *local* interactions. However, this is not the case with useful evaluation, which is context-sensitive, in the sense that steps involve side conditions of a *global* nature. In particular, a non-useful step can become useful when placed within a larger context. For example, the step $\mathtt{C}\langle x[x\backslash y]\rangle \to \mathtt{C}\langle y[x\backslash y]\rangle$, which substitutes x by y under a context $\mathtt{C}$, may be useful or not, depending on whether the expression $x[x\backslash y]$ is applied to an argument, and whether the variable y is (hereditarily) bound to an abstraction in $\mathtt{C}$. This makes it challenging to define useful evaluation by means of congruence rules, which are of inductive nature. As part of this work, we define an inductive specification of useful evaluation, which despite the difficulties faced, it provides several advantages, which we will discuss at the end of this paper.

Contributions. This paper defines the first inductive specification of useful evaluation. We work in the framework of open CBV. Although most functional programming languages are restricted to closed terms (forbidding free variables), working in an open setting (allowing free variables) broadens the applicability of our results. Furthermore, being able to treat the open CBV case is a prerequisite to deal in the future with other cases. For example, Grégoire and Leroy study *strong* CBV evaluation [13], motivated by the implementation of proof assistants based on dependent type theory.

In contrast to existing specifications of useful evaluation in the literature, our specification relies on *inductive* rules. To define the new strategy UCBV, we proceed in two stages. First, we recall in **Section 3** the *Value Substitution Calculus* (VSC) [7], which is the starting point of our development. We then introduce the *linear* CBV calculus (LCBV), which refines the VSC with *linear substitution*, while remaining observationally equivalent. Moreover, LCBV implements principle `UP1` but still not `UP2`. Second, we introduce in **Section 4** the UCBV strategy, by refining evaluation in LCBV to also implement principle `UP2`. Our inductive approach to usefulness has been inspired by the definition of strong call-by-need evaluation in [8]. We also prove key operational properties of UCBV; in particular, it enjoys the diamond property —and thus confluence. In **Section 5**, we show that UCBV is indeed a useful implementation of LCBV. The relationship between the two calculi is established syntactically, by means of rewriting techniques, and relying on *partial unfolding* operation that performs *all* the substitutions assigning values to variables. The main result is that UCBV preserves the semantics of LCBV, *i.e.*, if LCBV reaches a normal form from a starting term then, UCBV reaches the same normal form, up to partial unfolding.

Finally, in **Section 6**, we show that UCBV can in turn be implemented by a lower-level abstract machine for useful CBV, namely the GLAMoUr [2,5]. This result not only connects UCBV with a preexisting notion of usefulness, but it also entails that UCBV is *time-invariant*. This means that the length of reductions to normal form can be taken as a measure of time complexity. More precisely,

---

rule declaring that $\mathtt{N}\langle(\lambda x.\, t)\, s\rangle \to_{\mathsf{CBN}} \mathtt{N}\langle t\{x := s\}\rangle$, where $\mathtt{N}$ denotes a CBN evaluation context generated by the grammar $\mathtt{N} ::= \diamond \mid \mathtt{N}\, t$.

UCBV can be simulated by a *reasonable* cost model of computation [15] (such as a Turing machine or a RAM) with at most polynomial overhead in time.

Note. Proofs have been omitted, but they can be found in the technical report [10].

2 Preliminary Notions

In this section, we introduce the syntax used for all the calculi introduced in this work. Next, we explain the two principles UP1 and UP2 that underlie the notion of usefulness. Moreover, we recall some background notions of rewriting theory.

Syntax. Given a denumerable set of **variables** $(x, y, z, \ldots)$, the sets of **terms** $(t, s, u, \ldots)$, **substitution contexts** $(\mathtt{L}, \mathtt{L}', \ldots)$, and **values** $(v, w, \ldots)$ are given by the following grammars:

$$t ::= x \mid \lambda x.\, t \mid t\, t \mid t[x\backslash t] \qquad \mathtt{L} ::= \Diamond \mid \mathtt{L}[x\backslash t] \qquad v ::= x \mid \lambda x.\, t$$

The set of terms includes **variables**, **abstractions**, **applications**, and **closures** $t[x\backslash s]$ representing an **explicit substitution** (ES) $[x\backslash s]$ on a term t. **Free** and **bound occurrences** of variables are defined as usual, where free occurrences of x in t are bound in $t[x\backslash s]$. We write $\mathsf{fv}(t)$ and $\mathsf{fv}(\mathtt{L})$ for the set of free variables of a term and a context, respectively. Terms are considered up to *α-renaming* of bound variables. We write $t\mathtt{L}$ for the *variable-capturing* **replacement** of the hole $\Diamond$ in $\mathtt{L}$ by t. If $\mathtt{L} = \Diamond[x_1\backslash t_1] \ldots [x_n\backslash t_n]$, we write $\mathsf{dom}(\mathtt{L})$ for the set of variables that are bound by $\mathtt{L}$ at the place of the hole, that is, $\mathsf{dom}(\mathtt{L}) := \{x_1, \ldots, x_n\}$. The set of **reachable variables** of a term is defined as follows: $\mathsf{rv}(x) := \{x\}$, $\mathsf{rv}(\lambda x.\, t) := \emptyset$, $\mathsf{rv}(t\, s) := \mathsf{rv}(t) \cup \mathsf{rv}(s)$, and $\mathsf{rv}(t[x\backslash s]) := (\mathsf{rv}(t) \setminus \{x\}) \cup \mathsf{rv}(s)$. We write $t\{x := s\}$ for the *capture-avoiding* **substitution** of the free occurrences of x by s in t. Some recurring terms are the identity function $\mathtt{I} := \lambda x.\, x$ and the operator $\omega := \lambda x.\, x\, x$.

Usefulness Principles. As mentioned in the introduction, useful evaluation is motivated by the two following key *usefulness principles*:

Sharing Structures (UP1). A term is a **structure** if its *(full) unfolding* —the result of performing all the explicit substitutions by using the capture-avoiding substitution— results in an application headed by a variable, *i.e.*, of the form $x\, t_1 \ldots t_n$, where $(n \geq 0)$. For example, $(x\, x)[x\backslash y\, \mathtt{I}][y\backslash z\, z]$ is a structure, as it (fully) unfolds to $z\, z\, \mathtt{I}\, (z\, z\, \mathtt{I})$. Structures must remain *shared* in useful evaluation, as substituting a variable by a structure does not create a *function application redex*. Thus, a term like $(x\, x)[x\backslash y\, \mathtt{I}][y\backslash z\, z]$ is a normal form in useful evaluation. The notion of structure just defined is context-independent. In fact, we shall need a subtler *context-dependent* notion of structure relative to a "structure frame" (defined in Section 4). For instance, $x\, y$ should *not* be treated as a structure under a context that binds x to an abstraction.

Substituting Abstractions for Progress (UP2). In useful evaluation, abstractions are substituted only if they contribute to creating a function application redex, thus ensuring progress in the computation. For example, a reduction step like $x[x\backslash \mathtt{I}]\, y \to \mathtt{I}[x\backslash \mathtt{I}]\, y$ is useful, while the steps $x[x\backslash \mathtt{I}] \to \mathtt{I}[x\backslash \mathtt{I}]$ and $(t\, x)[x\backslash \mathtt{I}] \to (t\, \mathtt{I})[x\backslash \mathtt{I}]$ are not, as they do not contribute to creating function application redexes. Some of these ideas can also be found in the literature on *optimal reduction* (see *e.g.*, [17]).

Background on Rewriting Theory. We now introduce some general notions of reduction that will be used in the paper. Given a **reduction system** $\mathcal{R}$, we denote by $\to_{\mathcal{R}}$ the (one-step) reduction relation associated to system $\mathcal{R}$. We write $\to_{\mathcal{R}}^{=}$ and $\to_{\mathcal{R}}^{*}$ for the reflexive and reflexive-transitive closure of $\to_{\mathcal{R}}$, and $\to_{\mathcal{R}}^{n}$ for the composition of n-steps of $\to_{\mathcal{R}}$. A term t is said to be **$\mathcal{R}$-reducible** if there is s such that $t \to_{\mathcal{R}} s$, and t is said to be **$\mathcal{R}$-irreducible**, or in **$\mathcal{R}$-normal form**, written $t \not\to_{\mathcal{R}}$, if there is no s such that $t \to_{\mathcal{R}} s$. A term t is said to be **$\mathcal{R}$-terminating** if there is no infinite $\mathcal{R}$-sequence starting at t. A term t is **$\mathcal{R}$-diamond** (or enjoys the $\mathcal{R}$-diamond property) if $t \to_{\mathcal{R}} t_0$ and $t \to_{\mathcal{R}} t_1$ with $t_0 \neq t_1$ imply there exists t' such that $t_0 \to_{\mathcal{R}} t'$ and $t_1 \to_{\mathcal{R}} t'$. A term t is **$\mathcal{R}$-confluent** if $t \to_{\mathcal{R}}^{*} t_0$ and $t \to_{\mathcal{R}}^{*} t_1$ imply there exists t' such that $t_0 \to_{\mathcal{R}}^{*} t'$ and $t_1 \to_{\mathcal{R}}^{*} t'$. A relation $\mathcal{R}$ is **terminating**, (resp. **diamond**, **confluent**) if and only if every term is $\mathcal{R}$-terminating (resp. $\mathcal{R}$-diamond, $\mathcal{R}$-confluent). Any relation verifying the diamond property is in particular confluent. Moreover, if t is confluent, then its $\mathcal{R}$-normal form, if it exists, is unique [16].

