The Zoo of Lambda-Calculus Reduction Strategies, And Coq

Małgorzata Biernacka
University of Wrocław, Faculty of Mathematics and Computer Science, Poland

Witold Charatonik
University of Wrocław, Faculty of Mathematics and Computer Science, Poland

Tomasz Drab
University of Wrocław, Faculty of Mathematics and Computer Science, Poland

Abstract

We present a generic framework for the specification and reasoning about reduction strategies in the lambda calculus, representable as sets of term decompositions. It is provided as a Coq formalization that features a novel format of *phased* strategies. It facilitates concise description and algebraic reasoning about properties of reduction strategies. The formalization accommodates many well-known strategies, both weak and strong, such as call by name, call by value, head reduction, normal order, full β-reduction, etc. We illustrate the use of the framework as a tool to inspect and categorize the "zoo" of existing strategies, as well as to discover and study new ones with particular properties.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Operational semantics; Theory of computation → Automated reasoning

Keywords and phrases Lambda calculus, Reduction strategies, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.7

Supplementary Material *Software (Source Code)*: `https://bitbucket.org/pl-uwr/strategies` archived at `swh:1:dir:a1628e852357d77fe8d678b7adbbb536baa331ab`

Funding This research is supported by the National Science Centre of Poland, under grant number 2019/33/B/ST6/00289.

Acknowledgements We thank the anonymous reviewers for their comments and references to the literature. The last author thanks his wife, Monika, for taking care of everything around during proof completion.

1 Introduction

The behavior of the lambda calculus, be it as a computation model or a prototype programming language, is based on the notion of β-reduction that constitutes a basic computation step. In its general definition β-reduction is non-deterministic and unrestricted – a term can be reduced in different ways, and the consequences of the different reduction choices can vary tremendously, as for the term $K\,x\,\Omega$: it can normalize in two steps in call by name, reduce indefinitely in call by value [27], or it can be stuck in a variant of call by value with abstractions as the only values. In many applications it is useful or even necessary to trim down the full generality of β-reduction by following a specific (not necessarily deterministic) *reduction strategy*. In general terms, as Barendregt put it, "a reduction strategy provides a way of choosing how to reduce a term" [5].

When considering lambda calculus as a programming language or a computation model, Barendregt's specification is too general, and we typically impose quite strong restrictions for strategies to be considered useful or efficient. Computation with the lambda calculus consists in reducing terms until a normal form (i.e., an irreducible term) is found. Therefore, one can adopt the following properties as practically relevant characteristics of reduction strategies: termination (does the strategy always terminate for a term that has a normal form?), the number of steps it performs to reach a normal form, or its effectiveness (i.e., how easy it is to determine at each step what the reduced term should be).

The concept of effectiveness of the strategy poses the following problem to consider: how do we define reduction strategies, and in particular how do we define effective strategies? Fortunately, this question has many useful answers coming from the programming languages community, where various formats of operational semantics have been developed. The popular approaches include: big-step semantics that describes the relation between the term and its final value [18], and small-step semantics that describes single steps of computation, and is typically given in the format of structural operational semantics [28], or of reduction semantics with explicit representation of reduction contexts [12]. Effectiveness of these formats (and in consequence, of the strategies defined by them) relies on the fact that their semantic rules are typically defined inductively, and in such a way that the cost of applying a rule can be bounded by some low factor (ideally, constant) related to the inspection of the data structures involved. There exist numerous reduction strategies introduced to model desired properties of the lambda calculus that can be described in a common semantic format, there exist methodologies and tools to implement and test them [10, 11, 34], but how can we compare strategies, and how do we find new, useful ones?

In this work, we make an attempt at characterizing and categorizing effective strategies in the pure lambda calculus, as they pertain to the order of β-reductions in a reduction sequence, and we use Coq as our implementation platform [33]. A source of inspiration for our work was Sestoft's survey of reduction strategies in the lambda calculus [31] that collected existing strategies and their ML implementations, as a didactic tool. However, unlike Sestoft, we base our work on the small-step approach. We exploit the fact that in general small-step semantics provides more fine-grained control over computation, for example it makes explicit the order of evaluation (as in left-to-right vs. right-to-left), whereas big-step semantics may leave it unspecified. Therefore, we choose to take as a foundation the view of strategies as *decompositions* of terms into reduction contexts and redices, as offered by the format of reduction semantics. The framework that we built on these assumptions delineates the space where most of existing strategies live, and new ones can be discovered, systematically. One notable omission is the call-by-need strategy – it seems that this strategy should fit in our picture, but it is harder to characterize, and therefore still remains in the wild.

Related Work. The work closest to ours is Sestoft's survey of lambda-calculus reduction strategies. It is accompanied by implementation, but it does not mechanize the properties of the strategies. Recently, with the rising popularity of proof assistants numerous mechanizations of concrete programming languages have been done, but we are not aware of any comprehensive study of various reduction strategies.

Earlier formalizations of the λ-calculus theory include Shankar's mechanization of the Church-Rosser theorem in Boyer-Moore theorem prover [32] and Pfenning's mechanization in Elf [25], Huet's formalization of the residual theory of λ-calculus in Coq [17], McKinna & Pollack's formalization of Church-Rosser theorem, standardization theorem, and the basic theory of Pure Type Systems in LEGO Proof Development System [22], Norrish's formalization of basic λ-calculus theory in HOL [23]. Pierce et al. provide a series of textbooks on software verification including usage of operational semantics in the form of Coq proof scripts [26]. Biernacka et al. formalize derivations of refocusing abstract machines for various lambda-calculus reduction strategies in Coq [8]. Most recently, Forster et al. formalized in Coq reasonability for time and space of weak call by value (shortened *lcbv* and *cbv* here) [13, 14] and essential theorems for it as a model of computation [15].

Contributions and outline. The contributions of our work are the following:

1. We provide a generic, simple and intuitive formalization of lambda-calculus reduction strategies in Coq. The formalization is discussed in Section 2.
2. We collect and categorize common existing strategies studied in the functional programming-language community. We formalize and prove some of their properties, such as inclusion, determinism, and uniformity. This systematization is presented in Section 3.
3. We propose a novel semantic format to define *phased* reduction strategies, formalized in Coq, that facilitates concise specification and algebraic reasoning about strategies. It is introduced in Section 4, where some of its benefits and advantages over existing formats are also discussed.

2 Basic concepts and Coq formalization

We aimed at a simple, concise and intuitive formalization. It turns out we only need two `Inductive` definitions in the whole development: one for λ-terms, and one for context frames. Our formalization is also minimalistic in the sense that we do not introduce any more concepts than we need. In particular, we do not need substitution to talk about strategies, and so we do not include it in our formalization. This is not a limitation, since the framework is open to extensions and substitution can be added in order to prove dynamic properties of the lambda calculus under any definable strategy. Many notions are expressed in terms of basic set theory as presented next.

2.1 Sets

Sets of elements of type `A`, denoted with `𝒫 A`, are represented as functions of type `A → Prop`. This is similar to `Coq.Sets.Ensembles` but we do not use the axiom of extensionality. The definitions are standard, and excerpts are shown in Listing 1.[1]

Listing 1 Basic relations on sets.

```
Notation "'𝒫' A"    := (A → Prop) (at level 55, only parsing).
Notation "x '∈' A" := (A x)       (at level 70, only parsing).

Definition   subset {A:Type} (s t:𝒫 A) : Prop := ∀ x, x ∈ s → x ∈ t.
Definition   set_eq {A:Type} (s t:𝒫 A) : Prop := ∀ x, x ∈ s ↔ x ∈ t.
Definition disjoint {A:Type} (s t:𝒫 A) := ∀ x, x ∈ s → x ∈ t → False.

Infix     "⊆" := subset            (at level 70).
Infix    "==" := set_eq            (at level 70).
Notation "∅" := empty_set.
Notation "●" := full_set.
Infix     "∪" := union             (at level 65).
Notation "⋃" := family_union.
Infix     "×" := cartesian_product (at level 50).
```

Using the typeclass mechanism of Coq, we show that `set_eq` is an equivalence relation, that `union` is a properly defined operation on sets, and that cartesian product is monotone (see Listing 2).

