



HAL
open science

Equivalence of eval-readback and eval-apply big-step evaluators by structuring the lambda-calculus's strategy space

Pablo Nogueira, Alvaro Garcia Perez

► **To cite this version:**

Pablo Nogueira, Alvaro Garcia Perez. Equivalence of eval-readback and eval-apply big-step evaluators by structuring the lambda-calculus's strategy space. Logical Methods in Computer Science, 2024, 20 (4), pp.1-76. 10.46298/LMCS-20(4:24)2024 . cea-04955542

HAL Id: cea-04955542

<https://cea.hal.science/cea-04955542v1>

Submitted on 18 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

EQUIVALENCE OF EVAL-READBAC AND EVAL-APPLY BIG-STEP EVALUATORS BY STRUCTURING THE LAMBDA-CALCULUS'S STRATEGY SPACE

PABLO NOGUEIRA  AND ÁLVARO GARCÍA-PÉREZ 

^a UDIT University of Design, Innovation and Technology, Madrid, Spain.
e-mail address: pablo.nogueira@udit.es

^b Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France.
e-mail address: alvaro.garciaperez@cea.fr

ABSTRACT. We study the equivalence between eval-readback and eval-apply big-step evaluators in the general setting of the pure lambda calculus. We study ‘one-step’ equivalence (same strategy) and also discuss ‘big-step’ equivalence (same final result). One-step equivalence extends for free to evaluators in other settings (calculi, programming languages, proof assistants, etc.) by restricting the terms (closed, convergent) while maintaining the strategy. We present a proof that one-step equivalence holds when (1) the ‘readback’ stage satisfies straightforward well-formedness provisos, (2) the ‘eval’ stage implements a ‘uniform’ strategy, and (3) the eval-apply evaluator implements a ‘balanced hybrid’ strategy that has ‘eval’ as a subsidiary strategy. The proof proceeds by the ‘lightweight fusion by fixed-point promotion’ program transformation on evaluator implementations to fuse readback and eval into the balanced hybrid. The proof can be followed with no previous knowledge of the transformation. We use Haskell 2010 as the implementation language, with all evaluators written in monadic style to guarantee semantics (strategy) preservation, but the choice of implementation language is immaterial.

To illustrate the large scope of the equivalence, we provide an extensive survey of the strategy space using canonical eval-apply evaluators in code and big-step ‘natural’ operational semantics. We discuss the strategies’ properties, some of their uses, and their abstract machines. We improve the formal definition of uniform and hybrid strategy, use it to structure the strategy space, and to obtain generic higher-order evaluators which are used in the equivalence proof. We introduce a systematic notation for both evaluator styles and use it to summarise strategy and evaluator equivalences, including (non-)equivalences within a style and (non-)equivalences between styles not proven by the transformation.

Key words and phrases: Lambda calculus, big-step evaluators, definitional interpreters, evaluation strategies, operational semantics, program transformation, program equivalence, functional programming.

CONTENTS

Extended abstract	3
1. Introduction	5
1.1. The background	5
1.2. The problem	6
1.3. The setting	6
1.4. The contributions	8
2. Technical preliminaries	12
3. Call-by-value policy, strategies, and calculi	12
4. The eval-apply evaluator style	15
4.1. Monadic style eval-apply	16
4.2. Natural semantics style eval-apply	18
5. A survey of the pure lambda calculus's strategy space	19
5.1. Call-by-value (bv)	20
5.2. Call-by-name (bn)	22
5.3. Applicative order (ao)	22
5.4. Normal order (no)	23
5.5. Head reduction (hr)	24
5.6. Head spine (he)	24
5.7. Strict normalisation (sn)	25
5.8. Hybrid normal order (hn)	26
5.9. Hybrid applicative order (ha)	26
5.10. Ahead machine (am)	28
5.11. Head applicative order (ho)	28
5.12. Spine applicative order (so)	28
5.13. Balanced spine applicative order (bs)	29
5.14. Summary of the survey	29
6. The eval-readback evaluator style	29
6.1. Monadic style eval-readback	32
6.2. Natural semantics style eval-readback	34
6.3. Characterisation of the style	34
6.4. Equivalences within the style	35
7. Structuring the pure lambda calculus's strategy space	36
7.1. Evaluation templates and generic evaluators	36
7.2. Uniform/Hybrid as structuring criteria	38
7.3. Structuring the uniform space	42
7.4. Structuring the hybrid space	45
7.5. Structuring the eval-readback space	49
8. Equivalence proof by LWF transformation	50
8.1. Plain but arbitrary eval-readback evaluators	50
8.2. Plain but arbitrary balanced hybrid eval-apply evaluators	52
8.3. Statement of one-step equivalence	53
8.4. Proof by LWF transformation	53
9. Summary and conclusions	60
10. Related work	64
11. Future work	66

Acknowledgements	66
Appendix A. Pure lambda calculus concepts and notation	66
Appendix B. Call-by-value in the lambda-value calculus	68
Appendix C. Overview of LWF	69
Appendix D. Haskell's operational semantics	70
Appendix E. Completeness of leftmost and spine strategies	71
References	71

EXTENDED ABSTRACT

Big-step evaluators are ubiquitous in formal semantics and implementations of higher-order languages based on the lambda calculus, whether research languages or core languages of functional programming languages and proof-assistants. A big-step evaluator is an operational semantics device that defines an evaluation strategy (an evaluation order of redexes) by specifying how the redexes are located and contracted. It is ‘big-step’ because it delivers the final result of iterated small-step reduction. (Because an evaluator defines a unique strategy, it is common to blur the distinction and call the evaluator a strategy.) A big-step evaluator is typically written as a recursive function in natural semantics or in code. In the latter, the operational semantics of the implementation language must be taken into account to guarantee semantics (strategy) preservation.

Whether in code or natural semantics, big-step evaluators are typically written in two definitional styles, namely, *eval-apply* and *eval-readback*. In *eval-apply* style, the evaluator is defined by a single natural semantics or a single ‘eval’ function in code. The ‘eval’ function can have a local ‘apply’ function for application terms which can be obviated with `let`-clauses or pattern-matching expressions. In *eval-readback* style, the evaluator is defined by the composition of two functions (two natural semantics): an ‘eval’ function that evaluates terms up to a certain intermediate form, and an ‘readback’ function that pushes down eval directly or through an *eval-readback* composition on specific subterms of the intermediate form.

The problem we address is the *one-step equivalence* (‘define the same strategy’) between *eval-readback* and *eval-apply* big-step evaluators, and address ‘big-step’ equivalence (‘deliver the same final result’) in passing. We use the pure lambda calculus as the most general setting with more terms (closed, open, neutral, divergent, convergent, etc.) and strategies (weak-reducing, head-reducing, full-reducing, strict, non-strict, etc.), where the different evaluation criteria and their interaction give rise to different kinds of final results (weak, head, weak head, value head, and plain normal forms, etc.) and properties (completeness, spineness, etc.). One-step equivalence extends for free to evaluators in other settings (programming languages, proof assistants, etc.) by restricting the set of terms (*e.g.* type-erased closed terms for programming languages, convergent terms for proof assistants) while the strategy is maintained.

In previous work, we showed that the one-step equivalence between one *eval-readback* and one *eval-apply* evaluator can be proven *in code* using the ‘lightweight fusion by fixed-point promotion’ program transformation (LWF) which fuses the *eval-readback* composition into the single *eval-apply* evaluator by means of strategy-preserving, simple and syntactic program-transformation steps that are embeddable in a compiler’s inlining optimisation.

However, this method has limitations. LWF has to be deployed for every pair of eval-readback and eval-apply evaluators, some strategies have known evaluators only in one of the two styles, and LWF is not invertible, that is, given an eval-apply with no known eval-readback, the latter must be conjectured and must LWF-transform to the eval-apply evaluator. Whether an eval-readback evaluator exists for a given eval-apply evaluator is precisely the open question of expressive power derived from the open question of equivalence.

The main contribution of this paper is a single proof of one-step equivalence by a novel application of LWF on an eval-readback and an eval-apply *arbitrary* evaluators which respectively call (instantiate) a *generic* readback evaluator and a *generic* eval-apply evaluator that implement as fixed-points all the concrete readback and eval-apply evaluators for the standard (history-free, deterministic, non-parallel, left-to-right) strategies in the literature and more. The pair of arbitrary evaluators have type-checked but undefined (*i.e.* existentially quantified) parameters whose values are constrained by the well-formedness provisos of the generic evaluators and by the LWF steps. The constraints are elaborated during the transformation and collected at the end in a list of equations that capture the conditions for one-step equivalence.

We provide an extensive overview of the eval-apply and eval-readback styles using canonical examples from the literature. To illustrate the large scope of the equivalence proof, we provide an extensive survey of the strategy space using canonical eval-apply evaluators in natural semantics and code. We discuss the strategies' diverse properties, abstract machines, and uses, particularly in relation to evaluation of recursive functions in direct versus continuation-passing style and thunking. We introduce new strategies in the survey, in particular, an eager strategy that evaluates recursive functions in delimited continuation-passing style with a non-strict fixed-point combinator and thunking 'protecting by variable' rather than the usual 'by lambda'.

We motivate and arrive at an improved formal definition of uniform/hybrid strategy that we use to orthogonally split the diverse strategy space. We structure each uniform and hybrid strategy subspace according to the weakness, strictness, and headness properties of strategies. The uniform space makes up a lattice of eight strategies we call Gibbons's Beta Cube. The hybrid space is larger and is further split by introducing the 'balanced' concept (a hybrid strategy is balanced when it is non-strict or when it uses its subsidiary strategy, not itself, to evaluate operands of redexes). We introduce the well-formedness provisos that make uniform and hybrid eval-apply evaluators define uniform and hybrid strategies.

We introduce systematic notations for encoding uniform, hybrid, and eval-readback evaluators, and use the notations to introduce new strategies, study (non-)equivalences and other properties such as absorption, and provide an extensive table summary of strategies, evaluators, and equivalences. We end discussing the relevance of our contributions in the wider context of correspondences among operational semantics devices.

We use Haskell 2010 as the implementation language, but the choice of language is immaterial. We ignore efficiency issues because in the pure lambda calculus there are no optimal strategies, and because efficiency can be studied afterwards in a strategy space constrained by efficiency considerations.

We have addressed the paper to non-expert readers which may be unfamiliar with background material. We are explanatory in the text (or in the appendices when appropriate) for information or reminder. Expert readers can jump directly to the sections of their interest. The introduction elaborates this extended abstract with details, citations, and a list of contributions that serves as a roadmap of the paper.

1. INTRODUCTION

1.1. The background. Big-step evaluators are ubiquitous in formal semantics and implementations of higher-order languages based on the lambda calculus. A big-step evaluator is an intensional function definition (a *definiens*) for an evaluation strategy (the unique *definiendum*). An evaluation strategy (or just ‘strategy’) is synonymous with the evaluation order of the reducible expressions (‘redexes’) of terms. A strategy is defined generically as a function subset of the calculus’s reduction relation. We are interested in history-free sequential strategies where the choice of redex is based on fixed positional criteria independent of previous choices. A big-step evaluator is an operational semantics device that specifies how the redexes are located and evaluated. It is ‘big-step’ because it delivers the *final* result of iterated small-step reduction. A strategy can also be defined in small-step operational semantics styles such as structural [Plo81], context-based [Fel87, FH92, FFF09], or abstract machines [Lan64, DHS00]. Because an evaluator defines a unique strategy, it is common to blur the distinction and call the evaluator a strategy. A big-step evaluator is also called a ‘definitional interpreter’ [Rey72]. Some authors reserve the latter for evaluators with shallow (meta-level) embeddings, but ‘interpreter’ (symbolic evaluation) predates the deep versus shallow embedding dichotomy.

A big-step evaluator is typically written as a recursive function, in code [McC60] or in natural semantics [Kah87]. In code, the evaluator works on some data-type representation of the mathematical terms. The operational semantics of the implementation language must be taken into account to guarantee semantics (*i.e.* strategy) preservation [Rey72]. In natural semantics, the evaluator works on the mathematical terms and is defined by a set of syntax-directed and deterministic inference rules describing the function’s behaviour recursively on subterms. A natural semantics provides a compact definition, requires no knowledge of specific programming languages, and serves as a specification of the evaluator’s implementation in code. In a partiality setting, a big-step evaluator may terminate (converge) on a final result, or non-terminate (diverge) because it cannot deliver a final result when it exists (which may be delivered by other strategies) or because no final result exists (which may be delivered by no strategy).

Whether in code or natural semantics, big-step evaluators are typically written in two definitional *styles*.

In *eval-apply* style, the evaluator is defined by a *single function*. This style appeared early in programming and formal semantics [McC60, Lan64, Rey72, Plo75, Sto77, SS78] and was the main style until the 1990s [All78, ASS85, FH88, Hen87, Rea89, Mac90]. In its traditional code definition, the `eval` function evaluates subterms recursively and delegates the specific evaluation of application terms to a mutually-recursive local `apply` function. This function can be obviated within `eval` using `let`-clauses and pattern-matching expressions in modern functional languages. In natural semantics, the eval-apply evaluator is defined by a single set of inference rules for every case of term: variables, abstractions (function bodies), and redex and non-redex applications. The `apply` function corresponds to the inference rules for applications. Section 4 explains the eval-apply style with examples.

In *eval-readback* style, the evaluator is defined by a *composition of two functions* (also called stages), namely, `readback` after `eval`. The name ‘eval-readback’ comes from [GL02] but the style is older. Two classic evaluators in eval-readback style are `byName` and `byValue` in [Pau96, p. 390]. The eval-readback style is related to normalisation-by-evaluation [BS91] but should not be confused with it. In normalisation-by-evaluation, a term is evaluated

by first constructing its representative in a different target domain (evaluation stage) and then reading the representative back to a canonical result term (reification stage). In eval-readback there is only the domain of terms. The *less-reducing eval* function contracts redexes to an intermediate result with unevaluated subterms. The *further-reducing readback* does not contract redexes, it ‘reads back’ the intermediate result to a further-reduced result by recursively distributing eval down the unevaluated subterms to reach further redexes [GL02, p.236]. Readback is a sort of ‘selective-iteration-of-eval’ function that calls eval directly or in an eval-readback composition on specific unevaluated subterms. Section 6 explains the eval-readback style with examples. The separation of evaluation in two stages can be of help when writing an evaluator. The separation also enables independent optimisations. For instance, in [GL02] the eval stage is implemented by an abstract machine and its associated compilation scheme [Ler91].

1.2. The problem. We address the question of equivalence between eval-readback and eval-apply evaluators, which partly answers the question of their expressive power, *i.e.* whether a strategy can or cannot be defined in both styles. We are primarily interested in whether the evaluators define the same strategy (evaluate redexes in exactly the same order) and secondarily in whether the evaluators do not define the same strategy but deliver identical final results. We refer to the former equivalence as *one-step equivalence* and to the latter as *big-step equivalence*. We aim for a *single* one-step equivalence proof for a large collection of strategies.