3 Linear Call-by-Value

In this section, we define the Linear CBV calculus (LCBV), a first step towards useful evaluation. LCBV fulfils principle UP1 by keeping structures always shared, but it does not fulfil UP2, as it allows substituting variables by values unrestrictedly (see Section 2 for the definitions of UP1 and UP2).

Furthermore, LCBV adapts the substitution operation in the Value Substitution Calculus (VSC) [7] to be *linear* by allowing the substitution of one occurrence of a variable at a time. We begin by explaining the difference between the (non-linear) original substitution mechanism of the VSC, and the linear substitution of LCBV. We then formally define LCBV and conclude by stating its main properties, including the observational equivalence between LCBV and VSC.

The Value Substitution Calculus. Avoiding the substitution of variables by structures is a sufficient mechanism to obtain an invariant implementation of open CBV (see *e.g.*, [4, Lemma 6]). One calculus implementing this mechanism for open terms is the VSC, which consists of two kinds of reduction steps:

Distant beta (db) steps are given by the rule $(\lambda x.\, t)\mathtt{L}\, s \to t[x\backslash s]\mathtt{L}$, where L denotes a list of ES, called a *substitution context*. A db-step performs a *function application* by creating an ES. For example, $(\lambda x.\, \mathtt{I})\, (y\, y)\, z \to \mathtt{I}[x\backslash y\, y]\, z \to w[w\backslash z][x\backslash y\, y]$. After the first step, the ES $[x\backslash y\, y]$ appears to block the interaction between I and its argument z. However, the db-rule allows an arbitrary list

of ESs to occur between an abstraction and its argument, thus allowing to fire the second reduction step in the example.

Substitution steps (sv) are given by the rule $t[x\backslash v\mathtt{L}] \to t\{x := v\}\mathtt{L}$, which substitutes a variable x by a value v, *pushing outside* the substitution context $\mathtt{L}$ originally accompanying the value. For example, $(x\,x)[x\backslash\omega[y\backslash\mathtt{I}]] \to (\omega\,\omega)[y\backslash\mathtt{I}]$, where we recall that $\omega := \lambda z.\, z\,z$. The VSC implements principle UP1 because it only allows substituting variables by values (so that structures remain shared). For example, the step $(x\,x)[x\backslash z\,z] \to z\,z\,(z\,z)$ is not allowed.

Towards Linear Substitution. LCBV refines the VSC by implementing *linear substitution*, which substitutes one occurrence of a variable at a time. Linear substitution is a prerequisite —but not sufficient— to fulfil principle UP2, *i.e.*, to *fully* implement useful evaluation. Specifically, when a variable occurs multiple times in a term, substituting it with an abstraction may create a function application in some cases but not in others. For example, in the step $(z\,(\underline{x}\,y)\,\overline{x})[x\backslash\mathtt{I}] \to z\,(\mathtt{I}\,y)\,\mathtt{I}$, substituting the underlined occurrence of x by $\mathtt{I}$ creates a db-redex (and thus it is useful), while substituting the overlined occurrence of x by $\mathtt{I}$ does not contribute to creating a db-redex.

The Linear Call-by-Value Calculus. We now define the Linear CBV Calculus LCBV, which refines the VSC by implementing linear substitution. We adopt a formulation of rules based on the style of [8] that we also follow to formulate our inductive notion of useful evaluation, UCBV, in Section 4.

Formally, we define a family of binary relations $\xrightarrow{\circ}_\rho$, where $\rho \in \{\mathsf{db}, \mathsf{lsv}, \mathsf{sub}_{(x,v)}\}$ distinguishes the **step kind**, where x is a variable and v is a value such that $x \notin \mathsf{fv}(v)$. The set of **free variables** of a step kind ρ is defined as $\mathsf{fv}(\mathsf{db}) := \emptyset$, $\mathsf{fv}(\mathsf{lsv}) := \emptyset$, and $\mathsf{fv}(\mathsf{sub}_{(x,v)}) := \{x\} \cup \mathsf{fv}(v)$. The **reduction relation** $\xrightarrow{\circ}_\rho$ of LCBV is inductively defined as follows:

$$\frac{}{(\lambda x.\, t)\mathtt{L}\; s \xrightarrow{\circ}_{\mathsf{db}} t[x\backslash s]\mathtt{L}}\,\mathrm{DB}^\circ \qquad \frac{}{x \xrightarrow{\circ}_{\mathsf{sub}_{(x,v)}} v}\,\mathrm{SUB}^\circ \qquad \frac{t \xrightarrow{\circ}_{\mathsf{sub}_{(x,v)}} t'}{t[x\backslash v\mathtt{L}] \xrightarrow{\circ}_{\mathsf{lsv}} t'[x\backslash v]\mathtt{L}}\,\mathrm{LSV}^\circ$$

$$\frac{t \xrightarrow{\circ}_\rho t'}{t\,s \xrightarrow{\circ}_\rho t'\,s}\,\mathrm{APPL}^\circ \qquad \frac{s \xrightarrow{\circ}_\rho s'}{t\,s \xrightarrow{\circ}_\rho t\,s'}\,\mathrm{APPR}^\circ \qquad \frac{t \xrightarrow{\circ}_\rho t' \quad x \notin \mathsf{fv}(\rho)}{t[x\backslash s] \xrightarrow{\circ}_\rho t'[x\backslash s]}\,\mathrm{ESL}^\circ \qquad \frac{s \xrightarrow{\circ}_\rho s'}{t[x\backslash s] \xrightarrow{\circ}_\rho t[x\backslash s']}\,\mathrm{ESR}^\circ$$

We define **top-level** LCBV reduction as $\xrightarrow{\circ}_{\mathsf{top}} := \xrightarrow{\circ}_{\mathsf{db}} \cup \xrightarrow{\circ}_{\mathsf{lsv}}$. Rules DB°, SUB°, and LSV° define the three kinds of reduction steps, whereas APPL°, APPR°, ESL°, and ESR° are congruence rules, and propagate any step kind. Note that evaluation is *weak* and does not proceed under abstractions. A step of the form $t \xrightarrow{\circ}_{\mathsf{db}} s$ represents a *distant beta* step. The reduction steps $t \xrightarrow{\circ}_{\mathsf{sub}_{(x,v)}} s$ and $t \xrightarrow{\circ}_{\mathsf{lsv}} s$ constitute two kinds of *substitution* steps. In the first kind, $t \xrightarrow{\circ}_{\mathsf{sub}_{(x,v)}} s$, *one* free occurrence of x in t is substituted by v. In the second kind, $t \xrightarrow{\circ}_{\mathsf{lsv}} s$, *one* bound occurrence of x is substituted by v, provided x is bound to a term of the form $v\mathtt{L}$ by an ES; LSV° is the only rule that introduces an lsv-step. Moreover, each step substituting a bound variable ($\xrightarrow{\circ}_{\mathsf{lsv}}$) relies internally on a step that substitutes a free variable ($\xrightarrow{\circ}_{\mathsf{sub}_{(x,v)}}$). These substitution steps focus on a *single* variable occurrence, thus we say the substitution operation is *linear*.

As a technical point, note that the term s lies outside the scope of the substitution context $\mathtt{L}$ on the left-hand side of the db rule, and inside the scope of $\mathtt{L}$ on the right-hand side. In these cases, we assume that this does not cause a free variable of s to be captured by $\mathtt{L}$, *i.e.* that the variables bound by $\mathtt{L}$ have been α-renamed in such a way that $\mathsf{fv}(s) \cap \mathsf{dom}(\mathtt{L}) = \emptyset$. For example, $(\lambda y.\, x)[x\backslash t]\, x \xrightarrow{\circ}_{\mathsf{db}} z[y\backslash x][z\backslash t]$.