[1] It is worth noting that `set_eq` is independent of notion of equality on type `A` if it has its own one as for example real numbers.

Listing 2 Example properties of sets.

```
Instance set_eq_equiv {A:Type} : Equivalence (@set_eq A).
Instance union_proper {A:Type} :
  Proper (set_eq ==> set_eq ==> set_eq) (@union A).
Instance cartesian_product_monotone {A B:Type} :
  Proper (subset ++> subset ++> subset) (@cartesian_product A B).
```

2.2 Families of terms

The grammar of lambda terms

$$t ::= x \mid \lambda x.t \mid t_1\, t_2$$

is formalized in a straightforward way, with strings used to represent variable names, as shown in Listing 3.

Listing 3 Inductive definition of terms.

```
Inductive term : Type :=
| var (x : string) : term
| lam (x : string) (s : term) : term
| app (s : term)   (t : term) : term.
```

When discussing reduction strategies, we will stumble upon various normal forms – they occur as term families expressible with simple grammars.

$$\begin{array}{rclrcl}
\mathit{nf} \ni n & ::= & \lambda x.n \mid a & \mathit{neu} \ni a & ::= & x \mid a\, n \\
\mathit{wnf} \ni w & ::= & \lambda x.t \mid i & \mathit{inert} \ni i & ::= & x \mid i\, w \\
\mathit{hnf} \ni h & ::= & \lambda x.h \mid r & \mathit{rigid} \ni r & ::= & x \mid r\, t \\
\mathit{whnf} \ni q & ::= & \lambda x.t \mid r & & & \\
\mathit{abs} \ni \_ & ::= & \lambda x.t & \mathit{abs}^{\complement} \ni \_ & ::= & x \mid t\, t
\end{array}$$

$$\begin{tikzcd}
\mathit{neu} \arrow[r] \arrow[d] & \mathit{inert} \arrow[r] \arrow[dd] & \mathit{rigid} \arrow[r] \arrow[d] & \mathit{abs}^{\complement} \\
\mathit{nf} \arrow[rr] \arrow[dr] & & \mathit{hnf} \arrow[d] & \\
\mathit{abs} \arrow[r] & \mathit{wnf} \arrow[r] & \mathit{whnf} &
\end{tikzcd}$$

Figure 1 Term families and their inclusions.

In Figure 1, the grammars of common normal forms are shown, together with a diagram illustrating the relationships between them (the arrows represent inclusions between families). For example, *nf* is the family of full β-normal forms (resulting from full, unrestricted normalization), and is defined with the use of an auxiliary family of neutral terms *neu*. The family *wnf* of weak normal forms occurs as the final results of computation in the (weak) open call-by-value strategy, and is defined with the use of an auxiliary family of *inert* terms [2]. Weak-head normal forms *whnf* result from call-by-name reduction, and its subfamily of head normal forms *hnf* contains head-reduced terms. The name *rigid* for variable-headed application comes from Accattoli et al. [1].

In the implementation we define the various normal forms as sets of terms using `Fixpoints`, as shown for *nf* and *neu* in Listing 4. Such definitions contain superfluous information, e.g., the grammar of *neu* has to be repeated. Nevertheless, we show in lemmas `nf_grammar` and `neu_grammar` in Listing 4 that the fixpoint definitions coincide with the grammars in Figure 1, where `abs_of` and `app_of` are Coq versions of the constructors used in the grammars. Moreover, using simple set-based reasoning we can prove that neutral terms are exactly rigid normal terms (`neu_rigid_nf`), and we can prove all inclusions shown in Figure 1 (the example of $\mathit{neu} \subseteq \mathit{nf}$ is shown in the listing).

Listing 4 Definition, grammars and properties of normal and neutral terms.

```
Fixpoint neu (a:term) : Prop :=
  match a with
  | var x   => True
  | app a n => neu a ∧ nf n
  | _       => False
  end
with nf (n:term) : Prop :=
  match n with
  | lam x n => nf n
  (* _      => neu n *)
  | var x   => True
  | app a n => neu a ∧ nf n
  end.

Lemma   nf_grammar :  nf == abs_of nf ∪ neu.
Lemma  neu_grammar : neu ==  variable ∪ app_of neu nf.
Lemma neu_rigid_nf : neu == rigid ∩ nf.
Lemma    neu_is_nf : neu ⊆ nf.
```

2.3 Semantic formats

In this section we discuss the most popular approaches to defining computation in the lambda calculus in order to justify the choices made in our formalization. Our running example is the left-to-right open call-by-value strategy (shortened as *lcbw*) studied in detail in [2] but – as is often the case – occurring in the literature previously [24, 31].

Definitional interpreters

A natural way for a programmer to define a reduction is to write a definitional interpreter in some metalanguage [30]. This way *lcbw* is defined in Paulson's textbook [24] and we present his evaluator in Listing 5, translated to Coq syntax. The definition is clear and the deterministic order of reductions follows from the evaluation order of the metalanguage. However, Coq cannot accept this definition because of possible nontermination.

Listing 5 Pseudocode of Paulson's evaluator for left-to-right open call by value.

```
Parameter subst : string → term → term → term.

Fixpoint eval (t : term) :=
  match t with
  | app s t => match eval s with
               | lam x u => eval (subst x (eval t) u)
               |       u =>        app u (eval t)
               end
  |       t => t
  end.
```

A variant of this approach is to define a higher-order evaluator that represents some values as functions of the metalanguage, but this is also inapplicable in our situation because of nontermination.

Big-step operational semantics

The evaluator above can be intuitively translated to big-step operational semantics format. The big-step formulation of the *lcbw* strategy is given in Figure 2. It is isomorphic to the one given by Sestoft [31] under the name of "the call-by-value reduction" *bv*.

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t_1 \Downarrow \lambda x.t \quad t_2 \Downarrow t_2' \quad t[x := t_2'] \Downarrow t'}{t_1\, t_2 \Downarrow t'} \qquad \frac{t_1 \Downarrow t_1' \not\equiv \lambda x.t \quad t_2 \Downarrow t_2'}{t_1\, t_2 \Downarrow t_1'\, t_2'}$$

Figure 2 Sestoft's big-step operational semantics of open call by value.

The inductive definition of the relation $\Downarrow$ can be formulated in Coq. However, due to presentation of inference rules with multiple independent premises, the evaluation order happens to be only conventional. In order to discriminate between, e.g., left-to-right and right-to-left, we need more specific formats.

Another drawback shared by the big-step approaches is the fact that actual computation is welded with navigation in a term. Therefore, we consider these semantic formats as derived from a more fundamental, small-step semantics [29]. Interestingly, the specification of evaluation order can be restored in pretty-big-step semantics [9], which – being big-step in spirit – is convenient for certain types of reasoning.