In previous work [GPN14, GPNS14], we showed that given an eval-readback evaluator $\mathbf{rb} \circ \mathbf{ev}$, its one-step equivalence with a given eval-apply evaluator \mathbf{ea} can be proven *in code* using the ‘lightweight fusion by fixed-point promotion’ program transformation of [OS07], which we hereafter abbreviate as LWF. (We also abbreviate evaluator names using two letters (*e.g.* \mathbf{rb} , \mathbf{ev} , etc.) like [Ses02], to fit their code and natural semantics definitions in running text and figures.) The LWF transformation fuses $\mathbf{rb} \circ \mathbf{ev}$ into \mathbf{ea} by means of strategy-preserving, simple and syntactic program-transformation steps that are embeddable in a compiler’s inlining optimisation. The LWF transformation had been previously used to prove equivalences of operational semantics devices, but in so-called ‘syntactic-correspondence’ proofs between small-step evaluators and abstract machines, where LWF is used to fuse the decomposition and recomposition steps, *e.g.* [Dan05, DM08, DJZ11].

Using LWF to prove eval-apply and eval-readback equivalence has two problems. First, using LWF for every possible strategy and pair of evaluators is menial work. Second, only one evaluator style is known for particular strategies. Third, LWF is not invertible. An eval-readback evaluator is not obtained by ‘de-fusing’ a given eval-apply evaluator. The eval-readback evaluator must be conjectured, and then it must LWF-transform to the given eval-apply evaluator. Whether an eval-readback evaluator exists for a given eval-apply evaluator is precisely the open question of expressive power.

1.3. The setting. We address the problem in the setting of the *pure lambda calculus* for the following reasons. First, there are too many applied and typed lambda calculi¹ with significant differences among them. There is no principled reason to pick one in particular.

¹ An *applied* extension of the pure lambda calculus adds primitive terms and reduction rules for them. A *typed* extension adds type terms and type rules, and possibly modifies terms and reduction rules with type information. Most extensions combine both.

Second, the pure lambda calculus is the most general calculus, with more terms, strategies, and concomitant big-step evaluators. Third, we can restrict the set of terms (*e.g.* closed terms, convergent terms obtained by type-checking and type-erasure, etc.) as required in extended settings such as programming languages and proof assistants. We define strategies and evaluators for arbitrary pure terms, prove equivalences between evaluator styles, and then we can assume a restricted input term subset so that the identical or one-step equivalent programming or proof-assistant evaluator is *de facto* included in the equivalence (Section 3).

In the same line, we represent terms in code with a direct data-type representation (a ‘deep embedding’ [BGG⁺92]), and use a direct implementation of the capture-avoiding substitution operation of the calculus. The results presented in this paper extend to other term representations provided the evaluation order of redexes is preserved. The results also extend to calculi where the strategies we study can be simulated, *e.g.* [Plo75, Sec. 6].

We ignore efficiency issues because in the pure lambda calculus there are no optimal strategies—there exist terms for which any strategy duplicates work [Lév78, Lév80]. Our goal is to structure the strategy space to prove one-step equivalence. Efficiency can be studied afterwards, constraining the strategy space by efficiency considerations. Also, efficient versions of strategies may be undefinable in the pure lambda calculus. For example, call-by-need is an efficient version of call-by-name that evaluates copies of the same redex once but requires sharing and memoisation [FH88]. The results in this paper can inform further work for such strategy spaces (see contribution **C1**). For the wide-range of issues regarding efficiency see [Lév80, FH88, Lam89, GAL92, AG98, RP04, AL16, ALV21].

The pure lambda calculus allows free variables and hence *open* terms, with the particular case of *neutral terms* (*neutrals*, for short).² A free variable is not a parameter of an abstraction. The notion is relative to a term: x may be free in B but bound (not free) in $\lambda x.B$. Neutral terms are applications of the form $x N_1 \cdots N_n$ where $n \geq 1$. A neutral is not a redex, and cannot evaluate to a redex unless a suitable term can be substituted for the leftmost variable operator (called the ‘head variable’). When the head variable is free, it may be seen as a data-type value constructor that is applied to the n operands.

The pure lambda calculus allows strategies with diverse evaluation properties, in particular:

- Weak vs non-weak evaluation: respectively, the non-evaluation or evaluation of abstractions (function bodies) as in programming languages. Mind the opposites: weak = non-evaluation of abstractions; non-weak = evaluation of abstractions.³
- Strict vs non-strict evaluation: respectively, the evaluation or non-evaluation of operands of redexes before substitution. Technically, strictness is a denotational property of functions but it can be ‘abused’ as an evaluation property.
- Head vs non-head evaluation: respectively, the non-evaluation or evaluation of (operands of) neutrals, which gives the name to strategies such as ‘head reduction’ or ‘head spine’ (Section 5). Mind the opposites: head = non-evaluation of neutrals; non-head = evaluation of neutrals.
- Full evaluation (hereafter, ‘full-reduction’): evaluation to a result term with no redexes. Full-reduction with open terms is fundamental in optimisation by partial evaluation and type-checking in proof-assistants and automated reasoning, *e.g.* [GL02, Cré07, ACP⁺08,

²We thank Noam Zeilberger for pointing us to the ‘neutral’ terminology.

³Weak evaluation is called ‘lazy evaluation’ by several authors by association with the ‘lazy lambda calculus’ [Abr90]. But that calculus is weak *and* non-strict, two properties of lazy functional programming languages.

SR15, ABM15]. Many authors use ‘strong reduction’ rather than ‘full-reduction’. However, ‘strong’ is the antonym of ‘weak’ which only refers to the non-evaluation of abstractions. Full-reduction ‘goes under lambda’ (non-weak) and also under neutrals (non-head). Also, strong reduction may be confused with ‘strong normalisation’ which is a property of terms.⁴ Thus, we use ‘full-reduction’ [SR15] with a dash to make it a technical name.

These evaluation criteria *and their interaction* give rise to different kinds of final results and properties. For example, a weak and strict strategy does not evaluate the abstractions within the operands of redexes it does evaluate. A non-weak and head strategy does not evaluate the neutrals within the abstractions it does evaluate.

1.4. The contributions. The last contribution of this paper is a single proof of the one-step equivalence of eval-readback and eval-apply evaluators. The proof proceeds by a novel application of LWF on a plain but arbitrary eval-readback evaluator and eval-apply evaluator. They are ‘plain’ in that they respectively call (instantiate) a *generic* readback evaluator and a *generic* eval-apply evaluator that respectively implement as fixed-points all the plain and concrete readback and eval-apply evaluators for the standard (history-free, deterministic, non-parallel, left-to-right) strategies in the literature and more. They are ‘arbitrary’ in that some parameters to the generic evaluators have undefined (but type-checked) values, which are thus ‘existentially quantified’ in the sense of logic programming. The LWF steps are applied verbatim on the two plain and arbitrary evaluators. The concrete values substitutable for the undefined parameters are constrained by well-formedness provisos of the generic evaluators and by the LWF steps, and are collected at the end in a list of equations that capture the conditions for one-step equivalence.

This last contribution strongly depends on other significant contributions that we list below in increasing order of importance and of presentation in the paper. The list of contributions below also serves as a roadmap summary of the paper. In Section 9 we provide a detailed technical summary of the contributions that can be understood after reading the paper.

C1 Eval-apply style and strategy survey. We provide an extensive survey of pure lambda calculus strategies defined by their canonical eval-apply evaluators in natural semantics (Section 5). We previously overview the eval-apply style in code (Section 4.1) and natural semantics (Section 4.2) using as a first example the call-by-value (or call-by-weak-normal-form) strategy of the pure lambda calculus. We discuss this strategy in detail, compare it to other call-by-value strategies, and justify why we can call it ‘call-by-value’ (Section 3). We use the Haskell 2010 standard as the implementation language, with all evaluators (plain and generic) written in monadic style to guarantee semantics (strategy) preservation. The reasons for choosing Haskell are discussed in Section 4.1, but the choice of implementation language is ultimately immaterial

We discuss in the survey strategies that are well-known, less known, and interesting novel ones. We recall their properties, their abstract machines, and some of their uses, particularly in the evaluation of general recursive functions, where we use simple and well-known Church numerals as data for examples. We focus on this particular use for some strategies because it illustrates their evaluation properties in relation to thunking [Ing61, DH92] and to vanilla and delimited continuation-passing style [HD94, BBD05]. We want to draw attention to the novel strategy discussed in contribution **C2**.

⁴A term is strongly-normalising when all its evaluation sequences converge on a normal form [Bar84].

The survey excludes history-informed, parallel, non-deterministic, and right-to-left⁵ strategies. The survey also excludes strategies that cannot be defined in the pure lambda calculus (*e.g.* call-by-need which requires sharing and memoisation). Finally, the survey excludes simulations of strategies that belong to other calculi (*e.g.* Section 3). The survey is nonetheless extensive and informs future work on other strategy spaces. Strategy equivalence must be studied under equivalent conditions: in the same calculus, and with either the same term representation or with the same order of redexes (strategy) under different term representations.

- C2 Introduction of ‘spine applicative order’.** We introduce this strategy in the survey which evaluates general recursive functions in delimited continuation-passing style, *eagerly* but with a *non-strict* fixed-point combinator, and with thunking ‘*protecting by variable*’ instead of the common ‘protecting by lambda’ in the literature on thunking. This and the other strategies illustrate the diversity of the strategy space.
- C3 Characterisation of eval-readback style.** We overview the eval-readback style in code (Section 6.1) and natural semantics (Section 6.2) using well-known examples from the literature, and relate them to the eval-apply evaluators in the survey. We characterise the style precisely by means of two well-formedness provisos (Section 6.3), and discuss the limit on the amount of evaluation that can be moved between eval and readback, which determines equivalences within the style (Section 6.4).
- C4 Generic templates and evaluators.** We present the generic eval-apply and the generic readback evaluators by means of natural semantics templates and higher-order parametric evaluator implementations in code. The generic evaluators deliver plain evaluators as fixed-points by instantiation. The generic evaluators are obtained by parametrising on all the *variability points* of their respective style. The generic eval-apply evaluator delivers all the evaluators in the survey. The generic readback evaluator delivers all the readbacks that satisfy the provisos. The generic evaluators let us express the evaluator space in a single definition. We employ them in the equivalence proof by LWF.
- C5 Improved definition of uniform/hybrid strategy.** We improve the formal definition of uniform vs hybrid strategy that we presented in [GPN14, Sec. 4] and that we use in this paper to structure the strategy space in contribution **C6**. Intuitively, a hybrid (or layered, or stratified) strategy inextricably *depends* on other less-reducing subsidiary strategies to evaluate particular subterms such as operators or operands. When there is no dependency, the strategy is uniform. The dependency shows in the evaluation sequences which include whole evaluation sequences of the subsidiaries for those subterms. The dependency also usually shows in the *definitional device* used for defining the hybrid strategy (code or operational semantics, whether big-step, small-step, or abstract machines) which explicitly includes the definitional device of one or more subsidiary strategies. In this latter case we say the hybrid *strategy* is defined by a definitional device in hybrid *style*. For instance, a hybrid strategy’s big-step evaluator is in hybrid style when it calls a subsidiary strategy’s big-step evaluator. As explained in [GPN14], and as we elaborate in Section 7.2, the distinction is essential because, depending on the type of device, it may be possible to write uniform-style devices for hybrid strategies by hiding the subsidiary devices, or to write hybrid-style devices for uniform strategies by concocting spurious subsidiary devices.

⁵A right-to-left strategy evaluates the operand of a redex before the abstraction when it is non-weak, and the operands of a neutral from right-to-left when it is non-head.

As we detail in Sections 5.8 and 5.9, the uniform/hybrid notion was introduced by [Ses02] in the context of big-step evaluators in natural semantics, to respectively indicate the in/dependence on subsidiary big-step evaluators—what we call *style*. We relied on the style concept in [GPNG10, GPN13, GPNM13, GPNS14] and distinguished it from the style-independent property of a strategy in [GPN14], redundantly referring to a uniform/hybrid strategy as ‘uniform/hybrid in its nature’. Hereafter, we use ‘style’ for definitional devices and use ‘uniform/hybrid’ by itself for a property of a strategy.

In [GPN14] we formalised uniform/hybrid strategy using small-step context-based reduction semantics [Fel87, FH92, FFF09]. Since we wish to improve on [GPN14], in Section 7.2 we switch temporarily to that small-step definitional device and arrive at an improved definition from illustrative examples, rather than define the concept first and then illustrate it with examples.

We introduce in this paper the concept of a *balanced* (opposite, *unbalanced*) hybrid strategy (Definition 7.4). This concept is required for the equivalence proof by LWF. A hybrid strategy always uses a subsidiary to evaluate the operator M in applications MN to obtain a redex $(\lambda x.B)N$. A balanced hybrid is either a non-strict hybrid (does not evaluate N) or is a strict hybrid that uses the subsidiary to evaluate N .

C6 Structuring of the strategy space. We structure the strategy space using the uniform/hybrid dichotomy as an orthogonal criterion, and structure the uniform and hybrid subspaces according to the weakness, strictness, and headness properties of strategies considered as a triple.

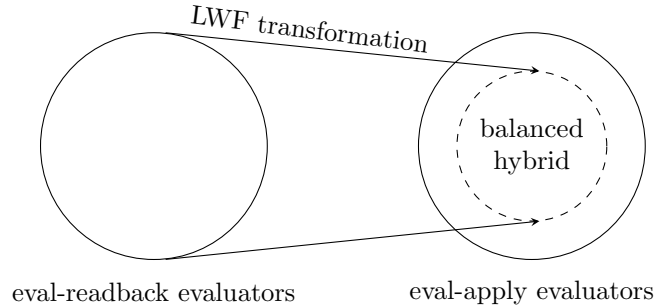
The uniform strategy space makes up a lattice of eight strategies we call *Gibbons’s Beta Cube*.⁶ The cube contains the well-known uniform strategies (call-by-value, call-by-name, applicative order, head spine) and four novel strategies. We introduce the well-formedness provisos that characterise uniform eval-apply evaluators and show that they define uniform strategies. We also introduce a systematic notation for encoding uniform evaluators in a triple, and study the property of absorption among uniform strategies.

The hybrid space is larger. We introduce the well-formedness provisos that characterise hybrid eval-apply evaluators and show they are required to define hybrid strategies. The hybrid evaluators rely on *one* uniform subsidiary evaluator to satisfy the desired properties of the strategy (completeness, spininess, etc.). In particular, the subsidiary is always used to evaluate operators. The hybrid evaluator evaluates more redexes than the subsidiary evaluator. We introduce a systematic notation for encoding hybrid evaluators by means of two triples: the Beta Cube triple of the uniform evaluator used as subsidiary by the hybrid evaluator, and the triple encoding the hybrid’s weakness, strictness, and headness. We use the notation to introduce novel strategies and discuss (non-)equivalences among hybrid evaluators.