Along with rule DB°, an application can be evaluated using rules APPL° and APPR°, which reduce within the left and right subterms, respectively. Since rules DB°, APPL°, and APPR° overlap, reduction is *non-deterministic*. For example:

$$x[x\backslash \mathtt{I}]\,(\mathtt{I}\,\mathtt{I}) \xleftarrow{\circ}_{\mathsf{db}} \mathtt{I}\,\mathtt{I}\,(\mathtt{I}\,\mathtt{I}) \xrightarrow{\circ}_{\mathsf{db}} \mathtt{I}\,\mathtt{I}\,x[x\backslash \mathtt{I}]$$

The congruence rules ESL° and ESR° allow reduction on the left side of a term and within the argument of an ES, respectively. These rules overlap as well. A key restriction in rule ESL° is that the variable x bound by the ES $[x\backslash s]$ must not occur free in the step kind ρ. This restriction is to avoid variable capture; for example, it ensures that a "pathological" step like $y[x\backslash z] \xrightarrow{\circ}_{\mathsf{sub}_{(y,x)}} x[x\backslash z]$ cannot be derived from the valid step $y \xrightarrow{\circ}_{\mathsf{sub}_{(y,x)}} x$.

Example 1. The following is a reduction sequence to normal form in LCBV:

$$\begin{array}{l} (\lambda x.\, z\, x\, (x\, y))\, \mathtt{I} \xrightarrow{\circ}_{\mathsf{db}} (z\, x\, (x\, y))[x\backslash \mathtt{I}] \xrightarrow{\circ}_{\mathsf{lsv}} (z\, \mathtt{I}\, (x\, y))[x\backslash \mathtt{I}] \\ \xrightarrow{\circ}_{\mathsf{lsv}} (z\, \mathtt{I}\, (\mathtt{I}\, y))[x\backslash \mathtt{I}] \xrightarrow{\circ}_{\mathsf{db}} (z\, \mathtt{I}\, (w[w\backslash y]))[x\backslash \mathtt{I}] \xrightarrow{\circ}_{\mathsf{lsv}} (z\, \mathtt{I}\, (y[w\backslash y]))[x\backslash \mathtt{I}] \end{array}$$

Confluence. Despite the overlaps of some reduction rules, it is straightforward to show that *top-level* $\xrightarrow{\circ}$ reduction is confluent:

Proposition 1. *Reduction* $\xrightarrow{\circ}_{\mathsf{top}}$ *is confluent.*

Proof. The proof follows from [10, Theorem C.14].

Correspondence with the Value Substitution Calculus. We formally relate the LCBV to the VSC.

Proposition 2. *Let t be a term. Then, t terminates in* LCBV *if and only if t terminates in the* VSC.

Proof. The proof of this statement follows a standard approach using semantic tools from *(intersection) type theory*. As it falls outside the scope of this paper, we do not include the proof here. Full details can be found in [9].

In the following section, we define *useful CBV* evaluation by refining LCBV, for which we incorporate the second key principle of usefulness, *substituting abstractions for progress* ($\mathsf{UP2}$).

4 Useful Call-by-Value

In this section, we refine the notion of LCBV evaluation introduced in Section 3, the resulting strategy is called UCBV. While preserving principle `UP1`, already present in LCBV, UCBV evaluation also implements principle `UP2`, which is non-trivial to be captured *inductively*. To achieve useful CBV evaluation, we define a family of reduction relations indexed by certain *parameters* representing the essential information from the surrounding evaluation contexts. This is the minimal data necessary to decide whether a substitution step is useful or not. Next, we give the main properties of UCBV, starting with a characterisation of normal forms (Theorem 1). Following this result, we show that UCBV satisfies the diamond property (Theorem 2), which ensures in particular that the length of a reduction sequence to normal form does not depend on the specific order of redexes chosen to reduce the terms.

Towards the Useful Call-by-Value Strategy. To implement principle `UP2` in UCBV it is necessary to ensure that substitution contributes to the *progress* of the evaluation. For example, the substitution step $(x\,t)[x\backslash y][y\backslash \mathtt{I}] \to (y\,t)[x\backslash y][y\backslash \mathtt{I}]$ is (indirectly) useful because the occurrence of x is applied to an argument t, and substituting x by y then contributes to creating the underlined redex $(\underline{\mathtt{I}\,t})[x\backslash y][y\backslash \mathtt{I}]$ after one more substitution step. This motivates the need to identify *hereditary abstractions*, which intuitively include abstractions and variables hereditarily bound to abstractions, such as y in the example. Formally, let $\mathcal{A}$ be a set of variables, called an **abstraction frame**. The set of **hereditary abstractions** under $\mathcal{A}$ is written $\mathsf{HA}_{\mathcal{A}}$ and defined below:

$$\frac{}{\lambda x.\, t \in \mathsf{HA}_{\mathcal{A}}} \qquad \frac{x \in \mathcal{A}}{x \in \mathsf{HA}_{\mathcal{A}}} \qquad \frac{t \in \mathsf{HA}_{\mathcal{A}} \quad x \notin \mathcal{A}}{t[x\backslash s] \in \mathsf{HA}_{\mathcal{A}}} \qquad \frac{t \in \mathsf{HA}_{\mathcal{A}\cup\{x\}} \quad x \notin \mathcal{A} \quad s \in \mathsf{HA}_{\mathcal{A}}}{t[x\backslash s] \in \mathsf{HA}_{\mathcal{A}}}$$

The abstraction frame $\mathcal{A}$ keeps track of variables bound to hereditary abstractions. Every abstraction is a hereditary abstraction (by the first rule), while a variable is a hereditary abstraction if it appears in the abstraction frame (by the second rule). In the case of terms with ESs, the crucial point is that the bound variable x may be added to the abstraction frame only if it is bound to a hereditary abstraction (third and fourth rule). For example, $x[x\backslash \lambda y.\, y] \in \mathsf{HA}_{\mathcal{A}}$ for any $\mathcal{A}$, and $x[z\backslash w][x\backslash y] \in \mathsf{HA}_{\mathcal{A}}$ if and only if $y \in \mathcal{A}$. Note also that applications are never hereditary abstractions.

On the other hand, it is necessary to identify *structures*, which intuitively include both applied variables and variables hereditarily bound to structures. Formally, let $\mathcal{S}$ be a set of variables, called a **structure frame**. The set of **structures under** $\mathcal{S}$ is written $\mathsf{St}_{\mathcal{S}}$ and defined below:

$$\frac{x \in \mathcal{S}}{x \in \mathsf{St}_{\mathcal{S}}} \qquad \frac{t \in \mathsf{St}_{\mathcal{S}}}{t\,s \in \mathsf{St}_{\mathcal{S}}} \qquad \frac{t \in \mathsf{St}_{\mathcal{S}} \quad x \notin \mathcal{S}}{t[x\backslash s] \in \mathsf{St}_{\mathcal{S}}} \qquad \frac{t \in \mathsf{St}_{\mathcal{S}\cup\{x\}} \quad x \notin \mathcal{S} \quad s \in \mathsf{St}_{\mathcal{S}}}{t[x\backslash s] \in \mathsf{St}_{\mathcal{S}}}$$

The structure frame $\mathcal{S}$ keeps track of variables that are bound to structures. Note that no abstraction belongs to the set of structures, for any $\mathcal{S}$. An application is

a structure if its left subterm is itself a structure, thus excluding db-redexes from the set of structures under any $\mathcal{S}$. Variables in the structure frame are structures as well. For terms with ESs, the intuitions behind the third and fourth rules are analogous to the corresponding ones for the predicate HA, respectively. For example, performing the substitution on the left subterm in $x[x\backslash y_1\, y_2]\, z$ does not create any db-redex, so $x[x\backslash y_1\, y_2]$ is considered a structure. Similarly, $(x\, y)[y\backslash \mathtt{I}]$ is a structure, even if y is bound to an abstraction.

The abstraction frame $\mathcal{A}$ and the structure frame $\mathcal{S}$ are assumed to be disjoint. To evaluate a term, one should always assume that $\mathsf{fv}(t) \subseteq \mathcal{A} \cup \mathcal{S}$, *i.e.*, all free variables are assumed to be bound to either a hereditary abstraction or a structure. At the top-level case, *i.e.*, when evaluating an *isolated* term t, one takes $\mathcal{A} := \emptyset$ and $\mathcal{S} := \mathsf{fv}(t)$ as starting conditions.