Small-step structural operational semantics

In Figure 3, we show the specification of *lcbw* by a relation $\overset{lcbw}{\to}$ defined in the format of Plotkin's structural operational semantics [28]. This definition is directly expressible in Coq.

$$\frac{}{(\lambda x.t)\, w \overset{lcbw}{\to} t[x := w]}\ (\beta) \qquad \frac{t_1 \overset{lcbw}{\to} t_1'}{t_1\, t_2 \overset{lcbw}{\to} t_1'\, t_2}\ (\mu) \qquad \frac{t_2 \overset{lcbw}{\to} t_2'}{w_1\, t_2 \overset{lcbw}{\to} w_1\, t_2'}\ (\nu)$$

Figure 3 Structural operational semantics for left-to-right open call by value.

Notice that the small-step formulation demands more information about the reduction strategy. In the rule (β), we have to restrict substitution to accept only weak normal forms w, as defined in Figure 1. Without this condition the defined strategy would be nondeterministic and not call-by-value. Similarly, the rule (ν) states explicitly that the left component of the application has to be a weak normal form. Therefore, in contrast to the big-step approach, the small-step formulation requires a precise description of the grammar of terms allowed in designated positions in a term, for a rule to be applied.

Small-step reduction semantics

We can observe that that rules (μ) and (ν) in structural operational semantics play a different role than (β) because they simply propagate inner steps while (β) is responsible for the actual reduction. In reduction semantics [12] these propagation rules are materialized with an explicit data structure representing contexts, and the one-step reduction is expressed via contextual closure of the basic reduction rule called *contraction*. In Figure 4, we present a reduction semantics for *lcbw*, which coincides with the left-to-right variant of Accattoli and Guerrieri's definition of open call by value [2].

$$\frac{}{(\lambda x.t)\,w \rightharpoonup_{\beta_w} t[x := w]} \qquad \frac{t \rightharpoonup_{\beta_w} t'}{E[t] \overset{lcbw}{\to} E[t']} \qquad E ::= \square \mid w\,E \mid E\,t$$

Figure 4 Reduction semantics for left-to-right open call by value.

The format of reduction semantics is more compact than the one of structural operational semantics. It is also more convenient in defining reduction strategies as it separates contraction (specifying how subterms are rewritten) from contexts (specifying where the contraction takes place). Therefore we choose this format to define strategies and to formalize in Coq. However, in the following, we will use the nonterminal ${}^{w}V^{t}$ instead of E for contexts of *lcbw* due to the shortage of capital letters in the latin alphabet.

2.4 Reduction contexts

Traditionally, contexts used in reduction semantics are defined by grammars. The grammar of general lambda-calculus contexts is the following[2]:

$$C ::= \square \mid \lambda x.C \mid C\,t \mid t\,C$$

Every context can be seen as a term with exactly one free occurrence of a special variable $\square$ called the hole. It is used to indicate a particular location in a term.

Alternatively, a context can be thought of and represented as a list of elementary contexts of the form $\lambda x.\square$, $\square\, t$, $t\,\square$, called *frames*, on the path between the root of the term and the hole. This is the representation that we use (see Listing 6). We often think of this representation as a sequence of navigation steps made from the root of the term in order to find a redex to contract.

Listing 6 Inductive definition of contexts.

```
Inductive frame : Type :=
|  Lam : string → frame
| Rapp : term → frame
| Lapp : term → frame.

Definition context : Type := list frame.
```

Technically, a general context (C above) is a *folded* representation of a list of frames. Correspondingly, reduction semantics introduces a `plug` function (often left implicit) to reconstruct a term from a context and a term put in its hole. It is denoted as $C[t]$, where C is a context and t is a term.

$$\square[s] = s \qquad (C\,t)[s] = C[s]\,t \qquad (t\,C)[s] = t\,C[s] \qquad (\lambda x.C)[s] = \lambda x.C[s]$$

Just as we need to define normal forms as term families to specify strategies, we also need to restrict the general grammar of contexts. We do it by defining specific context families, considered as sets of contexts, and therefore formalized as functions of type `context → Prop`.

[2] McBride showed that it can be derived through simple symbolic differentiation [21].

The common examples of reduction contexts are collected as a cheat sheet in Figure 5, where we use their familiar specification as terms with a hole. For example, CBN defines standard call-by-name contexts, that in the frames representation consist of just one type of frames: $\square\, t$. For another example, Weak defines a family of contexts representing a non-deterministic weak strategy that does not reduce under lambda abstractions.

$$
\begin{array}{rl@{\qquad}rl}
\text{CBN} \ni & Q ::= \square \mid Q\, t & & \\
\text{Head} \ni & H ::= Q \mid \lambda x.H & \text{Rigid} \ni & \overline{C} ::= \square \mid \overline{C}\, t \mid r\, C \\
\text{HS} \ni & J ::= \square \mid J\, t \mid \lambda x.J & & \\
\text{LO} \ni & N ::= \overline{N} \mid \lambda x.N & \overline{\text{LO}} \ni & \overline{N} ::= \square \mid \overline{N}\, t \mid a\, N \\
\text{LS} \ni & M ::= J \mid \overline{M} \mid \lambda x.M & \overline{\text{LS}} \ni & \overline{M} ::= \quad \overline{M}\, t \mid a\, M \\
\text{LI} \ni & L ::= \square \mid L\, t \mid n\, L \mid \lambda x.L & \text{RI} \ni & R ::= \square \mid R\, n \mid t\, R \mid \lambda x.R \\
\text{Weak} \ni & W ::= \square \mid t\, W \mid W\, t & & \\
\text{LCBV} \ni & {}^{\lambda}V^{t} ::= \square \mid (\lambda x.t)\, {}^{\lambda}V^{t} \mid {}^{\lambda}V^{t}\, t & \text{RCBV} \ni & {}^{t}V^{\lambda} ::= \square \mid t\, {}^{t}V^{\lambda} \mid {}^{t}V^{\lambda}\, (\lambda x.t) \\
\text{LCBW} \ni & {}^{w}V^{t} ::= \square \mid w\, {}^{w}V^{t} \mid {}^{w}V^{t}\, t & \text{RCBW} \ni & {}^{t}V^{w} ::= \square \mid t\, {}^{t}V^{w} \mid {}^{t}V^{w}\, w \\
\text{SDET} \ni & V^{\lambda} ::= \square \mid V^{\lambda}\, (\lambda x.t) & & \\
\text{LOW} \ni & {}^{i}V^{t} ::= \square \mid i\, {}^{i}V^{t} \mid {}^{i}V^{t}\, t & & \\
\text{LLCBW} \ni & {}^{w}_{a}V^{t}_{w} ::= {}^{w}V^{t} \mid \overline{{}^{w}_{a}V^{t}_{w}} \mid \lambda x.{}^{w}_{a}V^{t}_{w} & & \overline{{}^{w}_{a}V^{t}_{w}} ::= a\, {}^{w}_{a}V^{t}_{w} \mid \overline{{}^{w}_{a}V^{t}_{w}}\, w \\
\text{RLCBW} \ni & {}^{w}_{i}V^{t}_{n} ::= {}^{w}V^{t} \mid \overline{{}^{w}_{i}V^{t}_{n}} \mid \lambda x.{}^{w}_{i}V^{t}_{n} & & \overline{{}^{w}_{i}V^{t}_{n}} ::= i\, {}^{w}_{i}V^{t}_{n} \mid \overline{{}^{w}_{i}V^{t}_{n}}\, n \\
\text{LRCBW} \ni & {}^{t}_{a}V^{w}_{w} ::= {}^{t}V^{w} \mid \overline{{}^{t}_{a}V^{w}_{w}} \mid \lambda x.{}^{t}_{a}V^{w}_{w} & & \overline{{}^{t}_{a}V^{w}_{w}} ::= a\, {}^{t}_{a}V^{w}_{w} \mid \overline{{}^{t}_{a}V^{w}_{w}}\, w \\
\text{RRCBW} \ni & {}^{t}_{i}V^{w}_{n} ::= {}^{t}V^{w} \mid \overline{{}^{t}_{i}V^{w}_{n}} \mid \lambda x.{}^{t}_{i}V^{w}_{n} & & \overline{{}^{t}_{i}V^{w}_{n}} ::= i\, {}^{t}_{i}V^{w}_{n} \mid \overline{{}^{t}_{i}V^{w}_{n}}\, n \\
\text{SCBW} \ni & {}^{t}_{i}V^{t}_{w} ::= W \mid \overline{{}^{t}_{i}V^{t}_{w}} \mid \lambda x.{}^{t}_{i}V^{t}_{w} & & \overline{{}^{t}_{i}V^{t}_{w}} ::= i\, {}^{t}_{i}V^{t}_{w} \mid \overline{{}^{t}_{i}V^{t}_{w}}\, w
\end{array}
$$