C7 Equivalence proof by LWF. Finally, we present a proof of equivalence between eval-readback evaluators and *balanced hybrid* eval-apply evaluators. The proof assumes the provisos for both styles and proceeds by application of the LWF transformation. We start from a *plain but arbitrary* eval-readback evaluator $\mathbf{rb} \circ \mathbf{ev}$ where \mathbf{ev} is a uniform evaluator defined as a fixed-point of the generic eval-apply evaluator, and \mathbf{rb} is defined as a fixed-point of the generic readback evaluator. Some parameters of the generic

⁶We name the cube with permission after Jeremy Gibbons who suggested to us the use of booleans to construct a cube lattice (Section 7.3). Other unrelated named cubes in the lambda calculus literature are J. J. Lévy’s cube of residuals [Lév78] and H. Barendregt’s ‘Lambda Cube’ of typed lambda calculi [Bar91].

evaluators are type-checked but undefined and thus *arbitrary*. The LWF transformation fuses rb and ev into the plain but arbitrary balanced hybrid eval-apply evaluator hy that relies on a uniform subsidiary evaluator su , where both hy and su are fixed-points of the generic eval-apply evaluator. The type-checked but undefined parameters to the generic evaluators have their substitutable values constrained by the provisos and the LWF steps. The possible substitutions are collected in equations. In particular, some of the equations state that $\text{su} = \text{ev}$. When the equations hold then one-step equivalence holds and thus $\text{rb} \circ \text{ev} = \text{hy}$.



An expected corollary of the proof is that hy absorbs su/ev , that is, $\text{hy} \circ \text{su} = \text{hy}$. Absorption can be used to fuse an evaluator composition or to defuse an evaluator into a composition.

The LWF proof does not work for *unbalanced* hybrid strategies. These can be defined in eval-apply style but not in an LWF-equivalent eval-readback style. Recall from contribution **C5** above that an unbalanced hybrid hy is such that its subsidiary su evaluates applications that evaluate to a redex more than hy . Thus, hy cannot absorb su . It might be possible to obtain an equivalence between an eval-readback evaluator and an unbalanced hybrid eval-apply evaluator, but LWF cannot find that equivalence.

For a few strategies the equivalence by LWF holds *modulo commuting redexes* due to the particular steps of LWF. Commuting (*i.e.* non-overlapping) redexes occur in the operands N_i of a neutral $xN_1 \cdots N_n$. An equivalence modulo commuting redexes means the strategies are big-step equivalent but differ on the point of divergence, *i.e.* one strategy evaluates N_j first and diverges and the other evaluates N_k first and diverges, with $j < k$.

C8 Summary and discussion of (non-)equivalences. We provide an extensive summary of strategies, evaluators, and equivalences using the systematic notation, and also discuss the relevance of our contributions in the wider context of correspondences among operational semantics devices (Section 9).

These contributions are listed in increasing order of importance, but all are necessary and interdependent, and justify the content and length of the paper. The LWF proof goes through thanks to the generic evaluators and natural semantics templates. The latter are obtained after structuring the strategy space according to the improved definition of uniform/hybrid strategy. And the strategy space is presented in a survey which justifies the importance of each strategy and shows the scope of the equivalence proof. The LWF proof confirms the structuring is adequate. We also provide an extensive summary and discuss the relevance of the results in the wider context of operational semantics.

The paper is divided into four main sections (Sections 4 to 8) which can be read at separate times before the conclusions, and the related and future work sections. The survey

(Section 5) can be read cursorily and its details fetched on demand when reading about the structuring (Section 7). The equivalence proof by LWF (Section 8) can be followed with no previous knowledge of LWF, but we provide an overview of LWF in Appendix C. The code has been type-checked and tested, and can be obtained on request.

We have addressed the paper to non-expert readers which may be unfamiliar with some background material and thus we are explanatory in the text (or in the appendices when appropriate) for information or reminder. Expert readers can jump directly to the sections of their interest, with particular attention to Sections 7 to 9.

2. TECHNICAL PRELIMINARIES

We overview the pure lambda calculus notation and concepts required for this paper in Appendix A, namely, syntax and meta-notation of terms, reduction, evaluation sequence, evaluation strategy, completeness, and ‘completeness for’ a final result other than normal form. For the most part we adhere to the standard references [CF58, Bar84, HS08]. We use the notation for capture-avoiding substitution of [Plo75] (and justify why in Appendix A). We use grammars in Extended Backus-Naur Form to define sets of terms. Figure 1 lists the main sets of terms we use in this paper, which play a special role in evaluation. They are proper subsets of the set Λ of pure lambda calculus terms.

Λ	$::= V \mid \lambda V.\Lambda \mid \Lambda \Lambda$	Terms over a set of variables V .
Neu	$::= V \Lambda \{\Lambda\}^*$	Neutrals: applications $x N_1 \cdots N_n$ where $n \geq 1$.
NF	$::= \lambda V.\text{NF} \quad \mid \quad V \{\text{NF}\}^*$	Normal Forms.
WNF	$::= \lambda V.\Lambda \quad \mid \quad V \{\text{WNF}\}^*$	Weak Normal Forms.
HNF	$::= \lambda V.\text{HNF} \quad \mid \quad V \{\Lambda\}^*$	Head Normal Forms.
WHNF	$::= \lambda V.\Lambda \quad \mid \quad V \{\Lambda\}^*$	Weak Head Normal Forms.
VHNF	$::= \lambda V.\text{VHNF} \quad \mid \quad V \{\text{WNF}\}^*$	Value Head Normal Forms.

Figure 1: Main sets of terms in Extended BNF where $\{X\}^*$ denotes zero or more occurrences of non-terminal X , that is, $X_1 \cdots X_n$ with $n \geq 0$.

Appendix C provides an overview of LWF. Appendix D provides an overview of Haskell’s operational semantics in relation to LWF. Appendix E explains why strict strategies are incomplete and when non-strict strategies are complete, particularly the leftmost and spine strategies discussed in the survey. For the moment we assume the intuitive definition of uniform/hybrid strategy and of un/balanced hybrid strategy provided in contribution **C5**. (Alternatively, the reader may assume the formal definition of uniform/hybrid strategy in [GPN14].) We also assume that all the uniform-style and hybrid-style eval-apply evaluators in the survey respectively define uniform and hybrid strategies. The validity of this assumption will be proven in Section 7 where we give the improved formal definition of uniform/hybrid strategy. The definition of one-step equivalence ‘modulo commuting redexes’ has been introduced in contribution **C7**.

3. CALL-BY-VALUE POLICY, STRATEGIES, AND CALCULI

To discuss an evaluator style we must first pick some strategy. In this section we define a strategy in words to avoid the circularity of using an evaluator to define the strategy.

Definition of call-by-WNF. We pick the ‘call-by-weak-normal-form’ strategy that evaluates terms to weak normal form (WNF in Figure 1) by choosing the leftmost-innermost redex not inside an abstraction. Recall that WNF consists of arbitrary (unevaluated) abstractions, variables, and neutrals with operands in WNF.

The strategy is weak (does not evaluate abstractions), strict (evaluates operands of redexes to WNF), and non-head (evaluates operands of neutrals to WNF). Like all strategies, it is an identity on variables. It is also an identity on abstractions because it is weak. And it evaluates applications MN by first evaluating M :

- If M evaluates to an abstraction $\lambda x.B$, then the application $(\lambda x.B)N$ is a redex. The strategy evaluates the operand N to N' and then evaluates the contractum $[N'/x](B)$.
- If M does not evaluate to an abstraction, then it either evaluates to a variable x or to a neutral M' . In both cases, the application $(xN$ or $M'N)$ is a neutral and the strategy evaluates the operand N' as if the head variable (either x or the head variable of M') were a strict data-type value constructor.

The strategy is incomplete for WNF (and therefore for NF) because it is strict (Appendix E). For example, it diverges on $(\lambda x.y)\Omega$ whereas other strategies converge on y which is a WNF (and also a NF).

Note that the strategy contracts redexes when the operand is a neutral in WNF. For example, given a redex $(\lambda x.B)(zN)$, if the evaluation of N converges on N' , then the strategy evaluates the contractum $[(zN')/x](B)$, in accordance with the beta-rule of the pure lambda calculus which is unrestricted about what can be the operand of a redex.

Call-by-WNF and the call-by-value policy. The name ‘call-by-value’ refers to the parameter-passing *policy* in programming languages for applications of functions and value constructors to operands. The policy decrees that functions are not evaluated, and that operands are evaluated to a ‘value’ whose definition depends on the programming language. The value is then passed to the function or attached to the value constructor. In the pure lambda calculus the notion of ‘value’ cannot be the expected one of normal form because the abstractions within the operand would be evaluated to normal form, but abstractions (functions) must not be evaluated. (The strategy that embodies call-by-normal-form is ‘applicative order’ discussed in the survey.)

In the pure lambda calculus the call-by-value strategy is call-by-WNF which is weak, strict, and non-head. However, this strategy does not uphold *operational equivalence* when terms are open (concretely, when redexes have neutrals in WNF as operands) *and also* divergent. Operational equivalence is also called ‘contextual equivalence’ because it is defined conveniently using closing contexts, namely, terms with a placeholder hole for placing another term and closing all its free variables (see Section 7.2.1 for a precise definition of context). Two terms M and N are operationally equivalent when in any closing context \mathcal{C} then $\mathcal{C}(M)$ and $\mathcal{C}(N)$ both converge to the same result or both diverge. Operational equivalence is expected when one term evaluates to the other. However, take the redex $R \equiv (\lambda x.\lambda y.y)(zz)$, which is an instantiation of the aforementioned redex form $(\lambda x.B)(zN)$. Call-by-WNF evaluates R to the normal form $\lambda y.y$ because the neutral (zz) is a WNF. But R and $\lambda y.y$ are not operationally equivalent under call-by-WNF: with the context

$\mathcal{C} = (\lambda z. \square)(\lambda x. xx)$, the call-by-WNF evaluation of $\mathcal{C}(R)$ diverges whereas the evaluation of $\mathcal{C}(\lambda y. y)$ converges.⁷

A statement of the problem, its explanation and a solution, were notoriously introduced in [Plo75]. The explanation is that the neutral must be *given the opportunity to diverge before contraction*, as is the case in $\mathcal{C}(R)$. The solution is to block the contraction in case the head variable might be eventually substituted, and thus to restrict the notion of ‘value’ to variables and abstractions so that R is no longer a redex.

The problem, however, is not with the strategy itself but with the pure lambda calculus and its beta-reduction relation on which call-by-WNF is based. The blocking must occur in the beta rule for a beta-reduction with blocking to be confluent. The confluent and blocking ‘lambda-value’ calculus was proposed where evaluation to a value upholds operational equivalence [Plo75, pp. 135–136, 142–145]. The lambda-value calculus is the basis of other improved call-by-value calculi [FF86, Mog91, Wad03, CH00, KMR21, AG22]. Readers interested in further technical observations about lambda-value are referred to Appendix B.

Call-by-WNF as a call-by-value strategy. In this paper we refer to call-by-WNF as ‘call-by-value’ for a formal and a pragmatic reason. The formal reason is that it is the call-by-value strategy of the *pure* lambda calculus. To study blocking strategies with open and divergent terms requires a lambda-value or related calculus with blocking in the beta rule. Proving evaluator equivalences for such strict strategies (and for the non-strict strategies that can be simulated in that calculi [Plo75, Sec. 6]) is future work (Section 11).

The pragmatic reason is that operational equivalence holds for *closed* terms under weak evaluation and for *convergent* terms under weak or full evaluation. As stated in Section 1.3, studying strategies and evaluators for arbitrary pure terms allows us to then restrict to closed and/or convergent subsets to include strategies and evaluators in programming and proof-assistant settings.

With closed terms and weak evaluation, neutrals are given the opportunity to diverge because evaluation does not go under lambda and every neutral is given the binding for its free head variable. (In applied calculi, applications $c N_1 \cdots N_n$ with value constructor c are permissible operands.) Many so-called ‘call-by-value’ evaluators and abstract machines in the literature assume closed input terms: the SECD abstract machine [Lan64] [Lan65] [FH88, Chp. 10] (its evaluator defined in [Lan64, p. 316]), [FF86] [FH88, Chp. 9] [Rea89, Chp. 13] [Mac90, Chps. 11–12] [Ses02, pp. 421, 427] [Pau96, p. 390] (named `eval`), and [PR99, p. 17] (named ‘inner machine’). These evaluators implement call-by-WNF for closed input terms. Even the lambda-value big-step evaluator ‘eval_V’ of [Plo75, p. 130] assumes closed input terms and implements call-by-WNF for closed input terms (Appendix B).

With convergent terms, there is no opportunity to diverge, terms can be open or closed, and evaluation under lambda and full-reduction are unproblematic. Convergent pure terms are obtainable by strong type-checking and type-erasure [Pie02, GL02].⁸ Many programming languages have weak evaluation and type systems based on System F [Car97, Pie02]. However, these type systems are sound for full-reduction [SR15, p. 3]. Call-by-WNF and its full-reducing relatives we discuss in the survey (Section 5) work verbatim with convergent terms and

⁷ $\mathcal{C}(R) = (\lambda z. (\lambda x. \lambda y. y)(zz))(\lambda x. xx)$ evaluates to $(\lambda x. \lambda y. y)\Omega$ which diverges. $\mathcal{C}(\lambda y. y) = (\lambda z. \lambda y. y)(\lambda x. xx)$ converges to $\lambda y. y$.

⁸Convergent terms are not exclusively strongly-normalising terms. A simply-typed lambda calculus extended with terminating structural recursion primitives has convergent terms because self-application does not type-check [Pie02].

uphold operational equivalence. In Sections 5.7 and 6 we discuss the case in point of the weak-reducing (and referred to as ‘call-by-value’) evaluator \mathcal{V} [GL02, p. 236] that implements call-by-WNF for convergent terms. We also study the full-reducing call-by-value eval-readback evaluator named ‘strong reduction’ (denoted by \mathcal{N}) [GL02, p. 237] that uses \mathcal{V} for its eval stage.⁹

4. THE EVAL-APPLY EVALUATOR STYLE

An evaluator in eval-apply style is defined by an `eval` function that recursively evaluates subterms. In its traditional definition, `eval` relies on a mutually-recursive local `apply` function for the specific evaluation of application terms. In modern functional programming languages the local `apply` can be obviated within `eval` using `let`-clauses and pattern-matching expressions.