The Useful Call-by-Value Reduction. We can now formally define the UCBV strategy specified as a family of binary relations $\xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu}$, where $\rho \in \{\mathsf{db}, \mathsf{lsv}, \mathsf{sub}_{(x,v)}\}$ is a step kind, $\mathcal{A}$ is an abstraction frame, $\mathcal{S}$ is a structure frame, and $\mu \in \{@, @\!\!\!/\}$ is a positional flag. The **reduction relation** $\xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu}$ of UCBV is inductively defined as follows:

$$\frac{}{(\lambda x.\, t)\mathtt{L}\, s \xrightarrow{\bullet}_{\mathsf{db},\mathcal{A},\mathcal{S},\mu} t[x\backslash s]\mathtt{L}}\,\mathrm{DB}^{\bullet} \qquad \frac{}{x \xrightarrow{\bullet}_{\mathsf{sub}_{(x,v)},\mathcal{A}\cup\{x\},\mathcal{S},@} v}\,\mathrm{SUB}^{\bullet}$$

$$\frac{t \xrightarrow{\bullet}_{\mathsf{sub}_{(x,v)},\mathcal{A}\cup\{x\},\mathcal{S},\mu} t' \quad x \notin \mathcal{A}\cup\mathcal{S} \quad v\mathtt{L} \in \mathsf{HA}_{\mathcal{A}}}{t[x\backslash v\mathtt{L}] \xrightarrow{\bullet}_{\mathsf{lsv},\mathcal{A},\mathcal{S},\mu} t'[x\backslash v]\mathtt{L}}\,\mathrm{LSV}^{\bullet}$$

$$\frac{t \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},@} t'}{t\, s \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu} t'\, s}\,\mathrm{APPL}^{\bullet} \qquad \frac{t \in \mathsf{St}_{\mathcal{S}} \quad s \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},@\!\!\!/} s'}{t\, s \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu} t\, s'}\,\mathrm{APPR}^{\bullet}$$

$$\frac{s \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},@\!\!\!/} s'}{t[x\backslash s] \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu} t[x\backslash s']}\,\mathrm{ESR}^{\bullet}$$

$$\frac{t \xrightarrow{\bullet}_{\rho,\mathcal{A}\cup\{x\},\mathcal{S},\mu} t' \quad s \in \mathsf{HA}_{\mathcal{A}} \quad x \notin \mathcal{A}\cup\mathcal{S} \quad x \notin \mathsf{fv}(\rho)}{t[x\backslash s] \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu} t'[x\backslash s]}\,\mathrm{ESLA}^{\bullet}$$

$$\frac{t \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S}\cup\{x\},\mu} t' \quad s \in \mathsf{St}_{\mathcal{S}} \quad x \notin \mathcal{A}\cup\mathcal{S} \quad x \notin \mathsf{fv}(\rho)}{t[x\backslash s] \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu} t'[x\backslash s]}\,\mathrm{ESLS}^{\bullet}$$

Note that each UCBV step is also a LCBV step, *i.e.*, $\xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu} \subseteq \xrightarrow{\circ}_{\rho}$. Also, the reduction relation defined above is **non-erasing**: if $t \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu} t'$ with $\rho \in \{\mathsf{db}, \mathsf{lsv}\}$, then $\mathsf{fv}(t) = \mathsf{fv}(t')$. As in the previous section, rules $\mathrm{DB}^{\bullet}$, $\mathrm{SUB}^{\bullet}$, and $\mathrm{LSV}^{\bullet}$ introduce the three possible step kinds, while all other cases ($\mathrm{APPL}^{\bullet}$, $\mathrm{APPR}^{\bullet}$, $\mathrm{ESLA}^{\bullet}$, $\mathrm{ESLS}^{\bullet}$, and $\mathrm{ESR}^{\bullet}$) are congruence rules for steps of an arbitrary step kind ρ. Reduction is *weak* since there are no rules to evaluate under abstractions.

Rule $\mathrm{DB}^{\bullet}$ performs a function application step, identical to rule DB° in LCBV. Steps of the form $t \xrightarrow{\bullet}_{\mathsf{sub}_{(x,v)},\mathcal{A},\mathcal{S},\mu} s$ and $t \xrightarrow{\bullet}_{\mathsf{lsv},\mathcal{A},\mathcal{S},\mu} s$ represent two different kinds of *substitution steps*, as in LCBV: the former substitutes *one* free occurrence of a variable, while the latter substitutes *one* bound occurrence. Only rule $\mathrm{LSV}^{\bullet}$

creates an lsv-step. Each application of rule LSV$^\bullet$ relies internally on a $\mathsf{sub}_{(x,v)}$-step, where v must be a hereditary abstraction, a restriction not required in rule LSV$^\circ$ rule in LCBV, but that it is crucial to implement principle UP2 (*substituting abstractions for progress*).

Substituting a free variable in a term t using rule SUB$^\bullet$ is only possible when t is in an applied position because this ensures the progress of the computation. The following example shows an instance of the SUB$^\bullet$ rule:

$$\dfrac{\dfrac{\dfrac{}{x \xrightarrow{\bullet}_{\mathsf{sub}_{(x,\mathtt{I})},\{x\},\{z\},@} \mathtt{I}}\,\text{SUB}^\bullet \qquad \mathtt{I}[y\backslash z] \in \mathsf{HA}_\emptyset}{x[x\backslash \mathtt{I}[y\backslash z]] \xrightarrow{\bullet}_{\mathsf{lsv},\emptyset,\{z\},@} \mathtt{I}[x\backslash \mathtt{I}][y\backslash z]}\,\text{LSV}^\bullet}{x[x\backslash \mathtt{I}[y\backslash z]]w \xrightarrow{\bullet}_{\mathsf{lsv},\emptyset,\{z\},@\!\!} \mathtt{I}[x\backslash \mathtt{I}][y\backslash z]w}\,\text{APPL}^\bullet$$

The congruence rules APPL$^\bullet$ and APPR$^\bullet$ perform reduction on the application constructor. Rule APPR$^\bullet$ additionally requires the left subterm of the application to be a structure, avoiding a possible overlap with rule DB$^\bullet$. Still, rules APPL$^\bullet$ and APPR$^\bullet$ overlap, so reduction is non-deterministic, like in LCBV. For example:

$$x\,y[y\backslash \mathtt{I}]\,(\mathtt{I}\,\mathtt{I}) \xleftarrow{\bullet}_{\mathsf{db},\emptyset,\{x\},@\!\!} x\,(\mathtt{I}\,\mathtt{I})\,(\mathtt{I}\,\mathtt{I}) \xrightarrow{\bullet}_{\mathsf{db},\emptyset,\{x\},@\!\!} x\,(\mathtt{I}\,\mathtt{I})\,y[y\backslash \mathtt{I}]$$

The congruence rules ESR$^\bullet$, ESLA$^\bullet$, and ESLS$^\bullet$ apply to closures. Rule ESR$^\bullet$ allows reducing the argument of any ESs, while rules ESLA$^\bullet$ and ESLS$^\bullet$ allow reduction of the left side of the closure under specific conditions: ESLA$^\bullet$ (resp. ESLS$^\bullet$) applies only if the substitution argument is a hereditary abstraction (resp. a structure). Note that rules ESLA$^\bullet$ and ESLS$^\bullet$ force reducing the argument s of a closure $t[x\backslash s]$ until it becomes "stuck", after which evaluation can proceed to the body t. This behaviour aligns with a CBV strategy, covering all cases in terminating terms, as every terminating term is eventually either a hereditary abstraction or a structure ([10, Lemma B.4]). These rules also enforce that the variable x bound by the ES $[x\backslash s]$ does not occur free in the step kind ρ, as in rule ESL$^\circ$ in Section 3. This restriction prevents variable capture in steps like $y[x\backslash \mathtt{I}] \xrightarrow{\bullet}_{\mathsf{sub}_{(y,x)},\{y\},\emptyset,@} x[x\backslash \mathtt{I}]$. Note that there is a possible overlap between rule ESR$^\bullet$ and rules ESLA$^\bullet$ and ESLS$^\bullet$.

A term t is $(\rho, \mathcal{A}, \mathcal{S}, \mu)$**-reducible** if there exists a term t' such that $t \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu} t'$. A term t belongs to the set $\mathsf{Irred}^\bullet_{\mathcal{A},\mathcal{S},\mu}$ if t is not $(\rho, \mathcal{A}, \mathcal{S}, \mu)$-reducible.