Figure 5 Context families cheat sheet.

Even though we use the frame representation, we can still prove that the two representations are equivalent, i.e., that they define the same context family. For example, our formalization of *lcbw*-contexts is shown in Listing 7 as the `L_CBW` function, where `Uniform` is a function checking that all frames in a context are proper *lcbw* frames. On the other hand, we can express the LCBW grammar of Figure 5, and prove the equivalence stated in lemma `L_CBW_grammar`. Let's spell out the meaning of the LCBW grammar in Figure 5 and of the `L_CBW_grammar` lemma: an LCBW context (denoted by ${}^{w}V^{t}$ and `L_CBW`) is a hole or an application of weak normal form (denoted by w and `wnf`) to an LCBW context, or an application of an LCBW context to any term (denoted by t and ●).

Apart from this, we can also formalize and prove other properties, such as those stated in the remaining lemmas in Listing 7: LCBW contexts are weak (`L_CBW_is_Weak`), CBN contexts are exactly the weak head contexts (`CBN_Weak_Head`), and the auxiliary nonterminal in the grammar of leftmost-outermost contexts (denoted LO) describes exactly the rigid leftmost-outermost contexts (`RLO_Rigid_LO`).

Overlined nonterminals inform that they are rigid versions of their regular counterparts, i.e., not headed by a lambda abstraction. Letters around V in nonterminals inform which application frames appear in such call-by-value context family. Superscripts concern the weak fragment and subscripts the remaining part.

Listing 7 Example properties of contexts.

```
Definition L_CBW_frame (f:frame) : Prop :=
  match f with
  | Lapp e => wnf e
  | Rapp e => True
  | _      => False
  end.

Definition L_CBW  : 𝒫 context := Uniform L_CBW_frame.

Definition Lapp_of (t1:𝒫 term) (T2:𝒫 context) (C:context) : Prop :=
  match C with Lapp e1 :: C' => t1 e1 ∧ T2 C' | _ => False end.
Definition Rapp_of (T1:𝒫 context) (t2:𝒫 term) (C:context) : Prop :=
  match C with Rapp e2 :: C' => t2 e2 ∧ T1 C' | _ => False end.

Lemma L_CBW_grammar :
  L_CBW == Hole ∪ Lapp_of wnf L_CBW ∪ Rapp_of L_CBW ●.
Lemma L_CBW_is_Weak : L_CBW ⊆ Weak.
Lemma CBN_Weak_Head : CBN == Weak ∩ Head.
Lemma RLO_Rigid_LO : RLO == Rigid ∩ LO.
```

2.5 Decompositions and Strategies

The purpose of an (effective) reduction strategy is to indicate locations in a term where contraction can occur. It can be done using *decompositions* of a term into a designated subterm and its surrounding context. This approach does not require substitution to prove any of the properties of strategies that we study in this paper.

Strategies are defined as sets of decompositions, as shown in Listing 8. Recomposition is the uncurried `plug` function. With these definitions, we can define that a term is in normal form with respect to a given strategy if there is no decomposition of this term accepted by the strategy (cf. `normal_form`). We can also define that a strategy is deterministic if for any term there exists at most one of its decompositions accepted by the given strategy (cf. `det_strategy`). Our definition of strategy is still quite general and makes it possible to talk about undecidable strategies, but they are out of the scope of our interest.

Listing 8 Definitions of decomposition, strategy, normal forms and determinism.

```
Definition decomposition : Type := context * term.
Definition strategy      : Type := decomposition → Prop.

Definition recompose : decomposition → term := uncurry plug.

Definition normal_form (s:strategy) (t:term) : Prop :=
  ¬ ∃ d, t = recompose d ∧ d ∈ s.

Definition ex_le1 {A:Type} (P : A → Prop) : Prop :=
  ∀ x x', P x → P x' → x = x'.
Notation "∃≤1 x , p" := (ex_le1 (λ x, p))
  (at level 200, right associativity) : type_scope.

Definition det_strategy (s:strategy) : Prop :=
  ∀ t, ∃≤1 d, t = recompose d ∧ d ∈ s.
```

The final aspect to consider is how to recognize that a subterm plugged in the hole of a context can be contracted according to the given strategy. Often, such a term is called a *redex*. In the code, we use the name *contrex* (contractible expression) to stress that it has fit not just reduction, but exactly contraction. For example, we said that in the *lcbw* strategy a redex has to have the form $(\lambda x.t)\,w$. On the other hand, any term of the form $(\lambda x.t_1)\,t_2$ is a redex in *cbn*.

We define β, β_λ, β_w, β_{wh}, ${}_h\beta$, ${}_n\beta_n$ term families denoting accepted subjects of contraction as follows (letter symbols are as in Figure 1: $q \in whnf$, $w \in wnf$, $h \in hnf$, $n \in nf$; later on, the symbols β, β_λ, β_{wh} will also denote strategies that can perform the appropriate contraction only in the empty context):

$$\begin{array}{lll} \beta \ni (\lambda x.t_1)\,t_2 & \beta_w \ni (\lambda x.t)\,w & {}_h\beta \ni (\lambda x.h)\,t \\ \beta_\lambda \ni (\lambda x_1.t_1)\,(\lambda x_2.t_2) & \beta_{wh} \ni (\lambda x.t)\,q & {}_n\beta_n \ni (\lambda x.n_1)\,n_2 \end{array}$$

With these ingredients, we can now define specific strategies, e.g., *cbn* as CBN $\times\ \beta$ and *lcbw* as LCBW $\times\ \beta_w$. Example definitions are shown in Listing 9.

Listing 9 Example strategy definitions.

```
Definition    β_contrex  : term → Prop := app_of abstraction ●.
Definition  βwnf_contrex  : term → Prop := app_of abstraction wnf.
Definition  hnfβ_contrex  : term → Prop := app_of (abs_of hnf) ●.

Definition only_β_contraction : strategy :=  Hole × β_contrex.
Definition                cbn : strategy :=   CBN × β_contrex.
Definition              l_cbw : strategy := L_CBW × βwnf_contrex.
```

3 A trip to the zoo

3.1 Zoo picture description

Figure 6 depicts β-reduction strategies as Cartesian products of context families (located in the topmost row in the figure) and term families (located in the leftmost column). For example, *cbv* := Weak $\times\ \beta_\lambda$. There are some crossings that define strategies we are not interested in, e.g., LOW $\times\ \beta_w$. We mark them with "·", and we omit dots in the empty rectangles. In general there exist strategies that are not Cartesian products, e.g. an interbreeding: *cbn* ∪ *cbv*, but we have not found it interesting.