As explained in Section 1.3, we use a deep embedding of lambda-calculus terms into Haskell where lambda-calculus variables are represented by strings.

```
data Term = Var String | Lam String Term | App Term Term
```

We implement capture-avoiding substitution directly (and thus inefficiently) according to the textbook definition [HS08]. The implementation details are unimportant.¹⁰

```
subst :: Term -> String -> Term -> Term
subst n x b = ... -- substitute n for x in b.
```

The following evaluator, named `bv`, implements in traditional eval-apply style the call-by-value (call-by-WNF) strategy described in words in Section 3.

```
bv :: Term -> Term
bv v@(Var _) = v
bv l@(Lam _ _) = l
bv (App m n) = apply (bv m) (bv n)
  where apply :: Term -> Term -> Term
        apply (Lam x b) n' = bv (subst n' x b)
        apply m' n' = App m' n'
```

The evaluator pattern-matches on the input term. For variables (first clause) and abstractions (second clause) it behaves as an identity. We use Haskell’s ‘as-patterns’ to simplify the code. For applications (third clause), the local `apply` function is called with recursive calls to `bv` on the operator and the operand. The local `apply` pattern-matches on the operator to evaluate it. Its first clause evaluates the redex and its second clause evaluates the neutral. The local `apply` can be obviated within `eval` using Haskell’s `let` and `case` expressions:

```
bv :: Term -> Term
bv v@(Var _) = v
bv l@(Lam _ _) = l
bv (App m n) = let m' = bv m in case m' of
```

⁹To be precise, \mathcal{V} evaluates operands of neutrals right-to-left, so the equivalence with call-by-WNF is modulo commuting redexes. The result delivered by \mathcal{V} is a WNF, not a HNF as stated in [GL02, p. 236]. Both \mathcal{V} and \mathcal{N} are proven correct for strongly-normalising terms [GL02, Lems. 5 & 6, pp. 237–9].

¹⁰For interested readers, our string variables have trailing numbers (e.g. “x1”, “y5”, etc.). Function `subst` pattern-matches on its third argument and obtains fresh variables by incrementing the largest trailing number of all the string variables of all its inputs.


```
(Lam x b) -> let n' = bv n in bv (subst n' x b)
_         -> let n' = bv n in App m' n'
```

The two `let` expressions for the operand `n'` could be simplified or omitted (substituting `bv n` for `n'`) but they will prove handy when we compare evaluators and generalise in Section 7.

Readers familiar with Haskell will notice that `bv` actually does *not* implement call-by-value due to the classic ‘semantics-preservation’ problem of evaluators [Rey72]: the evaluation order of function and value-constructor application (non-strict call-by-need) in the implementation language (Haskell) does not match the intended evaluation order of function and term application (strict) for the object language (pure lambda calculus). Concretely, the operand `n'` (bound to the call `bv n` in the `let` expression) is passed *unevaluated* by Haskell to `subst`, which evaluates `n'` only if `x` occurs in `b`. Thus, for example, if we pass to `bv` the input term $(\lambda x.z)\Omega$ as an expression of type `Term`, then the evaluator wrongly delivers `z` (as `Var "z"`).

In a strict implementation language, such as Standard ML or OCaml, where function and value-constructor application are strict, the `n'` parameter would be evaluated before being passed to `subst`. Such strict evaluation would also work fine with evaluators that implement non-strict strategies such as, say, call-by-name (Section 5.2), because in that case `n'` would be bound to a term (a data-type value) rather than to a call expression like `bv n`.

The problem with the strictness of `subst` is compounded by the *partially-defined* values of type `Term`. This type represents terms of Λ with the addition of the totally-undefined value \perp and of partially-defined values such as `Lam "x" \perp` , `App \perp \perp` , etc. The totally-undefined value is necessary because we want to consider non-terminating strategies. It is the case that every evaluator `ev` is strict, that is, `ev \perp = \perp` , because it pattern-matches on its input term. It is also the case that an evaluator may not terminate for an input term `t`, namely, `ev t = \perp` . However, partially-defined values have no counterpart in Λ . We must avoid `ev t = p` for a partially-defined term `p`. This can be done using strict function application during evaluation.

4.1. Monadic style eval-apply. Strict function application can be simulated in Haskell using the principled and flexible monadic style. We use the Haskell 2010 standard which has a well-defined static and operational semantics (Appendix D). Evaluators are one of the motivating applications of monads [Wad92b, Wad92a], and monadic evaluators can be program-transformed to other operational semantics devices [ADM05]. Monads are provided in the Haskell 2010 library by means of the `Monad` type class and its datatype instances. Type classes have a trivial dictionary operational semantics [HHPW96]. The `Monad` type class provides operators for monadic identity (`return`), monadic function application (`=<<`), and monadic function composition (`<=<`), called Kleisli composition. The first two operators are defined by programmers for the particular monad instance. Monadic function composition is defined in the library in terms of monadic function application. For comparison, we show in Figure 2 the type signatures alongside ordinary identity, function application, and function composition.

Strict function application can be simulated by monadic application [Wad92a, Sec. 3.2] when `return \perp = \perp` and `f =<< \perp = \perp` for function `f`, which is the case for some strict monads.¹¹ In particular, we will use the `IO` monad because we are interested in printing

¹¹The statement holds in Haskell 2010. However, at the time of writing, the latest `Monad` class is a subclass of `Functor` and `Applicative` which are not in the 2010 standard, and the `Functor` laws use non-strict

```

-- Identity
id    ::          a ->  a
return :: Monad m => a -> m a

-- Application
($)   ::          (a ->  b) ->  a ->  b
(=<<) :: Monad m => (a -> m b) -> m a -> m b

-- Composition
(.)   ::          (b ->  c) -> (a ->  b) -> (a ->  c)
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)

(g . f) x = let y = f x
              in g y

(g <=< f) x = do y <- f x
              g y

```

Figure 2: Type signatures of monadic operators alongside their ordinary counterparts. Monadic function composition is defined in terms of monadic function application. The `do`-notation is syntactic sugar for the latter.

the results of evaluation. (We will use monadic composition later on when we discuss eval-readback evaluators in Section 6).

The monadic style has several benefits. First, we can keep Haskell which is our preferred choice of implementation language for, among other things, its convenient syntax for writing the generic evaluators and their fixed-points (Sections 7.1.1 and 7.1.2). Second, we can keep **subst** non-monadic. Third, we can toggle between strict and non-strict Haskell evaluation by instantiating the evaluator’s type signature with strict and non-strict monad instances. The non-strict Haskell evaluation of strict strategies like **bv** can be helpful in iterative tracing up to particular depths [Ses02] to locate divergence points, known in vernacular as ‘chasing bottoms’ [DJ04]. The parametrisation on a monad type parameter requires the ‘arbitrary rank polymorphism’ type-checking extension of the Glorious Haskell Compiler (GHC) which falls outside the Haskell 2010 standard. Nevertheless, the extension has no run-time effect, as it simply allows GHC to overcome the limitations of the default type-inference algorithm by letting programmers provide type signature annotations for perfectly type-checkable code. Hereafter, we write evaluators in monadic style with a monad type parameter, but the parameter is optional and immaterial.

The transcription of the naive **bv** evaluator to monadic style is immediate with Haskell’s `do`-notation:

```

bv :: Monad m => Term -> m Term
-- may assume   Term -> IO Term
bv v@(Var _)   = return v
bv l@(Lam _ _) = return l

```

composition [Wad92a, Sec. 3.2]. To use Haskell in this setting we would have to define an alternative strict monadic composition.

```

bv (App m n) = do m' <- bv m
               case m' of
                 (Lam x b) -> do n' <- bv n
                               bv (subst n' x b)
                 _         -> do n' <- bv n
                               return (App m' n')

```

The type of `bv` is bounded-quantified on the monad type parameter `m`, which can be instantiated to a strict or non-strict monad instance using a type annotation when executing the evaluator, *e.g.* `(bv :: IO Term) t`. Readers uninterested in the parameter may assume `m` already instantiated to the `IO` monad, as shown in the comment. Notice that `m` also occurs as a value variable (operator `m`) on the third clause for applications. This is usual Haskell coding style, as type and value variables have independent scopes.

The use of Haskell, monadic style, and strict monad instances are not required for the proof by LWF. Haskell 2010 is used as a vehicle for communication and, at any rate, we could treat all Haskell code as syntactic sugar for a strict implementation language. LWF is a simple, syntactic, and non-invertible transformation that only requires evaluators to be strict on their first parameter (Appendix C), which is the case for every evaluator because the first parameter is the input term. LWF does not involve equational reasoning, although monadic equational reasoning in Haskell is perfectly possible, *e.g.* [GH11]. A LWF proof can also be obtained for non-monadic or monadic code in a strict implementation language, as demonstrated in [GPN14] for two plain evaluators.

4.2. Natural semantics style eval-apply. A natural semantics is a proof-theory of syntax-directed and deterministic (exhaustive and non-overlapping) inference rules [Kah87]. Every rule has zero or more antecedent formulas above a line which are ordered from left to right, and only one consequent formula shown below the line. Every rule states that the formula below is the case if all formulas above are the case. A rule with no formulas above is an axiom. A formula holds iff a stacking up of rules (an evaluation tree) ends in axioms.

An eval-apply evaluator in natural semantics consists of a set of inference rules that specify the evaluator's behaviour recursively on the subterms. The call-by-value evaluator `bv` below is defined in natural semantics by four rules: one for evaluating VARIABLES, one for evaluating ABSTRACTIONS, one for evaluating applications which unfold to a redex that is CONTRACTED, and one for evaluating applications which unfold to NEUTRALS.

$$\begin{array}{c}
\text{VAR} \frac{}{\text{bv}(x) = x} \qquad \text{ABS} \frac{}{\text{bv}(\lambda x.B) = \lambda x.B} \\
\text{CON} \frac{\text{bv}(M) = \lambda x.B \quad \text{bv}(N) = N' \quad \text{bv}([N'/x](B)) = B'}{\text{bv}(MN) = B'} \\
\text{NEU} \frac{\text{bv}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{bv}(N) = N'}{\text{bv}(MN) = M'N'}
\end{array}$$

Every formula of the form $\text{bv}(M) = N$ reads ‘ N is the result in the finite evaluation sequence of M under `bv`’. The formula is proven by the finite evaluation tree above it. When $\text{bv}(M)$ has an infinite evaluation sequence (`bv` diverges on M) then it has an infinite evaluation tree and we write $\text{bv}(M) = \perp$ to indicate it. (The symbol ‘ \perp ’ is meta-notation because $\perp \notin \Lambda$.)

Let us explain the rules. The VAR axiom states that `bv` is an identity on variables (they are normal forms). The ABS axiom states that `bv` is also an identity on abstractions (`bv` is

weak). The CON rule states that the application MN evaluates by \mathbf{bv} to a term B' if \mathbf{bv} evaluates the operator M to an abstraction $\lambda x.B$, and also evaluates the operand N to a term N' , and also evaluates the contractum of the redex $(\lambda x.B)N'$ to a result B' . The NEU rule states that the application MN evaluates by \mathbf{bv} to a neutral $M'N'$ if \mathbf{bv} evaluates the operator M to a non-abstraction M' (*i.e.* a variable or a neutral), and also evaluates the operand N to a result N' .

The eval-apply natural semantics corresponds with the eval-apply evaluator in code, where the CON and NEU rules are implemented by `apply`. In rule CON, the contractum's occurrence within the \mathbf{bv} function is a notational convenience. The contractum can be placed as a premise right before the last premise that evaluates it.

$$\text{CON} \frac{\begin{array}{c} \vdots \\ \hline \mathbf{bv}(M) = \lambda x.B \end{array} \quad \begin{array}{c} \vdots \\ \hline \mathbf{bv}(N) = N' \end{array} \quad \begin{array}{c} \vdots \\ \hline [N'/x](B) \equiv B'' \end{array} \quad \begin{array}{c} \vdots \\ \hline \mathbf{bv}(B'') = B' \end{array}}{\mathbf{bv}(MN) = B'}$$

With this modification it is easier to illustrate the connection between an evaluation tree and an evaluation sequence. The latter can be obtained by the in-order traversal of the CON rules in an evaluation tree. The in-order traversal of the CON rule is as follows: in-order traversal of the stacked evaluation tree (vertical dots) above the first premise and then above the second premise, followed by the contractum premise in the middle, followed by the in-order traversal of the stacked evaluation tree above the last premise. Figure 3 illustrates with an example.

Derivation tree:

$$\mathbf{bv}(\lambda x.\mathbf{I}z) \equiv \lambda x.\mathbf{I}z \quad \frac{\mathbf{bv}(\mathbf{I}) = \mathbf{I} \quad \mathbf{bv}(z) = z \quad \boxed{[z/x](x)}^1 \equiv z \quad \mathbf{bv}(z) = z}{\mathbf{bv}(\mathbf{I}z) = z} \quad \boxed{[z/x](\mathbf{I}z)}^2 \equiv \mathbf{I}z \quad \frac{\dots \boxed{[z/x](x)}^3 \dots}{\mathbf{bv}(\mathbf{I}z) = z}}{\mathbf{bv}((\lambda x.\mathbf{I}z)(\mathbf{I}z)) = z}$$

Evaluation sequence:

$$(\lambda x.\mathbf{I}z)(\underline{\mathbf{I}z}) \rightarrow (\lambda x.\mathbf{I}z)(\underline{\boxed{[z/x](x)}}^1) \equiv (\lambda x.\underline{\mathbf{I}z})z \rightarrow \underline{\boxed{[z/x](\mathbf{I}z)}}^2 \equiv \underline{\mathbf{I}z} \rightarrow \underline{\boxed{[z/x](x)}}^3 \equiv z$$

Figure 3: Fragment of the evaluation tree of the term $(\lambda x.\mathbf{I}z)(\mathbf{I}z)$ where $\mathbf{I} \equiv \lambda x.x$. The evaluation sequence (with redexes underlined) is obtained by collecting the redexes and contracta in in-order traversal.

5. A SURVEY OF THE PURE LAMBDA CALCULUS'S STRATEGY SPACE

This section provides a survey of pure lambda calculus strategies, their properties (some of which are features of their definition and serve as classification criteria), their abstract machines, and some uses that illustrate the properties. The strategies are defined through their canonical eval-apply evaluators in natural semantics. For ease of reference, all the evaluators are collected in two figures. Figure 4 (page 21) shows the evaluators for the main well-known strategies in the literature. Figure 5 (page 27) shows the evaluators for less known but equally important strategies. The last three are novel and of particular interest. We discuss each evaluator/strategy in a separate section. With the exception of the novel strategies, the details discussed in the survey are known and can be found in the cited literature.

The survey can be read cursorily and particulars fetched on demand when reading about the structuring of the strategy space (Section 7). The role of the survey is to illustrate a varied strategy space that is nonetheless determined by a few variability points. The survey is thus wide in strategies but not too deep in strategy details beyond those needed to structure the space. A summary can be found at the end in Section 5.14.

All evaluators are defined in natural semantics by four rules named VAR, ABS, CON and NEU which have been explained in Section 4.2. We omit the VAR rule in the figures because every evaluator is an identity on variables. We also omit the rule names because we always show them in order (ABS, CON, and NEU) from left to right, top to bottom. The CON and NEU rules are related by their first premises. In all the CON rules the first premise has the form $\text{ea}(M) = \lambda x.B$ (here ea is a generic name for *eval-apply* evaluator) which states that the operator M evaluates to an abstraction. In all the NEU rules the first and second premises respectively have the form $\text{ea}(M) = M'$ and $M' \neq \lambda x.B$ which state that the operator M does not evaluate to an abstraction.