Hereditary abstractions and structures show distinct and disjoint behaviours within UCBV, which are put in evidence through the distinction between abstraction frames and structure frames. An abstraction frame $\mathcal{A}$ and a structure frame $\mathcal{S}$ are said to verify the **correctness property** for t, written $\mathsf{cp}(\mathcal{A}, \mathcal{S}, t)$, if $\mathcal{A} \cap \mathcal{S} = \emptyset$ and $\mathsf{fv}(t) \subseteq \mathcal{A} \cup \mathcal{S}$. This property is implicitly assumed in theorems and lemmas, *i.e.*, whenever we write $t \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu}$, we always keep $\mathsf{cp}(\mathcal{A}, \mathcal{S}, t)$ as an invariant. Note that $\mathsf{cp}(\emptyset, \mathsf{fv}(t), t)$ always holds, so for a top-level term t, we may take $\mathcal{A} := \emptyset$ and $\mathcal{S} := \mathsf{fv}(t)$.

Example 2. The following is an UCBV reduction sequence to normal form:

$$\begin{array}{c}(\lambda x.\, z\,x\,(x\,y))\,\mathtt{I} \xrightarrow{\bullet}_{\mathsf{db},\emptyset,\{z,y\},@\!\!} (z\,x\,(x\,y))[x\backslash \mathtt{I}] \\ \xrightarrow{\bullet}_{\mathsf{lsv},\emptyset,\{z,y\},@\!\!} (z\,x\,(\mathtt{I}\,y))[x\backslash \mathtt{I}] \xrightarrow{\bullet}_{\mathsf{db},\emptyset,\{z,y\},@\!\!} (z\,x\,(w[w\backslash y]))[x\backslash \mathtt{I}]\end{array}$$

Compare this reduction sequence with the corresponding evaluation of the same term in LCBV (Example 1). In UCBV, the leftmost occurrence of x and the occurrence of w are not substituted, because these substitutions would not satisfy principle UP2: they do not contribute to creating a db-redex.

As discussed in Section 1, useful evaluation is *context-sensitive*, as the context surrounding a term determines whether a step is useful or not. For instance, the term $t = x[x\backslash \mathtt{I}]$ is already in normal form in UCBV, as substituting x by $\mathtt{I}$ is not useful, because it does not create a db-redex. However, if t is located to the left of an application, such as in $x[x\backslash \mathtt{I}]\,(y\,\mathtt{I})$, the substitution of x by $\mathtt{I}$ becomes useful, and yields the following reduction sequence:

$$x[x\backslash \mathtt{I}]\,(y\,\mathtt{I}) \xrightarrow{\bullet}_{\mathsf{lsv},\emptyset,\{y\},\not\emptyset} \mathtt{I}[x\backslash \mathtt{I}]\,(y\,\mathtt{I}) \xrightarrow{\bullet}_{\mathsf{db},\emptyset,\{y\},\not\emptyset} z[z\backslash y\,\mathtt{I}][x\backslash \mathtt{I}]$$

Conversely, a useful step like $(x\,s)[x\backslash \mathtt{I}] \xrightarrow{\bullet}_{\mathsf{lsv},\emptyset,\emptyset,\not\emptyset} (\mathtt{I}\,s)[x\backslash \mathtt{I}]$ may not be allowed if the term is located below a context such as $\lambda y.\,\diamond$; in fact, note that $\lambda y.\,(x\,s)[x\backslash \mathtt{I}]$ cannot be reduced.

To conclude this section, we show some properties of UCBV beginning with an inductive characterisation of the set of normal forms:

Theorem 1 (Characterisation of Normal Forms). *The set of irreducible terms* $\mathsf{Irred}^{\bullet}_{\mathcal{A},\mathcal{S},\mu}$ *is exactly the set* $\mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},\mu}$ *inductively defined as:*

$$\frac{x \in \mathcal{A} \Rightarrow \mu = \not\emptyset}{x \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},\mu}}\text{NF-VAR}^{\bullet} \qquad \frac{}{\lambda x.\,t \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},\not\emptyset}}\text{NF-LAM}^{\bullet}$$

$$\frac{t \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},@} \quad s \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},\not\emptyset}}{t\,s \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},\mu}}\text{NF-APP}^{\bullet}$$

$$\frac{t \in \mathsf{NF}^{\bullet}_{\mathcal{A}\cup\{x\},\mathcal{S},\mu} \quad s \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},\not\emptyset} \quad s \in \mathsf{HA}_{\mathcal{A}}}{t[x\backslash s] \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},\mu}}\text{NF-ESA}^{\bullet}$$

$$\frac{t \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S}\cup\{x\},\mu} \quad s \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},\not\emptyset} \quad s \in \mathsf{St}_{\mathcal{S}}}{t[x\backslash s] \in \mathsf{NF}^{\bullet}_{\mathcal{A},\mathcal{S},\mu}}\text{NF-ESS}^{\bullet}$$

Proof. See [10, Corollary B.11].

UCBV enjoys a strong form of confluence, namely the diamond property.

Theorem 2 (Diamond Property). *Let* $t \xrightarrow{\bullet}_{\rho_1,\emptyset,\mathcal{S},\not\emptyset} t_1$ *and* $t \xrightarrow{\bullet}_{\rho_2,\emptyset,\mathcal{S},\not\emptyset} t_2$*, where* $t_1 \neq t_2$*, and* $\rho_1, \rho_2 \in \{\mathsf{db}, \mathsf{lsv}\}$*, and* $\mathcal{S} = \mathsf{fv}(t)$*. Then, there exists a term* t' *such that* $t_1 \xrightarrow{\bullet}_{\rho_2,\emptyset,\mathcal{S},\not\emptyset} t'$ *and* $t_2 \xrightarrow{\bullet}_{\rho_1,\emptyset,\mathcal{S},\not\emptyset} t'$*.*

Proof. The proof follows [10, Proposition B.3 and Proposition B.10].

This result has two consequences:

Corollary 1. *Any two reduction sequences to normal form in* UCBV *have the same number of* db *and* lsv*-steps.*

As with LCBV, *confluence* of UCBV holds only for the *top-level* step kinds db and lsv. Let $\xrightarrow{\bullet}_{\mathsf{top},\mathcal{S}} := \xrightarrow{\bullet}_{\mathsf{db},\emptyset,\mathcal{S},\not\emptyset} \cup \xrightarrow{\bullet}_{\mathsf{lsv},\emptyset,\mathcal{S},\not\emptyset}$ denote the top-level UCBV reduction, then:

Corollary 2. *The reduction* $\xrightarrow{\bullet}_{\mathsf{top},\mathcal{S}}$ *is confluent.*

5 Relating Linear and Useful Call-by-Value

This section formally relates LCBV and UCBV, defined respectively in Sections 3 and 4. We show that *usefulness* in UCBV is a (complete) restriction of LCBV: UCBV evaluates to *equivalent* normal forms while omitting certain substitution steps, specifically those that do not contribute to the creation of db-redexes.

Simulating LCBV with UCBV is straightforward, as each UCBV step is also a LCBV step, as we mention in Section 4. However, relating both calculi in the opposite direction is much more delicate, since not all LCBV steps are useful: UCBV disallows some substitution steps that LCBV allows. For example, the step $t = (x\,y)[y\backslash \mathtt{I}] \xrightarrow{\circ} (x\,\mathtt{I})[y\backslash \mathtt{I}] = s$ is not useful. Indeed, t is a normal form in UCBV, whereas in LCBV it is a redex whose normal form is s.

Although normal forms differ between the two formalisms, they are *structurally equivalent*, as LCBV evaluates terms further than UCBV. To relate them precisely, we restrict the standard notion of unfolding to a subtler one. Indeed, recall that in calculi with ESs the (full) unfolding of a term is an operation that performs *all* substitutions. In this section, however, the notion of unfolding must be defined in a controlled way. For example, (fully) unfolding the term $x[x\backslash \mathtt{I}]$, which is a normal form in UCBV, yields the corresponding normal form $\mathtt{I}[y\backslash \mathtt{I}]$ in LCBV. Yet the new notion of the unfolding operation will not need to unfold *all* ESs. For example, in LCBV, $(\lambda x.\, y)[y\backslash \mathtt{I}]$ is not unfolded to $(\lambda x.\, \mathtt{I})[y\backslash \mathtt{I}]$ (due to weak evaluation), and $x[x\backslash y\,y]$ is not unfolded to $(y\,y)[x\backslash y\,y]$ (since variables are never substituted by structures). Intuitively, the new notion of the unfolding operation selectively performs only those substitutions on *reachable* variables bound to values, which is why this new operation is called *partial unfolding*.

The remainder of this section is organised as follows. We first define the partial unfolding of a term t. We then relate LCBV and UCBV via two technical results (Proposition 3), showing that (i) the partial unfolding of normal forms in UCBV always yields a normal form in LCBV, and (ii) redexes in UCBV remain reducible in LCBV.