Some of the strategies are well known and can be identified by their reduction semantics from the literature: *cbn* as call by name, *no* as normal order, *lcbv* as left-to-right (closed) call by value [8]. Analogously, *rcbv* is right-to-left closed call by value and *cbv* is nondeterministic closed call by value. On the other hand, *cbw* can be identified as non-deterministic open call by value (a.k.a. fireball calculus) and *rcbw* as its right-to-left substrategy [3]. Analogously, *lcbw* is its left-to-right substrategy. In the following, we show that *llcbw* (and the three strategies written underneath it) are deterministic extensions of open call by value to strong call by value, and that *scbw* is their superstrategy, non-deterministic strong call by value.[3]

[3] Strong call by value is understood differently in [1]: it is not a superstrategy of open call by value, it is defined in a language with explicit substitutions and thus it does not fit directly into our zoo. Strong call by value presented here is a strategy studied in [7].

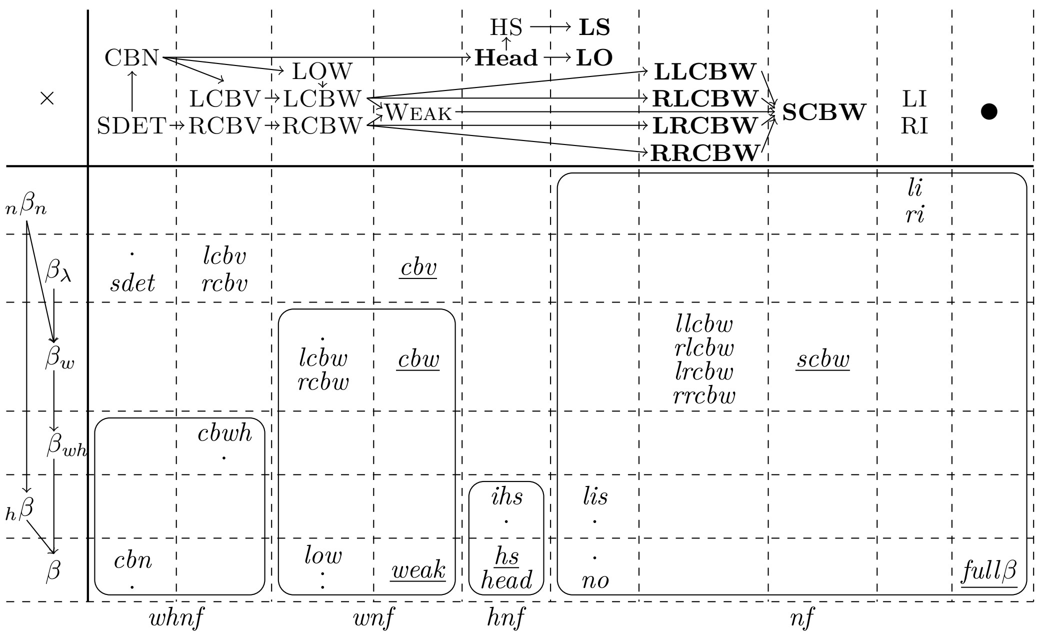

Figure 6 Zoo of strategies of the lambda calculus.

Furthermore, the strategies *weak*, *head*, *full*β, *li*, *ri* are the weak, head, full, leftmost-innermost and rightmost-innermost β-reductions, respectively. The remaining strategies (*sdet*, *cbwh*, *low*, *ihs* and *lis*) are discussed in Section 4.2.

The arrows shown in the topmost row and leftmost column represent family inclusions, such as `L_CBW_is_Weak` : LCBW $\subseteq$ WEAK. Thanks to the monotonicity of the Cartesian product, we can read inclusions of the strategies: *head* $\subseteq$ *no* because HEAD $\subseteq$ LO (and $\beta \subseteq \beta$), or *sdet* $\subseteq$ *lcbw* because SDET $\subseteq$ CBN $\subseteq$ LCBV $\subseteq$ LCBW and $\beta_\lambda \subseteq \beta_w$.

What is left to be demonstrated, is that the strategies living in the same pen (rounded rectangle) have the same normal forms, and these normal forms are written in the bottom of the figure under each pen. For example, normal forms of *head*, *ihs* and *hs* (but not of the dot for HEAD $\times\, {}_h\beta$) are exactly *hnf*. The strategies underlined in the figure are non-deterministic, and the non-underlined ones are deterministic. Context families in bold are non-uniform and the other ones are uniform (see definitions in Section 3.3).

3.2 Example reductions

In this section we show how strategies behave when we feed them with terms. To this end, we define some standard terms and their abbreviations[4]: $I := \lambda x.x$, $K := \lambda x.\lambda y.x$, $\omega := \lambda x.x\,x$, $\Omega := \omega\omega$, $(t_1, t_2) := \lambda f.f\,t_1\,t_2$, $(t,) := \lambda f.f\,t$, $\pi_1 := (K,)$.

The term $(I\,K)\,I$ reduces in *cbn*, but it is not contractible: $K\,I \overset{cbn}{\leftarrow} (I\,K)\,I \not\to_\beta$. In *cbn* II reduces to I which is its normal form ($II \overset{cbn}{\to} I \overset{cbn}{\not\to}$), because the decomposition of II into $\square$ and II is a *cbn*-decomposition, while the decomposition of I into $\square$ and I is not. The reason is that I is not contractible – it is not in the term family β.

[4] Notation $(t,)$ for 1-tuples comes from Python.

Call by name and call by value can go separate ways on the same term as shown by the following example: $\omega\, K \overset{cbn}{\leftarrow} I\,(\omega\, K) \overset{cbv}{\rightarrow} I\,(K\, K)$. Closed call by value can be stuck where the open is not: $\overset{lcbv}{\not\leftarrow} x\,(I\, x) \overset{lcbw}{\rightarrow} x\, x$. Weak strategies do not allow reduction under lambda abstractions: $(\lambda x.x\, I)\, K \overset{weak}{\not\leftarrow} (\lambda x.(\lambda y.x\, y)\, I)\, K \overset{weak}{\rightarrow} (\lambda y.K\, y)\, I$. For the same term, *head* and *ihs* can behave differently: $(\lambda y.K\, y)\, I \overset{head}{\leftarrow} (\lambda x.(\lambda y.x\, y)\, I)\, K \overset{ihs}{\rightarrow} (\lambda x.x\, I)\, K$. Normal-order reduction is known for avoiding nontermination whenever possible while leftmost-innermost and rightmost-innermost fall victim to it very easily: $I \overset{no}{\leftarrow} (\lambda x.I)\,(\Omega,) \overset{li,ri}{\rightarrow} (\lambda x.I)\,(\Omega,)$.

A very small strategy $\mathit{sdet} == \mathit{cbn} \cap \mathit{rcbv}$ is capable of performing pair projection: $\pi_1\,(K, I) \overset{sdet}{\rightarrow} (K, I)\, K \overset{sdet}{\rightarrow} K\, K\, I \overset{sdet}{\rightarrow} (\lambda y.K)\, I \overset{sdet}{\rightarrow} K$ and nontermination: $\Omega \overset{sdet}{\rightarrow} \Omega$.