The first six evaluators in Figure 4 and the first two in Figure 5 are discussed in [Ses02] which classifies strategies according to three criteria: weakness, strictness, and kind of final result. The present survey can be seen as an extension of that work where we add more strategies, properties, classification criteria, and background. In particular, we indicate whether the evaluators are in uniform or hybrid *style*, and whether they are *balanced* hybrid (recall the latter’s definition from contribution **C5**). In Section 7 we will confirm that all the uniform-style and hybrid-style evaluators in the survey respectively define uniform and hybrid strategies.

5.1. Call-by-value (bv). We have already discussed this strategy and its *eval-apply* evaluator **bv** in Sections 3 and 4. The strategy is call-by-value where ‘value’ means ‘weak normal form’ (call-by-WNF). It chooses the leftmost-innermost redex not inside an abstraction. It is weak, strict, non-head, and incomplete (because it is strict, Appendix E). It delivers weak normal forms as results. The call-by-value evaluator (**bv**) is defined in uniform style. The strategy is uniform.

As discussed in Section 3, the SECD abstract machine [Lan64] [FH88, Chp. 10] implements an applied version of call-by-WNF when input terms are closed [Ses02, p 421]. The CAM abstract machine [CCM84] [FH88, Chp. 13] ultimately implements call-by-WNF like SECD. The ZAM abstract machine of the ZINC project [Ler91] implements the weak evaluator \mathcal{V} mentioned in Section 3, which evaluates neutral operands right-to-left and is thus one-step equivalent to **bv** modulo commuting redexes. The CEK abstract machine [FF86, FH92] is related to the call-by-value strategy defined by the ‘ $\text{eval}_{\mathcal{V}}$ ’ big-step evaluator of the lambda-value calculus [Plo75] (Appendix B). However, as pointed out in [Ses02, p 421], the call-by-value strategy described in [FH92, Sec. 2] does not evaluate $(xy)((\lambda z.z)v)$ with value v to $(xy)v$ because it has no evaluation context for it.

The seminal framework for environment-based call-by-value abstract machines for a calculus of closures can be found in [Cur91, BD07]. In [BD07], the name ‘call-by-value’ is reserved for the abstract machine whereas the strategy is called ‘applicative-order’. However, the name ‘applicative order’ is usually reserved for the full-reducing call-by-NF strategy of the pure lambda calculus (Section 5.3). Traditionally, environment-based abstract machines correspond with calculi with closures or explicit substitutions [Cur91, ACCL91]. In [ABM14], the authors use the linear substitution calculus to vanish some machine transitions and

Call-by-value (bv) or call-by-WNF:

$$\frac{}{\text{bv}(\lambda x.B) = \lambda x.B} \quad \frac{\text{bv}(M) = \lambda x.B \quad \text{bv}(N) = N' \quad \text{bv}([N'/x](B)) = B'}{\text{bv}(MN) = B'}$$

$$\frac{\text{bv}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{bv}(N) = N'}{\text{bv}(MN) = M'N'}$$

Call-by-name (bn):

$$\frac{}{\text{bn}(\lambda x.B) = \lambda x.B} \quad \frac{\text{bn}(M) = \lambda x.B \quad \text{bn}([N/x](B)) = B'}{\text{bn}(MN) = B'}$$

$$\frac{\text{bn}(M) = M' \quad M' \not\equiv \lambda x.B}{\text{bn}(MN) = M'N}$$

Applicative order (ao) or call-by-NF:

$$\frac{\text{ao}(B) = B'}{\text{ao}(\lambda x.B) = \lambda x.B'}$$

$$\frac{\text{ao}(M) = \lambda x.B \quad \text{ao}(N) = N' \quad \text{ao}([N'/x](B)) = B'}{\text{ao}(MN) = B'}$$

$$\frac{\text{ao}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{ao}(N) = N'}{\text{ao}(MN) = M'N'}$$

Normal order (no):

$$\frac{\text{no}(B) = B'}{\text{no}(\lambda x.B) = \lambda x.B'}$$

$$\frac{\text{bn}(M) = \lambda x.B \quad \text{no}([N/x](B)) = B'}{\text{no}(MN) = B'}$$

$$\frac{\text{bn}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{no}(M') = M'' \quad \text{no}(N) = N'}{\text{no}(MN) = M''N'}$$

Head reduction (hr):

$$\frac{\text{hr}(B) = B'}{\text{hr}(\lambda x.B) = \lambda x.B'}$$

$$\frac{\text{bn}(M) = \lambda x.B \quad \text{hr}([N/x](B)) = B'}{\text{hr}(MN) = B'}$$

$$\frac{\text{bn}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{hr}(M') = M''}{\text{hr}(MN) = M''N}$$

Head spine (he):

$$\frac{\text{he}(B) = B'}{\text{he}(\lambda x.B) = \lambda x.B'}$$

$$\frac{\text{he}(M) = \lambda x.B \quad \text{he}([N/x](B)) = B'}{\text{he}(MN) = B'}$$

$$\frac{\text{he}(M) = M' \quad M' \not\equiv \lambda x.B}{\text{he}(MN) = M'N}$$

Strict normalisation (sn):

$$\frac{\text{sn}(B) = B'}{\text{sn}(\lambda x.B) = \lambda x.B'}$$

$$\frac{\text{bv}(M) = \lambda x.B \quad \text{bv}(N) = N' \quad \text{sn}([N'/x](B)) = B'}{\text{sn}(MN) = B'}$$

$$\frac{\text{bv}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{sn}(M') = M'' \quad \text{sn}(N) = N'}{\text{sn}(MN) = M''N'}$$

Figure 4: Canonical eval-apply evaluators in natural semantics for the foremost strategies in the literature.

‘distill’ abstract machines from the calculus while preserving the time complexity of the execution.

5.2. Call-by-name (bn). This strategy chooses the leftmost-outermost redex not inside an abstraction. It is weak, non-strict, head, and delivers (and is complete for) weak head normal forms (WHNF in Figure 1). These consist of variables and non-evaluated abstractions and neutrals.

The ABS rule does not evaluate abstractions (weak). The CON rule evaluates the operator to an abstraction (goes left) and then the operand N is passed unevaluated (non-strict ‘by name’) to the substitution. The NEU rule also leaves the neutral’s operand N unevaluated (head), as if the head variable were a non-strict data-type value constructor. The call-by-name evaluator (bn) is defined in uniform style. The strategy is uniform.

The presence of rule NEU marks the difference with call-by-name in programming languages where open terms are disallowed. As pointed out in [Ses02, p. 425], it is well-known that, with closed input terms, call-by-name (bn) is identical to those strategies and to the big-step evaluator ‘Eval $_N$ ’ of [Plo75]. The latter does not consider neutrals. Interestingly, the small-step call-by-name evaluator ‘ \rightarrow_N ’ does evaluate neutrals [Plo75, p. 146, 4th rule] but to a limited extent, as also pointed out in [Ses02, p. 421]. For example, $x((\lambda z.z)v)$ evaluates to xv but $xy((\lambda z.z)v)$ is a stuck term. The ‘Eval $_N$ ’ and ‘ \rightarrow_N ’ evaluators are one-step equivalent (proven on paper [Plo75, Thm. 2, p. 146], not by program transformation) only for closed input terms.

Call-by-name is the basis for the call-by-need evaluation mechanism of non-strict functional programming languages (broadly, call-by-name with sharing and memoisation [FH88, Chp. 11]). The seminal framework for environment-based call-by-name abstract machines for a calculus of closures can be found in [Cur91, BD07]. The Krivine abstract machine (KAM) [Kri07] implements call-by-name. It has been the inspiration for abstract machines of proof-assistants that add capabilities for full-reduction on KAM. The KN abstract machine is a well-known full-reducing improvement of KAM (Section 5.4).

5.3. Applicative order (ao). This strategy chooses the leftmost-innermost redex (whether or not inside an abstraction). It is non-weak, strict, non-head, full-reducing, and incomplete. It delivers normal forms as results (NF in Figure 1).

The ABS rule evaluates abstractions (non-weak). The CON rule evaluates the operator to an abstraction in NF (goes left), then the operand to NF (strict), and then the contractum to NF. The NEU rule evaluates the operator to NF (goes left), and then the operand to NF (non-head). The applicative order evaluator (ao) is defined in uniform style. The strategy is uniform.

Because applicative order evaluates operands to normal form before contraction, it is sometimes referred to as ‘call-by-value’ where value means ‘normal form’ (call-by-NF). It should not be called ‘call-by-value’ in applied versions of the pure lambda calculus for modelling programming languages because functions are not evaluated and the corresponding strategy is call-by-value as call-by-WNF (Section 5.1). As a historical curiosity, applicative order is called ‘standard order’ in [FH88, Chp. 6] perhaps because evaluating operands before contraction was somehow considered standard. However, ‘standard’ has a technical meaning in reduction (Section 5.4) which does not apply to applicative order. An early abstract

machine for applicative order based on a modification of the SECD abstract machine can be found in [McG70].

Applicative order illustrates that weakness and strictness are separate classification criteria, and that the notion of ‘value’ is determined by the properties of the strategy that determine how operands of redexes are evaluated.

5.4. Normal order (no). This strategy chooses the leftmost-outermost redex. It is non-weak, non-strict, non-head, full-reducing, and delivers (and is complete for) normal forms (NF). In the lambda-calculus literature, a leftmost-outermost redex is simply called ‘leftmost’ and normal order is called ‘leftmost reduction’ [CF58, HS08, Bar84]. Note that call-by-name (bn) is another leftmost strategy, but does not evaluate leftmost redexes inside abstractions.

The ABS rule evaluates abstractions (non-weak). The CON rule is more sophisticated. Given an application MN , normal order evaluates the operator M first (goes left), but when the operator evaluates to an abstraction $\lambda x.B$ then the application $(\lambda x.B)N$ is the leftmost-outermost redex to be contracted. Normal order must not evaluate the abstraction body B . That would be going innermost, like applicative order. Normal order contracts the outermost redex $(\lambda x.B)N$ passing N unevaluated to the substitution (non-strict). To avoid the full evaluation of abstractions in operator position, normal order relies on the weak call-by-name strategy (bn) as subsidiary. Finally, the NEU rule evaluates the operator further to NF and also evaluates the operand (non-head) further to NF, as the aim is to deliver a final NF.

The normal order evaluator (no) is hybrid in style, and it is balanced, as it evaluates applications MN in rule CON before contraction as much as its subsidiary call-by-name (bn). The strategy is balanced hybrid.

Normal order is the ‘standard’ full-reducing and complete strategy of the pure lambda calculus, both in the colloquial sense (it is the *de facto* strategy used for full evaluation in the calculus) and in the technical sense (roughly, it does not evaluate a redex to the left of the one just contracted [Bar84, HS08]). Due to the possibility of non-termination, normal order is a semi-decision procedure for the ‘has-a-normal-form’ property of terms. Because normal order is non-weak, it is inaccurate to refer to it as call-by-name in applied versions of the pure lambda calculus for modelling programming languages where functions are unevaluated. To our knowledge, no programming language has normal order as its evaluation strategy. It would evaluate functions non-strictly and value constructors strictly (no partially-defined terms). The opposite, strict function evaluation with non-strict value constructors was implemented by the Hope programming language [FH88].

The abstract machine KN [Cré07] implements normal order in lockstep [GPN19]. KN is the full-reducing version of the KAM machine mentioned in Section 5.2. In [ABM15], the authors present a KN-based full-reducing abstract machine with a global environment (the Strong Milner Abstract Machine) using the ‘distillery’ approach from the linear substitution calculus mentioned in Section 5.1. The underlying strategy, ‘linear leftmost-outermost reduction’ is a refinement of normal order and an extension of linear head reduction [DR04].

As motivated in contribution **C1**, we illustrate normal order’s evaluation properties in the evaluation of general recursive functions on Church numerals. Because normal order is full-reducing and complete, it delivers Church numeral results in their original representation

as normal forms. For illustration we use the typical factorial example.¹² The boldface terms denote the Church-encoded intended arithmetic functions and constants.

$$\mathbf{F} \equiv \lambda f. \lambda n. \mathbf{Cond} (\mathbf{IsZero} \ n) \ \mathbf{One} (\mathbf{Mult} \ n \ (f \ (\mathbf{Pred} \ n)))$$

Given a Church numeral \mathbf{n} , the evaluation of $(\mathbf{Y} \ \mathbf{F} \ \mathbf{n})$ under `no` delivers the Church numeral for the factorial of \mathbf{n} . The term \mathbf{Y} is the standard non-strict fixed-point combinator (Appendix A). The weak-reducing strategies we have discussed can also be used but they deliver terms with unevaluated ‘recursive’ calls. Applicative order cannot be used because it evaluates every subterm fully to normal form, including the fixed-point combinators which have no normal form. Strict strategies require other combinators and encodings. We defer this discussion to Section 5.7.

5.5. Head reduction (hr). This strategy, defined in [Wad76, p. 504], chooses the head redex, *i.e.* the redex that occurs in head position (Appendix A). It is non-weak, non-strict, and head, and delivers (and is complete for) head normal forms (HNF in Figure 1). These are approximate normal forms where neutrals are unevaluated. The head redex is in left position and thus head reduction is also a ‘leftmost’ strategy, concretely, an approximation of leftmost normal order.

The ABS rule evaluates abstractions (non-weak). Like normal order, head reduction relies on weak call-by-name (`bn`) as subsidiary in rules `CON` and `NEU` to avoid the evaluation of abstractions in operator position. Unlike normal order, head reduction does not evaluate neutrals (head). Indeed, normal order and head reduction differ only in rule `NEU`. The difference can be expressed in jest: unlike head reduction, normal order lost is head in pursuit of the normal form. Like normal order (`no`), head reduction (`hr`) is a balanced hybrid evaluator and strategy.

Head reduction is a semi-decision procedure for the ‘useful-as-a-function’ property of terms called ‘solvability’ [Wad76, Bar84]. Solvable terms consist of terms that evaluate to `NF` as well as terms that do not evaluate to `NF` but may deliver a `NF` when used as functions on suitable operands.

5.6. Head spine (he). This strategy, defined in [BKKS87, p. 209], is the first *spine* strategy in the survey. Spine strategies are non-strict and go under lambda just enough to uphold completeness (Appendix E). They evaluate the term’s spine when depicting the term as a tree. They are big-step equivalent to a leftmost strategy but evaluate operators more eagerly to (and yet are complete for) HNF. Concretely, spine strategies evaluate operators to HNF whereas their big-step equivalent leftmost strategies evaluate operators to less evaluated WHNF. The leftmost counterpart of head spine (`he`) is head reduction (`hr`). Both evaluate only needed redexes in the same number of different steps.