Partial Unfolding. By Proposition 1 we know that $\xrightarrow{\circ}_{\mathsf{lsv}}$ is confluent. Moreover, $\xrightarrow{\circ}_{\mathsf{lsv}}$ can be shown to be terminating [10, Corollary C.15]. As a consequence, any term t has a unique $\xrightarrow{\circ}_{\mathsf{lsv}}$-normal form.

Relying on the previous observation, we define the **partial unfolding** of a term t, written $t^{\downarrow}$, as the unique $\xrightarrow{\circ}_{\mathsf{lsv}}$-normal form of t. For instance, one has $x[x\backslash y\,y][y\backslash \mathtt{I}]^{\downarrow} = (\mathtt{I}\,\mathtt{I})[x\backslash \mathtt{I}\,\mathtt{I}][y\backslash \mathtt{I}]$. Note that all the ESs in the original term remain on the resulting term and that some occurrences of variables are not substituted, because the occurrence may be below an abstraction, as in $((\lambda y.\, x)\,x)[x\backslash \mathtt{I}]^{\downarrow} = ((\lambda y.\, x)\,\mathtt{I})[x\backslash \mathtt{I}]$, or because the variable may be bound to a structure, as in $(x\,x)[x\backslash z\,z]^{\downarrow} = (x\,x)[x\backslash z\,z]$.

Relating Reduction Steps and Normal Forms. We now formally state the existing relation between LCBV and UCBV. Our goal is to show that UCBV simulates LCBV while reaching equivalent normal forms, up to partial unfolding. To do this, we state a proposition, consisting of two parts: (1) the partial unfolding

of a useful normal form is a linear normal form, and (2) the partial unfolding of a reducible term in UCBV is either a db-redex in LCBV or an abstraction in an applied position. This second condition justifies the name "useful" in UCBV because it implies that any substitution step in the strategy must contribute to the creation of a db-redex (see Section 1).

Recall that $\xrightarrow{\circ}_{\mathsf{top}}$ and $\xrightarrow{\bullet}_{\mathsf{top},\mathcal{S}}$ denote top-level LCBV and UCBV reduction. The following proposition links LCBV and UCBV reductions at the top-level case:

Proposition 3 (Soundness and Completeness). *Let t be a term and $\mathcal{S} = \mathsf{fv}(t)$. Then:*

1. *If t is $\xrightarrow{\bullet}_{\mathsf{top},\mathcal{S}}$-irreducible, then $t^{\downarrow}$ is $\xrightarrow{\circ}_{\mathsf{top}}$-irreducible.*
2. *If $t \xrightarrow{\bullet}_{\mathsf{top},\mathcal{S}} t'$, then there exists t'' such that $t^{\downarrow} \xrightarrow{\circ}_{\mathsf{db}} t''$.*

Proof. The proof follows from [10, Proposition C.25], as it is a particular case of such proposition.

To illustrate statement 1, $(x\,y)[y\backslash \mathtt{I}]$ is $\xrightarrow{\bullet}_{\mathsf{top},\{x\}}$-irreducible, and its partial unfolding is $(x\,\mathtt{I})[y\backslash \mathtt{I}]$, which is $\xrightarrow{\circ}_{\mathsf{top}}$-irreducible. As an example of statement 2, consider the lsv-step $t = (x\,y)[x\backslash \mathtt{I}] \xrightarrow{\bullet}_{\mathsf{top},\{y\}} (\mathtt{I}\,y)[x\backslash \mathtt{I}] = t' = t^{\downarrow}$; note that $t' \xrightarrow{\circ}_{\mathsf{db}} x_1[x_1\backslash x][y\backslash \mathtt{I}]$. From these results, we conclude:

Corollary 3. *A term t is $\xrightarrow{\bullet}_{\mathsf{top},\mathsf{fv}(t)}$-irreducible if and only if $t^{\downarrow}$ is $\xrightarrow{\circ}_{\mathsf{top}}$-irreducible.*

6 Useful Call-by-Value is Invariant

In this section, we relate our UCBV strategy to an existing definition of usefulness in the literature, namely, to the GLAMoUr abstract machine [2]. As a consequence, we obtain that UCBV is *time-invariant*, *i.e.*, that the number of reduction steps to normal form can be used as a measure of time complexity.

The main result of this section is Theorem 3, stating that reduction in UCBV can be implemented by the GLAMoUr abstract machine with *linear* overhead in time. The key property to prove this implementation result is Proposition 4, which embeds the GLAMoUr abstract machine within UCBV.

Informal Discussion. The definitions and proofs to embed the GLAMoUr abstract machine in UCBV follow well-known methodologies (*e.g.* [1,2,4]). In particular, we first define a *decoding* function that maps each state s of the GLAMoUr machine to a term $\{\!\{s\}\!\}$ with ESs. As a second step, one would like to show that each transition step $s \rightsquigarrow s'$ in the GLAMoUr can be simulated by the UCBV strategy in at most one step, *i.e.* $\{\!\{s\}\!\} \xrightarrow{\bullet}{}^{=} \{\!\{s'\}\!\}$.[7] It is well-known that this does not suffice in general, because abstract machines usually keep the *code* (a term without ESs) an the *environment* (a list of ESs) as independent components, while the UCBV strategy allows ESs to be scattered throughout the term. This

[7] We omit the parameters for $\xrightarrow{\bullet}$ in this informal discussion.

means that one can only prove a slightly weaker simulation result, namely that $\{\!\{s\}\!\} \xrightarrow{\bullet}{}^{=} t$ for some term t that differs from $\{\!\{s'\}\!\}$ only in the positions of some of the ESs.

To bridge this gap, we define a notion of *structural equivalence* between terms $t \equiv s$ that allows ESs to float around. The simulation result then shows that each transition $s \rightsquigarrow s'$ can be simulated by the UCBV strategy in at most one step *up to structural equivalence*, *i.e.* that $\{\!\{s\}\!\} \xrightarrow{\bullet}{}^{=} \equiv \{\!\{s'\}\!\}$.

Finally, to simulate the GLAMoUr with UCBV, one would also need to show that structural equivalence is a strong bisimulation with respect to the UCBV strategy, *i.e.* that if $t \equiv s$ then every UCBV step $t \xrightarrow{\bullet} t'$ can be simulated by a corresponding step $s \xrightarrow{\bullet} s'$ such that $t' \equiv s'$. Unfortunately, this result fails. Indeed, structural equivalence should allow commuting an ES with an application, *i.e.* $t[x\backslash u]\, s \equiv (t\, s)[x\backslash u]$ provided that $x \notin \mathsf{fv}(s)$. But a reduction step on the right-hand side: $t[x\backslash u]\, s \xrightarrow{\bullet} t[x\backslash u]\, s'$ cannot always be simulated by a reduction step on the left-hand side: $(t\, s)[x\backslash u] \xrightarrow{\bullet} (t\, s')[x\backslash u]$. The problem is that the UCBV strategy only allows to reduce inside the body of a closure ($t\, s$ in the example) when the argument u is *rigid*, *i.e.* a structure or a hereditary abstraction.

To resolve this, we technically need to impose an invariant on terms, requiring that the arguments of ESs always be rigid. This invariant is obviously satisfied by any initial term without ESs. Moreover, having this invariant ensures that in the equivalence $t[x\backslash u]\, s \equiv (t\, s)[x\backslash u]$ the argument u is rigid, and thus a reduction step internal to s on the right-hand side can be simulated by a reduction step on the left-hand side. Furthermore, to ensure that reduction preserves this invariant, we need to restrict to a subrelation of the UCBV strategy, $\xrightarrow{\blacktriangle}_{\rho,\mathcal{A},\mathcal{S},\mu} \subseteq \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu}$, that we dub *stable reduction*. With these ingredients, we can show that each transition in the GLAMoUr machine can be simulated by at most one *stable* reduction step in UCBV, up to structural equivalence.

Stable Terms and Stable Reduction. Given an abstraction frame $\mathcal{A}$ and a structure frame $\mathcal{S}$, a term t is said to be **rigid** under $(\mathcal{A}, \mathcal{S})$ if $t \in \mathsf{HA}_{\mathcal{A}}$ or $t \in \mathsf{St}_{\mathcal{S}}$. In UCBV, evaluating a closure $t[x\backslash s]$ with abstraction frame $\mathcal{A}$ and structure frame $\mathcal{S}$ proceeds in t only when the argument s is rigid under $(\mathcal{A}, \mathcal{S})$. To better align with low-level abstract machines, we define the set of **stable terms** under $(\mathcal{A}, \mathcal{S})$ as those in which every ES argument is rigid. We also introduce **stable reduction** $\xrightarrow{\blacktriangle}_{\rho,\mathcal{A},\mathcal{S},\mu}$, such that $\xrightarrow{\blacktriangle}_{\rho,\mathcal{A},\mathcal{S},\mu} \subseteq \xrightarrow{\bullet}_{\rho,\mathcal{A},\mathcal{S},\mu}$. Sometimes we write $\xrightarrow{\blacktriangle}$ if the parameters are clear from the context. Stable reduction forces arguments of applications to be rigid before reduction proceeds. Thus in this setting, rule APPL$^\bullet$ permits evaluating the head of an application $t\, s$ only if s is rigid, and rule DB$^\bullet$ allows to contract a db-redex only if its argument is rigid. Stable reduction preserves stable terms and enjoys the diamond property.