3.3 Uniformity and determinism

There are some general properties that can be expressed and studied uniformly for any reduction strategy; by way of example we discuss two of such properties, illustrated already in Figure 6. Sestoft [31] observed that some strategies are uniform in the sense that the definition of such a strategy (in the big-step semantics) depends inductively only on that strategy itself, while other strategies are hybrid: they use some uniform ones as substrategies. Uniform and hybrid strategies were studied e.g. by García-Pérez and Nogueira [16]. Biernacka et al. [8] define a strategy to be uniform if its context family can be defined with a grammar with only one nonterminal symbol. That means that the shape of context frames can be uniformly checked for each frame separately, and it is reflected by our definition of `Uniform` (see Listing 10). To prove uniformity it is enough to define the given family in a uniform way. Disproving it is more complicated: one must show that every grammar with only one non-terminal symbol that generates all contexts in the family generates also some contexts that are not in this family.

Another property of interest concerns (non-)determinism. To refute determinism of a given strategy it is enough to show two different decompositions of the same term accepted by the strategy. Moreover, superstrategies of a non-deterministic strategy are non-deterministic (this property is reflected by `det_strategy_variance` in Listing 10 that says if s_1 is a substrategy of s_2 and s_2 is deterministic then s_1 is also deterministic). On the other hand, showing determinism is more demanding because it requires equating two decompositions of the same term accepted by a strategy. In order to prove it for more complex strategies, we will employ more sophisticated reasoning.

Listing 10 Uniformity and non-determinism.

```
Fixpoint Uniform (F:frame → Prop) (C:context) : Prop :=
  match C with
  | []     => True
  | f :: C => F f ∧ Uniform F C
  end.

Theorem uniform_strategies : ∀ C, In C [SDET; CBN; R_CBV; L_CBV;
  R_CBW; L_CBW; LOW; Weak; HS; RI; LI; ●] → ∃ F, C == Uniform F.
Theorem non_uniform_strategies : ∀ C, In C [Head; LO; LS;
  LL_CBW; LR_CBW; RL_CBW; RR_CBW; SCBW] → ¬ ∃ F, C == Uniform F.

Lemma det_strategy_variance : Proper (subset --> impl) det_strategy.

Theorem nondeterministic_strategies :
  ∀ s, In s [cbv; cbw; weak; hs; scbw; full_β] → ¬ det_strategy s.
```

4 Phased strategies

Reduction strategies are often implemented as procedures for locating the next redex to be contracted. For example, call by name can be defined as a strategy that reduces an application $t_1\,t_2$ by first reducing t_1 to weak-head normal form and then trying β-contraction. But then it is not clear if such a definition has anything in common with the product $\mathrm{CBN} \times \beta$, as defined in our formalization. In this section we present a novel semantic format of phased strategies that allows concise description of strategies in this (operational) spirit and facilitates algebraic reasoning about their properties. In particular we show that the two definitions of *cbn* (and similar definitions of all the other strategies in the zoo) are equivalent.

The basic idea of a phased strategy is that it is a recursively defined sequence of *phases*, each phase being a description of one small step in the search for the next redex. The phases are easy to read and to implement. For example the phased version of *lcbw* is $\swarrow lcbw; \searrow lcbw; \beta$. It consists of three phases, each of which tries to reduce an application of the form $t_1\,t_2$ (the absence of the constructor $\downarrow$ shows that *lcbw* never reduces under lambda abstractions). The first phase is $\swarrow lcbw$, it says: search for the redex in the left term, t_1, using the *lcbw* strategy. The second phase $\searrow lcbw$ says: if the first phase finds no redex, then continue the search in the right term, t_2, using the *lcbw* strategy. The third phase tries β-contraction when the first two phases find no redex.

Apart from their simplicity, phased strategies have other advantages. They are extremely concise in presentation, and they support equational reasoning that can be used to identify equal strategies that admit different definitions. The sequencing of phases preserves determinism, which greatly simplifies reasoning about deterministic strategies. In contrast to small-step semantics from Section 2.3, defining a phased strategy does not require knowledge of normal forms in this strategy. And in contrast to big-step semantics, the phased format can distinguish between divergent and stuck terms.

4.1 Definitions

The first ingredient of phased strategies is strategy sequencing. It joins two strategies, called phases, into one strategy that either makes a step of the first phase unconditionally, or it makes a step of the second phase – if the term is already in the normal form with respect to the first phase. This definition is formalized in Listing 11. On paper, we use the semicolon to denote sequencing, but in Coq we use double semicolon because the single one is already reserved. Strategy sequencing seems to be quite natural as strategies with the sequencing operator form a monoid.

Listing 11 Strategy sequencing monoid.

```
Definition sequence_strategy (r s: strategy) : strategy :=
  λ d, r d ∨ (normal_form r (recompose d) ∧ s d).
Infix ";;" := sequence_strategy (at level 60, right associativity).

Lemma sequence_strategy_empty_r : ∀ s, s;; ∅ == s.
Lemma sequence_strategy_empty_l : ∀ s, ∅;; s == s.
Lemma sequence_strategy_assoc : ∀ q r s, q;; (r;; s) == (q;; r);; s.
```

We define five unary strategy operators that allow us to peel one top frame from the term and execute the given strategy beneath. For example, $\swarrow cbn$ works only on applications and it can make a *cbn*-step on t_1 in $t_1\,t_2$. Similarly, $\downarrow cbn$ works only on abstractions. Analogously, $\overset{\lambda}{\swarrow} cbn$ can make a *cbn*-step on t_1 in $t_1\,t_2$ given that t_2 is already an abstraction. Formal definitions are given in Listing 12. We also use β, β_λ, β_{wh} to denote strategies that can only perform the contraction on the top term.

Listing 12 Phased strategies constructors.

```
Definition left_strategy (s : strategy) : strategy :=
  λ d, match d with (Rapp  _ :: C, c) => s (C, c) | _ => False end.
Definition right_strategy (s : strategy) : strategy :=
  λ d, match d with (Lapp  _ :: C, c) => s (C, c) | _ => False end.
Definition down_strategy (s : strategy) : strategy :=
  λ d, match d with (Lam   _ :: C, c) => s (C, c) | _ => False end.
Definition left_abs_strategy (s : strategy) : strategy :=
  λ d, match d with (Rapp t2 :: C, c) => abstraction t2 ∧ s (C, c)
                  | _                 => False end.
Definition right_abs_strategy (s : strategy) : strategy :=
  λ d, match d with (Lapp t1 :: C, c) => abstraction t1 ∧ s (C, c)
                  | _                 => False end.

Notation "↙"  := left_strategy.
Notation "↘"  := right_strategy.
Notation "↓"  := down_strategy.
Notation "↙λ" := left_abs_strategy.
Notation "↘λ" := right_abs_strategy.
Notation "'β'"   := only_β_contraction.
Notation "'βλ'"  := only_βλ_contraction.
Notation "'βwh'" := only_βwhnf_contraction.

Definition cbn_phased : strategy := ↙cbn;; β.
```

In Figure 7 we present formal definitions of 27 phased strategies, and at the same time we state 27 theorems formalizing how these strategies are equivalent to 24 strategies from the zoo of Figure 6. For example, the *cbn* strategy of Figure 6 can be shown to be equal to the phased strategy $\swarrow cbn; \beta$, and also to the phased strategy $\beta; \swarrow cbn$ (first line).