Head spine is non-weak, non-strict, and head, and delivers (and is complete for) HNFs. The head spine evaluator (`he`) is uniform in style. The strategy is uniform. The classic eval-apply evaluator `headNF` in [Pau96, p. 390] literally implements the head spine `he` evaluator in code (Section 6).

The more eager evaluation of an abstraction operator to HNF may in some cases eliminate copies of the bound variable in the body and consequently require less copies of

¹²The factorial function is actually *primitive* recursive, but the discussion applies to any *general* recursive function.

the operand after contraction. The trivial example is $(\lambda x.(\lambda y.\lambda z.y) x x)N$. Head reduction copies the operand N twice to the two occurrences of x . Head spine evaluates the operator to $(\lambda x.x)$ and consequently copies N once. There are terms for which copying is the same, *e.g.* $(\lambda x.\mathbf{Y}x)N$. The cost of copying is implementation dependent (Section 5.14).

5.7. Strict normalisation (sn). This strategy chooses the leftmost-innermost redex but it evaluates operators to WNF using call-by-value (bv) as subsidiary. If the result is an abstraction (rule CON) then it evaluates the redex's operand to WNF also using call-by-value. The contractum is evaluated fully to normal form. If the result is not an abstraction (rule NEU) then it evaluates the neutral fully to normal form. Finally, it evaluates abstractions fully to normal form (rule ABS).

Strict normalisation is non-weak, strict, non-head, full-reducing, and incomplete for arbitrary terms. Like normal order (no), strict normalisation (sn) is a balanced hybrid evaluator and strategy.

We introduced this strategy and its natural semantics evaluator in [GPN14, pp. 195-6] but under the name `byValue` because it is equivalent to the classic eval-readback evaluator `byValue` in [Pau96, p. 390]. In turn, the `byValue` evaluator antedates and is identical to the ‘strong reduction’ (\mathcal{N}) eval-readback evaluator in [GL02, p. 237], save for the evaluation order of operands of neutrals (left-to-right in `byValue`, right-to-left in \mathcal{N}). We discuss and compare `byValue` and \mathcal{N} in Section 6. Recall from Section 3 that \mathcal{N} is complete for convergent terms. Thus, `byValue` and strict normalisation (sn) are also complete for convergent terms. The LWF proof (Section 8) will show that strict normalisation (sn) and `byValue` are one-step equivalent modulo commuting redexes.

Abstract machines for full-reducing call-by-value are scarce. The SECD [Lan64] [FH88, Chp. 10], CAM [CCM84] [FH88, Chp. 13] and ZAM [GL02] abstract machines are weak-reducing. An abstract machine for a full-reducing call-by-value strategy of the pure lambda calculus with right-to-left evaluation of operands of neutrals *and* of redexes is presented in [BBCD20], with its performance improved in [BCD21]. Interestingly, the abstract machine is obtained by deconstructing the normal order KN machine using program transformation and the concept of hybrid style. The machine uses a right-to-left weak call-by-value subsidiary. The relation with right-to-left versions of strict normalisation (sn), `byValue`, and \mathcal{N} is not explored.

A seminal full-reducing call-by-value strategy for the lambda-value calculus of [Plo75] is defined by the abstract machine obtained as a instance of the ‘principal reduction machine’ of [RP04, p. 70]. A related full-reducing call-by-value strategy, called ‘value normal order’, is presented in [GPN16, Sec. 7.1]. The latter differs from the former in that blocks are evaluated left-to-right.

Strict normalisation can be used to evaluate general recursive functions on Church numerals using the strict fixed-point combinator \mathbf{Z} and thinking ‘protecting by lambda’. Thinking [Ing61, DH92] consists in delaying the evaluation of subterms that have recursive calls which may diverge. These subterms are ‘serious’ in opposition to the ‘trivial’ subterms that converge [Rey72, p. 382]. The serious subterms are delayed by placing them inside an abstraction, hence ‘protecting by lambda’. To illustrate, take the following thunked definition of the factorial function:

$$\mathbf{F} \equiv \lambda f.\lambda n. \mathbf{Cond} (\mathbf{IsZero} n) (\lambda v. \mathbf{One}) (\lambda v. \mathbf{Mult} n (f (\mathbf{Pred} n)))$$

Given a Church numeral \mathbf{n} , the term $(\mathbf{ZF} \mathbf{n} \mathbf{T})$ evaluates under \mathbf{sn} to the Church numeral for the factorial of \mathbf{n} . An arbitrary closed term \mathbf{T} , *e.g.* the identity \mathbf{I} , can be passed to $\mathbf{unthunk}$. The serious subterm $(\mathbf{ZF})(\mathbf{Pred} \mathbf{n})$ is protected by the binding λv .

A continuation-passing style encoding is also possible using the identity as initial continuation:

$$\mathbf{F} \equiv \lambda f. \lambda n. \mathbf{Cond} (\mathbf{IsZero} \mathbf{n}) (\lambda k. k \mathbf{One}) (\lambda k. f (\mathbf{Pred} \mathbf{n}) (\lambda x. k (\mathbf{Mult} \mathbf{n} x)))$$

The serious subterm $(\mathbf{ZF})(\mathbf{Pred} \mathbf{n})(\lambda x. k (\mathbf{Mult} \mathbf{n} x))$ is protected by λk . At each step the passed continuation k is composed with the multiplication which is passed as a new continuation $\lambda x. k (\mathbf{Mult} \mathbf{n} x)$.

5.8. Hybrid normal order (hn). Hereafter we discuss the evaluators in Figure 5 (page 27). Hybrid normal order is non-weak, non-strict, non-head, full-reducing, and delivers (and is complete for) normal forms (NF). This strategy is presented in [Ses02, p. 429] and given its name because it ‘is a hybrid of head spine reduction and normal order reduction’. Normal order is itself hybrid, so ‘hybrid normal order’ can be confusing, but it is the original name given in [Ses02, p. 429]. Here ‘hybrid’ is used informally but closely to the concept of dependence on a subsidiary evaluator, what we call hybrid *style* (recall contribution **C5**). What the phrase means is that hybrid normal order (hn) is obtained from normal order (no) by replacing from the latter the subsidiary call-by-name (bn) by head spine (he). A more precise description is that hybrid normal order is the spine counterpart of leftmost normal order because it relies on a spine subsidiary to evaluate operators up to HNF, whereas normal order relies on a leftmost subsidiary to evaluate operators to WHNF (Section 5.6). Hybrid normal order (hn) is a balanced hybrid evaluator and strategy.

The eval-apply hybrid normal order evaluator (hn) is one-step equivalent to the classic eval-readback evaluator `byName` in [Pau96, p. 390] (Section 6). This is proven by the LWF proof (Section 8). Recall from Section 5.6 that hybrid normal order’s subsidiary, head spine (he), is literally implemented in code by `headNF` which is the eval stage of the eval-readback evaluator `byName` (Section 6).

5.9. Hybrid applicative order (ha). Hybrid applicative order is non-weak, strict, non-head, full-reducing, incomplete, and delivers normal forms (NF). This strategy is presented in [Ses02, p. 428] and given its name because it ‘is a hybrid of call-by-value and applicative order’. The word ‘hybrid’ is used again informally and closely to the concept of dependence on a subsidiary evaluator, what we call hybrid *style*. Hybrid applicative order (ha) relies on weak call-by-value (bv) as subsidiary to evaluate operators, just like normal order (no) relies on weak call-by-name (bn) as subsidiary for that. In short, hybrid applicative order (ha) differs from applicative order (ao) in that it does not evaluate abstractions in operator position. Although hybrid applicative order (ha) is incomplete, it can find more normal forms than applicative order (ao) because it is less eager (less undue divergence) and in fewer steps than normal order (no) because it is strict [Ses02].

Hybrid applicative order (ha) is a hybrid evaluator and strategy, *but it is unbalanced*: it evaluates operands of redexes *more* than its subsidiary `bv` because it calls itself recursively on the operand rather than calling `bv`. The assertion in [Ses02, p. 428] that hybrid applicative order is equivalent to the classic eval-readback `byValue` evaluator is thus incorrect: hybrid applicative order does not evaluate operands of redexes using `bv` (Section 6).

Hybrid normal order (hn):

$$\frac{\text{hn}(B) = B'}{\text{hn}(\lambda x.B) = \lambda x.B'} \qquad \frac{\text{he}(M) = \lambda x.B \quad \text{hn}([N/x](B)) = B'}{\text{hn}(MN) = B'}$$

$$\frac{\text{he}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{hn}(M') = M'' \quad \text{hn}(N) = N'}{\text{hn}(MN) = M''N'}$$

Hybrid applicative order (ha):

$$\frac{\text{ha}(B) = B'}{\text{ha}(\lambda x.B) = \lambda x.B'} \qquad \frac{\text{bv}(M) = \lambda x.B \quad \text{ha}(N) = N' \quad \text{ha}([N'/x](B)) = B'}{\text{ha}(MN) = B'}$$

$$\frac{\text{bv}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{ha}(M') = M'' \quad \text{ha}(N) = N'}{\text{ha}(MN) = M''N'}$$

Ahead machine (am):

$$\frac{\text{am}(B) = B'}{\text{am}(\lambda x.B) = \lambda x.B'} \qquad \frac{\text{bv}(M) = \lambda x.B \quad \text{bv}(N) = N' \quad \text{am}([N'/x](B)) = B'}{\text{am}(MN) = B'}$$

$$\frac{\text{bv}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{am}(M') = M'' \quad \text{bv}(N) = N'}{\text{am}(MN) = M''N'}$$

Head applicative order (ho):

$$\frac{\text{ho}(B) = B'}{\text{ho}(\lambda x.B) = \lambda x.B'} \qquad \frac{\text{ho}(M) = \lambda x.B \quad \text{ho}(N) = N' \quad \text{ho}([N'/x](B)) = B'}{\text{ho}(MN) = B'}$$

$$\frac{\text{ho}(M) = M' \quad M' \not\equiv \lambda x.B}{\text{ho}(MN) = M'N}$$

Spine applicative order (so):

$$\frac{\text{so}(B) = B'}{\text{so}(\lambda x.B) = \lambda x.B'} \qquad \frac{\text{ho}(M) = \lambda x.B \quad \text{so}(N) = N' \quad \text{so}([N'/x](B)) = B'}{\text{so}(MN) = B'}$$

$$\frac{\text{ho}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{so}(M') = M'' \quad \text{so}(N) = N'}{\text{so}(MN) = M''N'}$$

Balanced spine applicative order (bs):

$$\frac{\text{bs}(B) = B'}{\text{bs}(\lambda x.B) = \lambda x.B'} \qquad \frac{\text{ho}(M) = \lambda x.B \quad \text{ho}(N) = N' \quad \text{bs}([N'/x](B)) = B'}{\text{bs}(MN) = B'}$$

$$\frac{\text{ho}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{bs}(M') = M'' \quad \text{bs}(N) = N'}{\text{bs}(MN) = M''N'}$$

Figure 5: Canonical eval-apply evaluators in natural semantics for less known strategies.

Hybrid applicative order can be used to evaluate general recursive functions on natural numbers with the same fixed-point combinator and thunking style than strict normalisation (sn). But hybrid applicative order is more eager on operands.

5.10. Ahead machine (am). This strategy is defined in [PR99, Def. 5.1]. It uses call-by-value (bv) as subsidiary to evaluate operators and operands in redexes, like strict normalisation (sn), but it also uses call-by-value to evaluate neutrals. (Recall from Section 3 that in [PR99, p. 17] call-by-value (bv) is called ‘inner machine’.) Ahead machine evaluates abstractions using itself recursively.

Ahead machine is non-weak, strict, non-head, incomplete, and delivers so-called ‘value head normal forms’ (VHNF in Figure 1) which are terms in $WNF \cap HNF$. Ahead machine (am) is a balanced hybrid evaluator and strategy. Although it is incomplete in the pure lambda calculus, it is a semi-decision procedure for an approximate notion of solvability in the pure lambda-value calculus [PR99].

5.11. Head applicative order (ho). This is a novel strategy. It is the natural head variation of applicative order (ao). Its rule structure is similar to ao’s save for the NEU rule where neutrals are not evaluated. Head applicative order (ho) is non-weak, strict, head, incomplete, and delivers head normal forms (HNF). It is a uniform evaluator and strategy.

5.12. Spine applicative order (so). This is a novel strategy. It is non-weak, strict, non-head, full-reducing, and incomplete, and delivers NFs. It is the spine-ish counterpart of hybrid applicative order (ha). Recall from Section 5.6 that spine strategies, which live in the non-strict subspace, evaluate the term’s spine and operators to HNF. A spine strategy is big-step equivalent to a leftmost strategy that evaluates operators to WHNF. In the strict space we cannot speak of spineness literally, because evaluated operands are not in the term’s spine, but we think the spine analogy is pertinent when operators are evaluated to HNF by a strict subsidiary. Spine applicative order (so) evaluates operators to HNF using head applicative order (ho), whereas hybrid applicative order (ha) evaluates operators to WNF using call-by-value (bv). Spine applicative order (so) is a hybrid strategy *but it is unbalanced*: it evaluates operands of redexes *more* than its subsidiary by calling itself recursively.

Spine applicative order (so) is more eager than hybrid applicative order (ha). It has several interesting features. First, it is the *most eager* hybrid strategy in the survey. Second, it evaluates general recursive functions on Church numerals with thunking ‘protecting by variable’ (rather than ‘by lambda’), with delimited CPS (continuation-passing style) [BBD05], and with the *non-strict* fixed-point combinator **Y**. (Notice the contrast: ‘most eager’ and ‘non-strict fixed-point’.) To illustrate, take the delimited-CPS definition of the factorial function:

$$\mathbf{F} \equiv \lambda f.\lambda n.\lambda k. \mathbf{Cond} (\mathbf{IsZero} \ n) (k \ \mathbf{One}) (k (f (\mathbf{Pred} \ n) (\mathbf{Mult} \ n)))$$

Given a Church numeral **n**, the term $(\mathbf{Y} \ \mathbf{F} \ \mathbf{n} \ \mathbf{I})$ evaluates under **so** to the Church numeral for the factorial of **n**. The identity **I** is passed as the initial continuation. The variable *k*, which stands for the delimited continuation, protects the ‘serious’ subterm $(\mathbf{Y} \ \mathbf{F}) (\mathbf{Pred} \ n) (\mathbf{Mult} \ n)$. In the evaluation under **so** there are no administrative evaluation steps resulting from a strict fixed-point combinator and from the composition of continuations because they are used at each call, passing the multiplication $(\mathbf{Mult} \ n)$ as a continuation to each serious subterm.

5.13. Balanced spine applicative order (bs). This is a novel strategy. It is a balanced hybrid variation of spine applicative order (so) where the subsidiary, head applicative order (ho), is employed to evaluate operands of redexes to recover the balanced property. It is non-weak, strict, non-head, full-reducing, incomplete, and delivers NFs. Although it is less eager than spine applicative order (so), balanced spine applicative order (bs) can be used to evaluate general recursive functions in the same fashion as so.