The GLAMoUr Abstract Machine. We start by briefly recalling the syntax of the GLAMoUr. Full details on the GLAMoUr transitions and the remaining definitions can be found in [2,4,10].

The syntax of the GLAMoUr is given by the following grammar, defining **states** $(s, s', \ldots)$, **dumps** $(D, D', \ldots)$, **stacks** $(\pi, \pi', \ldots)$, **stack items** $(\phi, \psi, \ldots)$,

and **environments** $(E, E', \ldots)$:

$$\begin{array}{ll} s ::= (D, \mathbf{t}, \pi, E) & D ::= \epsilon \mid D : (\mathbf{t}, \pi) \\ \pi ::= \epsilon \mid \phi^l : \pi & \phi ::= \mathbf{t} \mid (\mathbf{t}, \pi) \\ E ::= \epsilon \mid [x\backslash\phi^l] : E & \end{array}$$

where $l \in \{\mathbb{A}, \mathbb{S}\}$ are **labels** decorating stack items. Here, **codes** $(\mathbf{t}, \mathbf{s}, \ldots)$ are terms with no ESs, but they are **not** considered up to α-equivalence. By convention, labels indicate the shape of ϕ. In particular, $l = \mathbb{A}$ if and only if ϕ is of the form $\mathbf{t}$, and $l = \mathbb{S}$ if and only if ϕ is of the form $(\mathbf{t}, \pi)$. Intuitively, these labels indicate whether a stack item unfolds to a hereditary abstraction ($\mathbb{A}$) or to a structure ($\mathbb{S}$). In a state $s = (D, \mathbf{t}, \pi, E)$, the code $\mathbf{t}$ is called the **focus** of s. A state s_0 is **initial** if it is of the form $s_0 = (\epsilon, \mathbf{t}, \epsilon, \epsilon)$.

There are seven kinds of transitions in the GLAMoUr: transitions $(s \rightsquigarrow_{\mathtt{um}} s')$ for function applications, transitions $(s \rightsquigarrow_{\mathtt{ue}} s')$ for linear substitutions, and *administrative* transitions $s \rightsquigarrow_{\mathtt{c}_i} s'$ with $i \in \{1..5\}$, for changing the evaluation focus without computing. Formally, the transitions of the GLAMoUr abstract machine are:

$$\begin{array}{|c|c|c|c|c|c|c|c|c|} \hline D & \mathbf{t}\,\mathbf{u} & \pi & E & \rightsquigarrow_{\mathtt{c}_1} & D : (\mathbf{t}, \pi) & \mathbf{u} & \epsilon & E \\ D & \lambda\mathbf{x}.\,\mathbf{t} & \phi^l : \pi & E & \rightsquigarrow_{\mathtt{um}} & D & \mathbf{t} & \pi & [x\backslash\phi^l] : E \\ D : (\mathbf{t}, \pi) & \lambda\mathbf{x}.\,\mathbf{u} & \epsilon & E & \rightsquigarrow_{\mathtt{c}_2} & D & \mathbf{t} & (\lambda\mathbf{x}.\,\mathbf{u})^{\mathbb{A}} : \pi & E \\ D : (\mathbf{t}, \pi) & \mathbf{x} & \pi' & E & \rightsquigarrow_{\mathtt{c}_3} & D & \mathbf{t} & (\mathbf{x}, \pi')^{\mathbb{S}} : \pi & E \\ D : (\mathbf{t}, \pi) & \mathbf{x} & \pi' & E_1 : [x\backslash\phi^{\mathbb{S}}] : E_2 & \rightsquigarrow_{\mathtt{c}_4} & D & \mathbf{t} & (\mathbf{x}, \pi')^{\mathbb{S}} : \pi & E_1 : [x\backslash\phi^{\mathbb{S}}] : E_2 \\ D : (\mathbf{t}, \pi) & \mathbf{x} & \epsilon & E_1 : [x\backslash\mathbf{u}^{\mathbb{A}}] : E_2 & \rightsquigarrow_{\mathtt{c}_5} & D & \mathbf{t} & \mathbf{x}^{\mathbb{A}} : \pi & E_1 : [x\backslash\mathbf{u}^{\mathbb{A}}] : E_2 \\ D & \mathbf{x} & \phi^l : \pi & E_1 : [x\backslash\mathbf{u}^{\mathbb{A}}] : E_2 & \rightsquigarrow_{\mathtt{ue}} & D & \mathbf{u}^{\alpha} & \phi^l : \pi & E_1 : [x\backslash\mathbf{u}^{\mathbb{A}}] : E_2 \\ \hline \end{array}$$

where $\mathbf{s}^{\alpha}$ is any code α-equivalent to $\mathbf{s}$ that preserves the well-naming of the machine. Moreover, in the transition $\rightsquigarrow_{\mathtt{c}_3}$, the variable $\mathbf{x}$ does not belong to the domain of environment E. The domain of an environment E is written $\mathtt{dom}(E)$ and is defined recursively by $\mathtt{dom}(\epsilon) := \emptyset$ and $\mathtt{dom}([x\backslash\phi^l] : E') := \{x\} \cup \mathtt{dom}(E')$.

Embedding the GLAMoUr into Useful Call-by-Value. Each GLAMoUr state s can be **decoded** into a term $\{\!\{s\}\!\}$ with ESs—note that this is exactly the decoding used in [10, Subsection D.1.1]:

- Decoding of states: $\{\!\{(D, \mathbf{t}, \pi, E)\}\!\} := \{\!\{E\}\!\langle\{\!\{D\}\!\langle\{\!\{\pi\}\!\langle t\rangle\rangle\rangle$.
- Decoding of dumps: $\{\!\{\epsilon\}\!\} := \diamond$ and $\{\!\{D : (\mathbf{t}, \pi)\}\!\} := \{\!\{D\}\!\langle\{\!\{\pi\}\!\langle t\diamond\rangle\rangle$.
- Decoding of stacks: $\{\!\{\epsilon\}\!\} := \diamond$ and $\{\!\{\phi^l : \pi\}\!\} := \{\!\{\pi\}\!\langle\diamond\{\!\{\phi^l\}\!\}\rangle$.
- Decoding of codes: $\{\!\{\mathbf{t}\}\!\} := t$, where t is the α-equivalence class of $\mathbf{t}$.
- Decoding of stack items: $\{\!\{(\mathbf{t}, \pi)^l\}\!\} := \{\!\{\pi\}\!\langle\{\!\{\mathbf{t}\}\!\}\rangle$ and $\{\!\{\mathbf{t}^l\}\!\} := \{\!\{\mathbf{t}\}\!\}$.
- Decoding of environments: $\{\!\{\epsilon\}\!\} := \diamond$ and $\{\!\{[x\backslash\phi^l] : E\}\!\} := \{\!\{E\}\!\langle\diamond[x\backslash\{\!\{\phi^l\}\!\}]\rangle$.

As discussed before, the GLAMoUr abstract machine can be simulated in UCBV up to **structural equivalence** of terms [3], denoted $\equiv$, defined as the least equivalence relation closed by weak contexts (*i.e.*, contexts that do not enter inside abstractions) of the four following axioms:

$$\begin{array}{cc} t[x\backslash s][y\backslash u] \equiv t[y\backslash u][x\backslash s] & t[x\backslash s][y\backslash u] \equiv t[x\backslash s[y\backslash u]] \\ (t\,s)[x\backslash u] \equiv t[x\backslash u]\,s & (t\,s)[x\backslash u] \equiv t\,s[x\backslash u] \end{array}$$

All these axioms assume that variable-capture is avoided, for example, the first equation assumes that $x \notin \mathsf{fv}(u)$ and $y \notin \mathsf{fv}(s)$.