$$
\begin{array}{rcl@{\qquad}rcl}
cbn & = & \swarrow cbn; \beta = \beta; \swarrow cbn & weak & = & \swarrow weak \cup \searrow weak \cup \beta \\
head & = & (\beta; \swarrow head) \cup \downarrow head & lcbv & = & \swarrow lcbv; \overset{\lambda}{\searrow} lcbv; \beta_\lambda \\
ihs & = & (\swarrow ihs; \beta \quad) \cup \downarrow ihs & rcbv & = & \searrow rcbv; \overset{\lambda}{\swarrow} rcbv; \beta_\lambda \\
hs & = & \beta \cup \swarrow hs \cup \downarrow hs & cbv & = & \swarrow cbv \cup \searrow cbv \cup \beta_\lambda \\
no & = & \qquad (\beta; \swarrow no; \searrow no) \cup \downarrow no & lcbw & = & \swarrow lcbw; \searrow lcbw; \beta \\
lis & = & (\swarrow ihs; \beta; \swarrow lis; \searrow lis) \cup \downarrow lis & rcbw & = & \searrow rcbw; \swarrow rcbw; \beta \\
full\beta & = & \beta \cup \swarrow full\beta \cup \searrow full\beta \cup \downarrow full\beta & cbw & = & (\swarrow cbw \cup \searrow cbw); \beta \\
li & = & (\swarrow li; \searrow li; \beta) \cup \downarrow li & llcbw & = & (lcbw; \swarrow llcbw; \searrow llcbw) \cup \downarrow llcbw \\
ri & = & (\searrow ri; \swarrow ri; \beta) \cup \downarrow ri & rlcbw & = & (lcbw; \searrow rlcbw; \swarrow rlcbw) \cup \downarrow rlcbw \\
cbwh & = & \swarrow cbwh; \overset{\lambda}{\searrow} cbwh; \beta_{wh} & lrcbw & = & (rcbw; \swarrow lrcbw; \searrow lrcbw) \cup \downarrow lrcbw \\
sdet & = & \overset{\lambda}{\swarrow} sdet; \beta_\lambda = \beta_\lambda; \overset{\lambda}{\swarrow} sdet & rrcbw & = & (rcbw; \searrow rrcbw; \swarrow rrcbw) \cup \downarrow rrcbw \\
low & = & \beta; \swarrow low; \searrow low = \swarrow low; \beta; \searrow low & scbw & = & (cbw; (\swarrow scbw \cup \searrow scbw)) \cup \downarrow scbw
\end{array}
$$

Figure 7 Phased forms.

Phased strategy operators exhibit some algebraic properties that ease reasoning about them. Some examples are shown in Listing 13. Normal forms of a sequenced strategy are exactly terms that are normal w.r.t. both phases (`normal_form_sequence_strategy`).

The left phase of a weak strategy on the left application branch commutes with a β-phase (`left_weak_strategy_β_contraction_commutative`). It is because β needs an abstraction on the left to contract, and weak strategies do not reduce abstractions. This is why *cbn* has two different phased forms. Unary phased operators are distributive over sequencing (`phase_distributive_over_sequencing`). If a strategy is sequenced with its superstrategy that is deterministic then the obtained strategy is equivalent to the superstrategy alone (`deterministic_extension`). To prove that constructively, we need the decidability of the smaller strategy to decide if the term is its normal form in the proof of the right-to-left inclusion.

Listing 13 Example properties of phased strategy operators.

```
Lemma normal_form_sequence_strategy : ∀ r s,
  normal_form (r;; s) == normal_form r ∩ normal_form s.
Lemma left_weak_strategy_β_contraction_commutative : ∀ w,
  w ⊆ weak →↙ w;; β == β;; ↙ w.
Theorem phase_distributive_over_sequencing : ∀ X,
  In X [↙;↘;↓;↙λ;↘λ] → ∀ s s', (X s;; X s') == X (s;; s').
Theorem deterministic_extension : ∀ s1 s2, (∀ x, {x ∈ s1} + {x ∉ s1})
  → det_strategy s2 → s1 ⊆ s2 → s1;; s2 == s2.
```

4.2 Benefits of phased strategies

In order to see the benefits of phased formulation of the strategies, we first establish the equalities from Figure 7. For many strategies, if we want to define their reduction semantics we need to discover what normal forms are with respect to the strategy we are defining. Most of the time, we prove the equality between reductive form and phased form by induction over terms, where the induction hypothesis is extended with the equality between a fixed family of normal forms and normal forms of the phased form. Then we get theorems about normal forms as corollaries. We collect them in Listing 14.

Listing 14 Normal forms and determinism theorems.

```
Theorem weak_head_normal_forms : ∀ s, In s [cbn; cbwh] →
  normal_form s == whnf.
Theorem weak_normal_forms : ∀ s, In s [l_cbw; r_cbw; cbw; low; weak]
  → normal_form s == wnf.
Theorem head_normal_forms : ∀ s, In s [head; ihs] →
  normal_form s == hnf.
Theorem full_normal_forms : ∀ s, In s [no; lis; ll_cbw; lr_cbw;
  rl_cbw; rr_cbw; scbw; li; ri; full_β] → normal_form s == nf.

Theorem deterministic_strategies : ∀ s, In s
  [sdet; cbn; l_cbv; r_cbv; cbwh; l_cbw; r_cbw; low; ihs; head;
   lis; no; ll_cbw; lr_cbw; rl_cbw; rr_cbw; li; ri] → det_strategy s.
```

As the phased form is affirmed, determinism of the strategies can be proven by simple, repetitive inductions over terms. This seems to be a more structured approach than ad-hoc proofs of determinism (cf. the proof of determinism of *rrcbw* in [7]).

Similarly, the reduction semantics of *lis* $=$ LS $\times_h \beta$ (leftmost-innermost-spine) is quite involved (cf. Figure 5). Thanks to the phased formulation, it is clearer how it works and that the reduction semantics formulation is indeed correct (cf. Figure 7). We can see that *head* and *ihs* (innermost-head-spine) are two different deterministic, head-reducing strategies. Both are extended to two different full-reducing strategies: *lis* and *no*. The study of *ihs* and *lis*, studied earlier by Barendregt et al. [6], is particularly interesting because they exhibit some call-by-need traits. For example, their decomposition function reaches the head variable before any reduction takes place.

The framework of phased strategies facilitates the study of new strategies. It is widely known that *cbn* performs weak-head reduction. We have formulated the *cbwh* strategy that performs weak-head reduction of arguments before they are substituted, and we have shown that it is also a weak-head-reducing strategy. Thus, it enjoys Accattoli and Guerrieri's harmony property [2].

By swapping the phases of *lcbw* and *rcbw*, we have discovered the *low* (leftmost-outermost-weak) strategy. It resembles *no* but does not go under lambda abstractions. It is a complete weak-reducing strategy, i.e., it always reaches a weak normal form if it exists.

We have formulated the *sdet* strategy that is a substrategy of all strategies mentioned in Figure 6, except for *ihs*, *lis*, *li*, and *ri*. This strategy is sufficient to run Dal Lago and Accattoli's simulation of Turing machines [19], and so all of its superstrategies are also sufficient.