5.14. Summary of the survey. Figure 6 (page 30) summarises the strategies, their properties, and their use in evaluating general recursive functions. Applicative order (ao) is missing from the bottom table because, as explained in Section 5.4, it evaluates every subterm fully to normal form, including the fixed-point combinators which have no normal form, and thus it defeats any thunking mechanism.

The four criteria of weakness, strictness, headness, and uniform/hybrid are the few variability points in the natural semantics. Their configurations determine the strategy space. The staged evaluation of the operator in rule NEU is not a variability point as it is subsumed by the uniform/hybrid configuration: only hybrid strategies use the staged evaluation because they use the subsidiary to evaluate operators.

Each strategy has its uses. As mentioned in Section 1.3, there is no optimal strategy in terms of time (*e.g.* contraction counts) or space (*e.g.* term size) because there are terms for which any strategy duplicates work [Lév78, Lév80, Lam89, AG98]. Optimisations are implementation dependent, particularly on term representation. For example, we implement evaluators in Haskell where we can share our deeply-embedded terms:

```
m :: Term
m = App n n
where n :: Term
      n = ... -- some large term
```

There is only a copy of `n` in `m`. The evaluation of `ev m` does not overwrite `m` because they are different Haskell expressions. The first may be memoised but the second remains intact. We have left efficiency considerations aside because the particulars constrain the strategy space.

Remark 5.1. We end this summary drawing the reader’s attention to a symmetry which may be related to the notion of duality [CH00]. Call-by-name (bn) and call-by-value (bv) are uniform and weak-reducing. These strategies are respectively used as subsidiaries by hybrid and full-reducing normal order (no) and strict normalisation (sn). Normal order uses call-by-name as subsidiary on operators. Strict normalisation uses call-by-value on operators and on operands. Call-by-name is dual to lambda-value’s call-by-value [Wad03]. We conjecture that, with convergent terms, call-by-value as call-to-WNF is dual to call-by-name in the weak-reducing space, and strict normalisation is dual to normal order in the full-reducing space.

6. THE EVAL-READBACK EVALUATOR STYLE

An evaluator in eval-readback style is defined as the composition of two functions: a less-reducing ‘eval’ function that contracts redexes to an intermediate result with no redex in outermost position, and a further-reducing ‘readback’ function that ‘reads back’ the

Strategies and properties							
bv	weak	strict	non-head	to WNF	incomplete		uniform
bn	weak	non-strict	head	to WHNF	complete-for	leftmost	uniform
ao	non-weak	strict	non-head	to NF	incomplete		uniform
no	non-weak	non-strict	non-head	to NF	complete-for	leftmost	hybrid bal.
hr	non-weak	non-strict	head	to HNF	complete-for	leftmost	hybrid bal.
he	non-weak	non-strict	head	to HNF	complete-for	spine	uniform
sn	non-weak	strict	non-head	to NF	incomplete		hybrid bal.
hn	non-weak	non-strict	non-head	to NF	complete-for	spine	hybrid bal.
ha	non-weak	strict	non-head	to NF	incomplete		hybrid
am	non-weak	strict	non-head	to VHNF	incomplete		hybrid bal.
ho	non-weak	strict	head	to HNF	incomplete		uniform
so	non-weak	strict	non-head	to NF	incomplete		hybrid
bs	non-weak	strict	non-head	to NF	incomplete		hybrid bal.

Big-step equivalence	
Leftmost	Spine
no	hn
hr	he

Usage in evaluating general recursive functions				
Strategy	Result	Combinator	Style	Thinking
bn	WHNF	Y	direct or CPS	none
hr, he	HNF	Y	direct or CPS	none
no, hn	NF	Y	direct or CPS	none
bv	WNF	Z	direct or CPS	protecting by lambda
am	VHNF	Z	direct or CPS	protecting by lambda
sn, ha	NF	Z	direct or CPS	protecting by lambda
ho	HNF	Y	delimited CPS	protecting by variable
so, bs	NF	Y	delimited CPS	protecting by variable

Figure 6: Summary of strategies, properties, and use in evaluating general recursive functions.

intermediate result to distribute ‘eval’ down the appropriate unevaluated subterms of the intermediate result to reach further redexes. Readback is a sort of ‘selective-iteration-of-eval’ function that calls eval directly or in an eval-readback composition on specific unevaluated subterms.

The style is related to the staged evaluation of normalisation-by-evaluation [BS91] but without recourse to an external domain. The separation of evaluation in eval and readback stages can be of help in writing evaluators. For example, the eval stage can be simply weak or non-head and written in uniform eval-apply style. The readback stage can then distribute eval under lambda, or within neutral operands, or both. The separation also enables independent optimisations, as mentioned in the introduction.

We illustrate the style using first the two classic eval-readback evaluators `byValue` and `byName` in [Pau96, p. 390]. We discuss more eval-readback evaluators later in the section. The `byValue` and `byName` evaluators are both full-reducing and deliver normal forms. The first is strict and incomplete. The second is non-strict and complete for NF. We first show and discuss them in their original non-monadic definitions in the strict Standard ML programming language, but adapted to our data-type `Term` and `subst` function (in

the originals, terms have strings only for free variables and de Bruijn indices for bound variables). We want readers to recognise the originals rather than show only the monadic Haskell versions (shown in Section 6.1). Furthermore, the original non-monadic definition of `byValue` can be compared directly with the also non-monadic definition of the ‘strong reduction’ evaluator [GL02, p.237] which we have claimed in Section 5.7 to be identical to `byValue` modulo commuting redexes.

The `byValue` evaluator is the composition of an eval evaluator, simply called `eval`, and a readback evaluator called `bodies`. The `eval` evaluator is literally the Standard ML version of the non-monadic call-by-value evaluator `bv` discussed in Section 5.1, which works fine in strict Standard ML.

```

fun eval (App(m,n)) =
  (case eval m of
     Lam(x,b) => eval (subst x (eval n) b)
  | m'       => App(m', eval n))
| eval t     = t

fun byValue t = bodies (eval t)
and bodies (Lam(x,b)) = Lam(x, byValue b)
  | bodies (App(m,n)) = App(bodies m, bodies n)
  | bodies t         = t

```

The readback evaluator `bodies` delivers a NF by fully evaluating the abstraction `bodies` (hence its name ‘`bodies`’) in the WNF delivered by `eval`. As the shape of a WNF indicates, the unevaluated abstractions occur at the top level or within the operands of neutrals.

In its first clause, `bodies` evaluates an abstraction to NF by calling the eval-readback composition (`byValue`) that recursively distributes `eval` down the body. In its second clause, `bodies` calls itself recursively on the operator of the neutral to distribute itself down nested applications, and on the operand to reach the unevaluated abstractions on which it will eventually call the eval-readback composition (`byValue`) of the first clause. Finally, `bodies` is the identity on variables. It has no clause for redexes because it is a readback and merely distributes evaluation.

As anticipated in Section 5.7, `byValue` is the precursor of the eval-readback ‘strong reduction’ evaluator \mathcal{N} in [GL02, p.237]. The latter only differs from the former in that neutrals have their operands flattened and evaluated right-to-left. This is illustrated by the following definition of \mathcal{N} that mirrors the original in [GL02, p.237] but with left-to-right evaluation. The W_i terms stand for WNFs.

$$\begin{aligned}
\mathcal{N} &= \mathcal{R} \circ \mathcal{V} \\
\mathcal{R}(\lambda x.B) &= \lambda x.\mathcal{N}(B) \\
\mathcal{R}(x W_1 \cdots W_n) &= x \mathcal{R}(W_1) \cdots \mathcal{R}(W_n) \quad n \geq 0
\end{aligned}$$

Substitute `byValue` for \mathcal{N} , `bodies` for \mathcal{R} , and `eval` for \mathcal{V} , and the result is `byValue`.

As discussed in Sections 5.1 and 5.7, strong reduction is incomplete for arbitrary terms but complete for convergent terms, as would any strategy in that setting.

The `byName` evaluator is the composition of an eval evaluator called `headNF` and a readback evaluator called `args`. The eval evaluator `headNF` literally implements in code the head spine (he) eval-apply evaluator of Section 5.6.

```

fun headNF (Lam(x,b)) = Lam(x, headNF b)
  | headNF (App(m,n)) =

```



```

      (case headNF m of
        Lam(x,b) => headNF (subst x n b)
      | m'       => App(m', n))
    | headNF t   = t

fun byName t = args (headNF t)
and args (Lam(x,b)) = Lam(x, args b)
  | args (App(m,n)) = App(args m, byName n)
  | args t          = t

```

The readback evaluator `args` delivers a NF by fully evaluating the unevaluated (operands of) neutrals (‘arguments’, hence its name ‘`args`’) in the HNF delivered by `headNF`. As the shape of a HNF indicates, the unevaluated neutrals occur at the top level or within abstraction bodies.

In its first clause, `args` evaluates an abstraction to NF by calling itself recursively to reach the unevaluated neutrals on whose operands it will eventually call the eval-readback composition (`byName`) of the second clause. In the second clause, `args` calls itself recursively on the operator of the neutral to distribute itself down nested applications, and calls the eval-readback composition (`byName`) that recursively distributes `headNF` down the operand. Finally, `args` is the identity on variables. It has no clause for redexes because it is a readback and merely distributes evaluation.

Recall from Section 5.8 that eval-readback `byName` is one-step equivalent to the eval-apply hybrid normal order evaluator (`hn`) discussed in that section. This can be gleaned from their respective definitions and is proven by the LWF proof (Section 8).

6.1. Monadic style eval-readback. We show the monadic definitions of `byValue` and `byName` in Haskell. Recall from the previous section that `eval` is identical to `bv` and `headNF` is identical to `he`. Hereafter we use `bv` and `he` and assume their eval-apply definitions given in Sections 4.1 and 5.6 respectively.

In monadic style, `byValue` and `byName` are defined by a monadic (Kleisli) composition. We have respectively replaced the calls to `byValue` and `byName` in `bodies` and `args` by their respective definitions as Kleisli compositions.

```

byValue :: Monad m => Term -> m Term
byValue = bodies <=< bv

bodies :: Monad m => Term -> m Term
bodies v@(Var _) = return v
bodies (Lam x b) = do b' <- (bodies <=< bv) b
                  return (Lam x b')
bodies (App m n) = do m' <- bodies m
                  n' <- bodies n
                  return (App m' n')

byName :: Monad m => Term -> m Term
byName = args <=< he

args :: Monad m => Term -> m Term
args v@(Var _) = return v
args (Lam x b) = do b' <- args b

```

```

    return (Lam x b')
args (App m n) = do m' <- args m
                n' <- (args <=< he) n
                return (App m' n')

```

We turn to a different example, namely, the eval-readback version of normal order presented in [GPN14, p. 183]. This evaluator is the composition of eval-apply call-by-name `bn` (Section 5.2) with the following readback evaluator `rn` (from `readback` `normal` order):

```

no :: Monad m => Term -> m Term
no = rn <=< bn

rn :: Monad m => Term -> m Term
rn v@(Var _) = return v
rn (Lam x b) = do b' <- (rn <=< bn) b
                return (Lam x b')
rn (App m n) = do m' <- rn m
                n' <- (rn <=< bn) n
                return (App m' n')

```

As a proper readback, `rn` has no clause for redexes because it simply distributes evaluation (the Kleisli composition) on the unevaluated abstractions and neutrals of the WHNF delivered by `bn`. A WHNF has no redex in outermost position.

The eval-readback examples discussed so far implement full-reducing strategies. The style can also be used for less-reducing strategies. We show two examples which are variations of `byValue` obtained by modifying the readback `bodies`. The first example is the ‘ahead machine’ (`am`) discussed in Section 5.10. In its eval-apply definition the strategy uses weak, strict and non-head call-by-value `bv` to evaluate operators and operands. Its eval-readback version uses `bv` for eval, like `byValue`, but its readback does not distribute evaluation on operands. We call this readback `bodies2` for easy comparison with `bodies`. We use the identity strategy `id` for the non-evaluation of operands.

```

am :: Monad m => Term -> m Term
am = bodies2 <=< bv

bodies2 :: Monad m => Term -> m Term
bodies2 v@(Var _) = return v
bodies2 (Lam x b) = do b' <- (bodies2 <=< bv) b
                    return (Lam x b')
bodies2 (App m n) = do m' <- bodies2 m
                    n' <- id n
                    return (App m' n')

```

The second example does not appear in the literature. It uses `bv` for eval but the readback calls eval (`bv`) directly on abstraction bodies rather than via the Kleisli composition. The strategy delivers a WNF where the abstractions in the spine are evaluated. We refrain from naming the strategy and will use the systematic notation that we introduce in Section 7.5 to identify it as $ER \circ bv$. We call the readback `bodies3` for easy comparison with `bodies`.

```

unnamed :: Monad m => Term -> m Term
unnamed = bodies3 <=< bv

bodies3 :: Monad m => Term -> m Term
bodies3 v@(Var _) = return v
bodies3 (Lam x b) = do b' <- bv b
                      return (Lam x b')
bodies3 (App m n) = do m' <- bodies3 m
                      n' <- bodies3 n
                      return (App m' n')

```

6.2. Natural semantics style eval-readback. In natural semantics, an eval-readback evaluator requires two separate sets of rules, one for eval and one for readback. Readback *has no CON rule* as it simply distributes evaluation in its ABS and NEU rules to reach the redexes within the intermediate form which are evaluated by eval (which does have a CON rule). Figure 7 (page 35) shows the natural semantics for the readbacks discussed in code in Section 6.1. We omit VAR rules because readbacks are identities on variables. The compositions are meta-notation to abbreviate a sequence of two premises, *e.g.* $(\text{bodies} \circ \text{bv})(B) = B''$ abbreviates the sequence of premise $\text{bv}(B) = B'$ followed by premise $\text{bodies}(B') = B''$. A proper natural semantics needs separate premises on which to stack evaluation trees, but we use compositions to abbreviate and to match the compositions in code. Compositions will be also handy when we generalise in Section 7.1. The identity strategy `id` is used for non-evaluation instead of an omitted premise.

6.3. Characterisation of the style. The examples so far illustrate some of the variations within the eval-readback style. But we must rule out spurious eval-readback evaluators defined by concocting vacuous readbacks. For example, the uniform eval-apply `bv` used as an eval stage can be defined spuriously in eval-readback style as the composition $\text{rb} \circ \text{bv}$ where `rb` is the following vacuous readback:

$$\text{ABS} \frac{\text{id}(B) = B}{\text{rb}(\lambda x.B) = \lambda x.B} \qquad \text{NEU} \frac{\text{id}(M) = M \quad \text{id}(N) = N}{\text{rb}(MN) = MN}$$

Substituting `rb` for the identity strategy `id` in any of the premises obtains a merely recursive readback that is also vacuous.