The main technical result of this section is the following:

Proposition 4 (GLAMoUr Simulation). *Let s be a state reachable from an initial state with focus t_0 and let $\mathcal{S}_0 := \mathsf{fv}(t_0)$. Then:*

1. *If $s \rightsquigarrow_{\mathtt{um}} s'$, then $\{\!\{s\}\!\} \xrightarrow{\blacktriangle}_{\mathsf{db}} \equiv \{\!\{s'\}\!\}$.*
2. *If $s \rightsquigarrow_{\mathtt{ue}} s'$, then $\{\!\{s\}\!\} \xrightarrow{\blacktriangle}_{\mathsf{lsv}} \equiv \{\!\{s'\}\!\}$.*
3. *If $s \rightsquigarrow_{\mathtt{c}_i} s'$, then $\{\!\{s\}\!\} = \{\!\{s'\}\!\}$, for all $i \in \{1..5\}$.*
4. *If s is $\rightsquigarrow$-irreducible, then $\{\!\{s\}\!\}$ is $\xrightarrow{\blacktriangle}$-irreducible (progress property).*

Proof. See [10, Lemma 6.1].

As a consequence, any *sequence* of GLAMoUr transitions corresponds to a *sequence* of (stable) UCBV reduction steps interleaved with equivalences. For this correspondence, we need Lemma 1, stating that $\equiv$ is a strong bisimulation with respect to $\xrightarrow{\blacktriangle}$, for ensuring a postponement property.

Lemma 1 (Strong Bisimulation). *Let t_0 and s_0 be stable terms such that $s_0 \equiv t_0$. If $t_0 \xrightarrow{\blacktriangle}_{\rho,\mathcal{A},\mathcal{S},\mu} t_1$, then there exists s_1 such that $s_0 \xrightarrow{\blacktriangle}_{\rho,\mathcal{A},\mathcal{S},\mu} s_1 \equiv t_1$.*

Proof. See [10, Lemma 6.2].

This result relies crucially on the stability notion introduced earlier in this section: if u is not stable, equivalences like $t[x\backslash s][y\backslash u] \equiv t[y\backslash u][x\backslash s]$ cannot be postponed after the (non-stable) step $t[y\backslash u][x\backslash s] \xrightarrow{\bullet} t[y\backslash u][x\backslash s']$ because rules like ESLA$^\bullet$ and ESLS$^\bullet$ cannot apply when u is not rigid.

Based on the previous results, we can now present the main result of this section: UCBV can be implemented in the GLAMoUr with bilinear overhead in time, as a function of the size of the starting term and the length of the reduction sequence to normal form.

Theorem 3 (Bilinear Simulation of UCBV). *Let t be a term with no ESs. If $t \xrightarrow{\blacktriangle}{}^n t'$, where t' is in normal form and s is an initial state such that $\{\!\{s\}\!\} = t$, then $s \rightsquigarrow^k s'$, where $\{\!\{s'\}\!\} \equiv t'$ and $k \in O(|t| \cdot (n+1))$.*

Proof. See [10, Theorem 6.3].

In turn, this entails that UCBV is time-invariant, *i.e.*, that it can be simulated with at most polynomial overhead by a time-invariant cost model (such as Turing machines or RAMs):

Corollary 4. *The UCBV strategy is time-invariant.*

7 Conclusions

This paper proposes an inductive specification of usefulness for open CBV evaluation called UCBV, that contrasts with previous notions of usefulness in CBN, CBV, and call-by-need that are non-inductive. Our specification is inspired by the definition of strong call-by-need evaluation in [8]. Although these strategies are very different, one point in common is that they crucially depend on the information of the surrounding evaluation context, reflected in the form of parameters.

We show that UCBV is sound and complete with respect to LCBV, in the sense that, given a starting term, LCBV and UCBV compute the same normal form, up to unfolding. We also show that the LCBV calculus which implements linear substitution is observationally equivalent to the VSC. Our last contribution is establishing a relation between UCBV and an existing definition of usefulness in the literature, the GLAMoUr abstract machine [2].

Benefits of an Inductive Approach. The inductive formulation of useful evaluation is the key that has enabled us to define a *semantic model* for useful open CBV. In particular, our formulation of usefulness turned out to be appropriate to be captured by a *non-idempotent intersection type system* [11,12]. Semantic models are *compositional*, in the sense that the meaning of a program is defined inductively, in terms of the meanings of its subprograms. The inductive approach that we propose in this paper allowed us to reason about its semantics in a compositional way.[8]

Another benefit of an inductive definition of useful CBV is that all the invariants of key properties become explicit, making such notion closer to being formalised along with its results in an interactive proof assistant.

Finally, we think that the techniques and tools developed in this paper could be adapted to other evaluation strategies, and in particular to other context-sensitive notions of evaluation, such as useful CBN, whose formulation is not inductive [6]. A challenging question is how to extend our specification of usefulness to *strong* CBV so that evaluation is also allowed under abstractions. This is relevant for the implementation of proof assistants based on dependent type theory, in which type checking requires to decide the definitional equality of type expressions up to full β-conversion, thus requiring strong evaluation. Even more challenging would be to adapt all this technology to call-by-need, and to call-by-push-value.

Acknowledgments. *This project has received funding from the EU Horizon 2020 research and innovation programme under the MSCA grant No 945332.*

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

[8] The semantic study is outside the scope of this paper, but it can be found in the associated technical report [10].

References

1. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: Jeuring, J., Chakravarty, M.M.T. (eds.) Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014. pp. 363–376. ACM (2014)
2. Accattoli, B., Coen, C.S.: On the relative usefulness of fireballs. In: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015. pp. 141–155. IEEE Computer Society (2015). https://doi.org/10.1109/LICS.2015.23, https://doi.org/10.1109/LICS.2015.23
3. Accattoli, B., Guerrieri, G.: Open call-by-value. In: Igarashi, A. (ed.) Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10017, pp. 206–226 (2016). https://doi.org/10.1007/978-3-319-47958-3_12, https://doi.org/10.1007/978-3-319-47958-3_12
4. Accattoli, B., Guerrieri, G.: Implementing open call-by-value. In: Dastani, M., Sirjani, M. (eds.) Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10522, pp. 1–19. Springer (2017). https://doi.org/10.1007/978-3-319-68972-2_1, https://doi.org/10.1007/978-3-319-68972-2_1
5. Accattoli, B., Guerrieri, G.: Abstract machines for open call-by-value. Sci. Comput. Program. **184** (2019). https://doi.org/10.1016/J.SCICO.2019.03.002, https://doi.org/10.1016/j.scico.2019.03.002
6. Accattoli, B., Lago, U.D.: Beta reduction is invariant, indeed. In: Henzinger, T.A., Miller, D. (eds.) Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014. pp. 8:1–8:10. ACM (2014). https://doi.org/10.1145/2603088.2603105, https://doi.org/10.1145/2603088.2603105
7. Accattoli, B., Paolini, L.: Call-by-value solvability, revisited. In: Schrijvers, T., Thiemann, P. (eds.) Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7294, pp. 4–16. Springer (2012)
8. Balabonski, T., Lanco, A., Melquiond, G.: A strong call-by-need calculus. Logical Methods in Computer Science **19**(1) (2023)
9. Barenbaum, P., Kesner, D., Milicich, M.: Semantic equivalence between the linear call-by-value calculus and the value substitution calculus (technical note) (2025), https://www.irif.fr/_media/users/milicich/technicalnotelcbv.pdf
10. Barenbaum, P., Kesner, D., Milicich, M.: Useful evaluation: Syntax and semantics (technical report) (2025), https://arxiv.org/abs/2404.18874
11. Carvalho, D.d.: Sémantiques de la logique linéaire et temps de calcul. Ph.D. thesis, Ecole Doctorale Physique et Sciences de la Matière (Marseille) (2007)
12. Gardner, P.: Discovering needed reductions using type theory. In: Theoretical Aspects of Computer Software. pp. 555–574. Springer (1994)
13. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: Wand, M., Jones, S.L.P. (eds.) Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002. pp. 235–246. ACM (2002). https://doi.org/10.1145/581478.581501, https://doi.org/10.1145/581478.581501

14. Kesner, D.: A theory of explicit substitutions with safe and full composition. Logical Methods in Computer Science **5**(3) (2009), http://arxiv.org/abs/0905.2539
15. Slot, C.F., van Emde Boas, P.: On tape versus core; an application of space efficient perfect hash functions to the invariance of space. In: DeMillo, R.A. (ed.) Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA. pp. 391–400. ACM (1984). https://doi.org/10.1145/800057.808705, https://doi.org/10.1145/800057.808705
16. Terese: Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
17. Yoshida, N.: Optimal reduction in weak-λ-calculus with shared environments. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture. p. 243–252. FPCA '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/165180.165217, https://doi.org/10.1145/165180.165217