Finally, in Figure 8 we show an example of algebraic reasoning, where we use some of the algebraic laws that we proved. This example demonstrates that *lrcbw* (right-to-left-to-right call by value), which is one of the four self-evident deterministic strong call-by-value evaluation orders, has a special phased form with left weak phase optimized away. It corresponds to a special optimization in its implementation by an abstract machine [7]. Similarly, we can show that *scbw* is a conservative extension of *cbw*, and formalize it as $scbw == cbw; scbw$. Thus *scbw* intuitively inherits the strong confluence (diamond property) of *cbw* [2].

$$
\begin{aligned}
lrcbw &== (rcbw; \swarrow lrcbw; \searrow lrcbw) \cup \downarrow lrcbw \overset{\texttt{sequence\_strategy\_assoc}}{==} \\
&== (\searrow rcbw; \swarrow rcbw; \beta; \swarrow lrcbw; \searrow lrcbw) \cup \downarrow lrcbw \overset{\texttt{left\_weak\_strategy\_}\beta\texttt{\_...\_commutative}}{==} \\
&== (\searrow rcbw; \beta; \swarrow rcbw; \swarrow lrcbw; \searrow lrcbw) \cup \downarrow lrcbw \overset{\texttt{phase\_distributive\_over\_sequencing}}{==} \\
&== (\searrow rcbw; \beta; \swarrow (rcbw; lrcbw), \searrow lrcbw) \cup \downarrow lrcbw \overset{\texttt{deterministic\_extension}}{==} \\
&== (\searrow rcbw; \beta; \swarrow lrcbw; \searrow lrcbw) \cup \downarrow lrcbw
\end{aligned}
$$

Figure 8 An equational reasoning on the *lrcbw* phased form.

4.3 Phased strategies on their own

Phased strategies as presented above depend on the prior definitions of strategies. Nevertheless, there are properties inviting to define them independently. Listing 15 shows that *cbn* is the strategy unique up to set equality that satisfies the equation $cbn == \swarrow cbn; \beta$. From this follows that *cbn* is the smallest and the largest of such strategies. However, these definitions (with $\bigcap$ and $\bigcup$) were again impractical because we could not prove the identity of normal forms without use of *cbn*.

Listing 15 Uniqueness of *cbn*.

```
Definition cbn_eqn (s : strategy) : Prop := s == ↙s;; β.
Lemma       cbn_unique : ∀ s, cbn_eqn s → s == cbn.
Theorem   cbn_ind_form : cbn == ⋂ cbn_eqn.
Theorem cbn_coind_form : cbn == ⋃ cbn_eqn.
```

Nonetheless, a standalone definition in Coq is possible. We can define counterparts of strategy operators working on decomposition functions (e.g., `sequence_decompose`). Then we can define a decomposition function that corresponds to a phased form, take it as a base of the strategy and prove the set equality of the strategies as presented in Listing 16.

Listing 16 Independent phased definition of *cbn*

```
Fixpoint cbn_decompose (t : term): option decomposition :=
  sequence_decompose
    (left_decompose cbn_decompose)
    contrex_decompose
  t.
Definition decompose_strategy f := λ d, f (recompose d) = Some d.
Definition cbn_decomposition := decompose_strategy cbn_decompose.
Theorem cbn_decompose_form : cbn == cbn_decomposition.
```

5 Conclusion

We have presented a minimalistic, concise formalization of a class of reduction strategies in the λ-calculus, that are representable as term decompositions. It can be extended to richer languages based on the lambda calculus, or adapted to other term-rewriting formalisms. We have collected and systematized existing strategies, and shown how some of their properties can be proved in our framework. Finally, we have introduced a novel semantic format of phased strategies that enables simple equational reasoning about strategies. As future work, we plan to accommodate further strategies within our framework, such as variants of call-by-need strategies, or optimal strategies [20, 4] and study abstract machines derived from phased forms.

References

1 Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–14. IEEE, 2021. `doi:10.1109/LICS52264.2021.9470630`.

2 Beniamino Accattoli and Giulio Guerrieri. Open call-by-value. In Atsushi Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 206–226, 2016. `doi:10.1007/978-3-319-47958-3_12`.

3 Beniamino Accattoli and Giulio Guerrieri. Abstract machines for open call-by-value. *Sci. Comput. Program.*, 184, 2019. `doi:10.1016/j.scico.2019.03.002`.

4 Thibaut Balabonski. Weak optimality, and the meaning of sharing. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 263–274. ACM, 2013. `doi:10.1145/2500365.2500606`.

5 Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

6 Hendrik Pieter Barendregt, Richard Kennaway, Jan Willem Klop, and M. Ronan Sleep. Needed reduction and spine strategies for the lambda calculus. *Inf. Comput.*, 75(3):191–231, 1987. `doi:10.1016/0890-5401(87)90001-0`.

7 Malgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. An abstract machine for strong call by value. In Bruno C. d. S. Oliveira, editor, *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, volume 12470 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2020. `doi:10.1007/978-3-030-64437-6_8`.

8 Malgorzata Biernacka, Witold Charatonik, and Klara Zielinska. Generalized refocusing: From hybrid strategies to abstract machines. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 10:1–10:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.FSCD.2017.10`.

9 Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2013. `doi:10.1007/978-3-642-37036-6_3`.

10 Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics, 2004.

11 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 2009. URL: `http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885`.

12 Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992. `doi:10.1016/0304-3975(92)90014-7`.

13 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ-calculus is reasonable for both time and space. *Proc. ACM Program. Lang.*, 4(POPL):27:1–27:23, 2020. `doi:10.1145/3371095`.

14 Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. A mechanised proof of the time invariance thesis for the weak call-by-value λ-calculus. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 19:1–19:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ITP.2021.19`.

15 Yannick Forster and Gert Smolka. Call-by-value lambda calculus as a model of computation in coq. *J. Autom. Reason.*, 63(2):393–413, 2019. `doi:10.1007/s10817-018-9484-2`.

16 Álvaro García-Pérez and Pablo Nogueira. On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. *Sci. Comput. Program.*, 95:176–199, 2014. `doi:10.1016/j.scico.2014.05.011`.

17 Gérard P. Huet. Residual theory in lambda-calculus: A formal development. *J. Funct. Program.*, 4(3):371–394, 1994. `doi:10.1017/S0956796800001106`.

18 Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987. `doi:10.1007/BFb0039592`.

19 Ugo Dal Lago and Beniamino Accattoli. Encoding turing machines into the deterministic lambda-calculus. *CoRR*, abs/1711.10078, 2017. `arXiv:1711.10078`.

20 Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 160–191. Academic Press, 1980.

21 Conor McBride. The derivative of a regular type is its type of one-hole contexts (extended abstract), 2001.

22 James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reason.*, 23(3-4):373–409, 1999. `doi:10.1023/A:1006294005493`.

23 Michael Norrish. Mechanising lambda-calculus using a classical first order theory of terms with permutations. *High. Order Symb. Comput.*, 19(2-3):169–195, 2006. `doi:10.1007/s10990-006-8745-7`.

24 Lawrence C. Paulson. *ML for the working programmer (2. ed.).* Cambridge University Press, 1996.

25 Frank Pfenning. A proof of the church-rosser theorem and its representation in a logical framework, 1992. URL: `https://www.cs.cmu.edu/~fp/papers/cr92.pdf`.

26 Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018. Version 5.5.

27 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. `doi:10.1016/0304-3975(75)90017-1`.

28 Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

29 Casper Bach Poulsen and Peter D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 270–289. Springer, 2014. `doi:10.1007/978-3-642-54833-8_15`.

30 John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. `doi:10.1023/A:1010027404223`.

31 Peter Sestoft. Demonstrating lambda calculus reduction. In Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 420–435. Springer, 2002. `doi:10.1007/3-540-36377-7_19`.

32 Natarajan Shankar. A mechanical proof of the church-rosser theorem. *J. ACM*, 35(3):475–522, 1988. `doi:10.1145/44483.44484`.

33 The Coq Development Team. The Coq proof assistant, January 2021. `doi:10.5281/zenodo.4501022`.

34 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001. `doi:10.1007/3-540-45127-7_27`.