Readback must be called after eval and it must evaluate more than eval by calling eval either directly or in an eval-readback composition over some terms that were unevaluated by eval. For that, readback must also call itself recursively or in a composition. The following provisos characterise authentic readbacks in natural semantics:

- (ER₁) Readback must call itself recursively on the operator M to distribute itself down nested applications. (Notice that eval also calls itself recursively on operators.)
- (ER₂) In the other two premises, namely, evaluation of the body B (non-weakness) and evaluation of the operand N (non-headness):
 - (rb-ev) Readback must call *eval* on at least one of the two premises where eval was the identity, calling eval directly or in an eval-readback composition.
 - (rb-rb) Readback must call *readback* recursively on at least one of the two premises. In the case where eval was the identity on the premise, readback must call the eval-readback composition.

byValue's readback (bodies):

$$\text{ABS} \frac{(\text{bodies} \circ \text{bv})(B) = B''}{\text{bodies}(\lambda x.B) = \lambda x.B''} \quad \text{NEU} \frac{\text{bodies}(M) = M' \quad \text{bodies}(N) = N'}{\text{bodies}(MN) = M'N'}$$

byName's readback (args):

$$\text{ABS} \frac{\text{args}(B') = B'}{\text{args}(\lambda x.B) = \lambda x.B'} \quad \text{NEU} \frac{\text{args}(M) = M' \quad (\text{args} \circ \text{he})(N) = N''}{\text{args}(MN) = M'N''}$$

Normal order's readback (rn):

$$\text{ABS} \frac{(\text{rn} \circ \text{bn})(B) = B''}{\text{rn}(\lambda x.B) = \lambda x.B''} \quad \text{NEU} \frac{\text{rn}(M) = M' \quad (\text{rn} \circ \text{bn})(N) = N''}{\text{rn}(MN) = M'N''}$$

Ahead machine's readback (bodies₂):

$$\text{ABS} \frac{(\text{bodies}_2 \circ \text{bv})(B) = B''}{\text{bodies}_2(\lambda x.B) = \lambda x.B''} \quad \text{NEU} \frac{\text{bodies}_2(M) = M' \quad \text{id}(N) = N}{\text{bodies}_2(MN) = M'N}$$

A novel strategy's readback (bodies₃); the strategy is ER ◦ bv (Section 7.5):

$$\text{ABS} \frac{\text{bv}(B) = B'}{\text{bodies}_3(\lambda x.B) = \lambda x.B'} \quad \text{NEU} \frac{\text{bodies}_3(M) = M' \quad \text{bodies}_3(N) = N'}{\text{bodies}_3(MN) = M'N'}$$

Figure 7: Natural semantics of readback evaluators discussed in Section 6.1. The VAR rules are omitted because readbacks are identities on variables. There is no CON rule because readbacks merely distribute evaluation and do not contract redexes. The identity strategy id is used for non-evaluation instead of an omitted premise. The compositions are meta-notation.

The proviso (ER₂) holds when readback calls the eval-readback composition on one of the two premises and the identity on the other premise. It also holds when readback calls eval on a premise unevaluated by eval and calls readback on the other premise evaluated by eval. Other combinations are possible, all of which call eval at least once and readback at least once, directly on in a composition. It is trivial to check that the provisos guarantee $\text{rb} \circ \text{ev} \neq \text{ev}$ and $\text{rb} \circ \text{ev} \neq \text{rb}$.

6.4. Equivalences within the style. The splitting into eval and readback stages provides a modular approach but the stages are tied-in by eval's evaluation of redexes which restricts the amount of evaluation that can be moved from eval to readback and vice-versa. For illustration, consider the weak, non-strict, and non-head strategy implemented by the following eval-*apply* evaluator ea:

$$\frac{\text{id}(B) = B}{\text{ea}(\lambda x.B) = \lambda x.B} \quad \frac{\text{ea}(M) = \lambda x.B \quad \text{ea}([N/x](B)) = B'}{\text{ea}(MN) = B'}$$

$$\frac{\text{ea}(M) = M' \quad M' \neq \lambda x.B \quad \text{ea}(N) = N'}{\text{ea}(MN) = M'N'}$$

This novel strategy has an interesting property and will be discussed and named IIS in Section 7.3. The point now is that a one-step equivalent eval-*readback* evaluator is easily

defined by moving the evaluation of neutrals to a readback rb so that $\text{ea} = \text{rb} \circ \text{bn}$ where bn (call-by-name) is the head version of ea . We show the definition of rb . (The definition of bn is in Figure 4.)

$$\frac{\text{id}(B) = B}{\text{rb}(\lambda x.B) = \lambda x.B} \qquad \frac{\text{rb}(M) = M' \quad (\text{rb} \circ \text{bn})(N) = N'}{\text{rb}(MN) = M'N'}$$

How much evaluation can be moved between eval and readback to obtain equivalent eval -readback evaluators depends on the equivalence of eval -readback and eval -apply evaluators, because eval is naturally defined in eval -apply style as a uniform evaluator and the question relates to how much evaluation can be moved between the uniform subsidiary and the hybrid.

7. STRUCTURING THE PURE LAMBDA CALCULUS'S STRATEGY SPACE

7.1. Evaluation templates and generic evaluators. The four criteria of weakness, strictness, headness, and uniform/hybrid are manifested in the natural semantics by the presence or absence of a formula for evaluation. For instance, strictness is manifested in the CON rule by the presence of a formula for the evaluation of the operand N . Non-strictness is manifested by the absence of the formula. Both formulas can be generalised into a single formula $\text{ar}_1(N) = N'$ where ar_1 is a parameter. For strict evaluation, ar_1 is instantiated to a recursive call or to a subsidiary call. For non-strict evaluation, ar_1 is instantiated to id .

7.1.1. Eval-apply evaluation template and generic evaluator. The eval -apply evaluation template shown at the top of Figure 8 (page 37) generalises every formula with a parameter. The template defines an eval -apply evaluator ea with evaluator parameters la , op_1 , ar_1 , op_2 and ar_2 . The coloured parameters respectively determine the evaluator's **weakness**, **strictness**, and **headness**. (We explain the need for colours in a moment.) The use of parameters also permits the instantiation of subsidiaries and the control of staged evaluation in rule NEU (op_1 followed by op_2). Concretely, in the ABS rule, the la parameter determines the evaluation of (lambda) abstractions. In the leftmost formula of CON and NEU , the *shared* op_1 parameter determines the evaluation of operators in applications. In the CON rule, the ar_1 parameter determines the evaluation of operands (**arguments**) of redexes. In the NEU rule, the op_2 parameter determines the further evaluation of operators of neutrals. The ar_2 parameter determines the evaluation of operands of neutrals. The values provided for all these parameters, and their interaction, determine the type of evaluation and form of final result (WHNF , WNF , etc.). The meaning of the numbering of parameters differ. The op_1 and op_2 parameters are stages (NEU) whereas ar_1 and ar_2 are independent parameters for respectively evaluating operands in a redex (CON) and in a neutral (NEU). Thus, we use colours for the important parameters la , ar_1 , and ar_2 . The colours are also a visual aid and will be most useful from Section 7.3 onwards.

Recall from Section 5.14 that the staged evaluation of the operator in rule NEU with op_1 and op_2 is not a variability point because it is subsumed by the uniform/hybrid configuration. Only hybrid strategies use op_2 in NEU because they use the subsidiary on op_1 in NEU and CON . Consequently, there is no op_2 parameter in rule CON . It would be nonsensical because the operator abstraction $\lambda x.B$ would be further evaluated using op_2 but the redex $(\lambda x.B)N'$ would be contracted with the abstraction.

The instantiation table in the middle of Figure 8 shows that the evaluation template expresses all the eval -apply evaluators in Figures 4 and 5. More evaluators can be obtained

Eval-apply evaluation template:

$$\text{ABS } \frac{\text{la}(B) = B'}{\text{ea}(\lambda x.B) = \lambda x.B'}$$

$$\text{CON } \frac{\text{op}_1(M) = \lambda x.B \quad \text{ar}_1(N) = N' \quad \text{ea}([N'/x](B)) = B'}{\text{ea}(MN) = B'}$$

$$\text{NEU } \frac{\text{op}_1(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{op}_2(M') = M'' \quad \text{ar}_2(N) = N'}{\text{ea}(MN) = M''N'}$$

Eval-apply instantiation table:

Name	Eval-apply (ea)	la	op ₁	ar ₁	op ₂	ar ₂
Call-by-value	bv	id	bv	bv	id	bv
Call-by-name	bn	id	bn	id	id	id
Applicative order	ao	ao	ao	ao	id	ao
Normal order	no	no	bn	id	no	no
Head reduction	hr	hr	bn	id	hr	id
Head spine	he	he	he	id	id	id
Strict normalisation	sn	sn	bv	bv	sn	sn
Hybrid normal order	hn	hn	he	id	hn	hn
Hybrid applicative order	ha	ha	bv	ha	ha	ha
Ahead machine	am	am	bv	bv	am	bv
Head applicative order	ho	ho	ho	ho	id	id
Spine applicative order	so	so	ho	so	so	so
Balanced spine applicative order	bs	bs	ho	ho	bs	bs

Eval-apply generic evaluator:

```

type Red = forall m. Monad m => Term -> m Term
gen_eval_apply :: Red -> Red -> Red -> Red -> Red -> Red
gen_eval_apply la op1 ar1 op2 ar2 t = case t of
  v@(Var _) -> return v
  (Lam x b) -> do b' <- la b
                return (Lam x b')
  (App m n) -> do m' <- op1 m
                 case m' of (Lam x b) -> do n' <- ar1 n
                                           this (subst n' x b)
                             -          -> do m'' <- op2 m'
                                           n' <- ar2 n
                                           return (App m'' n')
  where this = gen_eval_apply la op1 ar1 op2 ar2

id :: Red
id = return

bal_hybrid_sn :: Red
bal_hybrid_sn = sn
  where bv :: Red
        bv = gen_eval_apply id bv bv id bv
        sn :: Red
        sn = gen_eval_apply sn bv bv sn sn

```

Figure 8: Eval-apply evaluation template, instantiation table, and generic evaluator.

by playing with the parameters. This is how we originally obtained head applicative order (ho), spine applicative order (so), and balanced spine applicative order (bs). We will obtain more novel evaluators in this fashion throughout this Section 7.

The code at the bottom of Figure 8 shows a generic (higher-order, parametric) eval-apply evaluator `gen_eval_apply` that implements the eval-apply evaluation template. It takes five plain evaluator parameters and delivers a plain evaluator. The type `Red` (from `Reducer`) is the type of a plain evaluator (Section 4). The local identifier `this` is an abbreviation to get a fixed-point definition. Plain evaluators are obtained as fixed-points of the generic evaluator. The figure shows the example for strict normalisation, which we call `bal_hybrid_sn` in order to define `bv` and `sn` as fixed points of `gen_eval_apply`.

7.1.2. Eval-readback evaluation template and generic readback. An evaluation template for eval-readback evaluators can be similarly provided. Actually, only a template for readback is needed because eval is naturally expressed in eval-apply style. Rarely will it be expressed in eval-readback style, and rarely still with the latter's eval again in eval-readback style, etc. Eventually, the last eval is written in eval-apply style.

Considering the readback rules and provisos in Sections 6 and 5.4, and assuming the abbreviation of compositions introduced in Section 6, there are two possible parameters for a readback evaluator `rb` in its `ABS` and `NEU` rules, namely `la` and `ar2`. Readback has no `CON` rule (no `op1` and `ar1` parameters) because the intermediate results delivered by eval have no redexes in outermost position. Also, according to proviso (`ER1`), the `op2` parameter must be a recursive call to readback to distribute it down nested applications.

Figure 9 (page 39) shows the readback evaluation template, an instantiation table that expresses all the readback evaluators discussed in Section 6, and a generic (higher-order, parametric) readback evaluator that implements the readback evaluation template. Plain eval-readback evaluators are obtained by monadic composition of an eval defined using `gen_eval_apply` and a readback defined using `gen_readback`. The figure shows an example for `byValue`, which we call `eval_readback.byValue` in order to define `bv` and `bodies` as fixed points. We will define more eval-readback evaluators in Section 7.5 by playing with the parameters.

7.2. Uniform/Hybrid as structuring criteria. The interaction of weakness, strictness and headness affects the amount of evaluation. Hence, these criteria do not provide an orthogonal classification of the strategy space. An orthogonal classification is provided by the uniform/hybrid criterion that splits the strategy space in two and is independent of the other three criteria.

As discussed in contribution **C5**, we must distinguish uniform/hybrid *strategy* from uniform/hybrid *style* of definitional device. It may be possible to write a uniform-style device (without subsidiary devices) for what is actually a hybrid strategy. We provide examples in Section 7.2.2. It may be also possible to write a hybrid-style device (with spurious subsidiary devices) for a uniform strategy. We will show an example in Section 7.4 (page 48).

Hybrid-style definitional devices have appeared and continue to appear in the literature, either under no denomination (*e.g.* [Rea89, Mac90, Pau96, Pie02]), or under names such as ‘levelled’ or ‘of-level-*n*’ (*e.g.* [PR99, RP04]), ‘stratified’ (*e.g.* [AP12, CG14]), or ‘layered’ (see [GPN14] for references). There are many hybrid evaluators and strategies (see Figure 13 on page 61). The notion has theoretical interest, to obtain properties such as completeness,

Readback evaluation template:

$$\text{ABS} \frac{\text{la}(B) = B'}{\text{rb}(\lambda x. B) = \lambda x. B'} \qquad \text{NEU} \frac{\text{rb}(M) = M' \quad \text{ar}_2(N) = N'}{\text{rb}(MN) = M'N'}$$

Eval-readback instantiation table:

Name	Eval	Readback (rb)	la	ar ₂
byValue	bv	bodies	bodies ◦ bv	bodies
byName	he	args	args	args ◦ he
Normal order	bn	rn	rn ◦ bn	rn ◦ bn
Ahead machine	bv	bodies ₂	bodies ₂ ◦ bv	id
<i>unnamed</i>	bv	bodies ₃	bv	bodies ₃

Readback generic evaluator:

```

type Red = forall m. Monad m => Term -> m Term
gen_readback :: Red -> Red -> Red
gen_readback la ar2 t = case t of
  v@(Var _) -> return v
  (Lam s b) -> do b' <- la b
                 return (Lam s b')
  (App m n) -> do m' <- this m
                 n' <- ar2 n
                 return (App m' n')
  where this = gen_readback la ar2

eval_readback_byValue :: Red
eval_readback_byValue = bodies <=< bv
  where bv      :: Red
        bv      = gen_eval_apply id bv bv id bv
        bodies :: Red
        bodies = gen_readback (bodies <=< bv) bodies

```

Figure 9: Readback evaluation template, instantiation table, and generic readback evaluator.

spininess, etc., and has proven useful in gradual typing [GPNS14], solvability [RP04, CG14, GPN16, AG22], machine-checked program derivations of operational semantics devices [BCZ17, BC19, BBCD20], and reduction theories for effect handlers [SPB23]. The concept of hybrid style is also intuitively connected to that of a staged eval-readback evaluator [Pau96, GL02]. Making the connection precise is partly the motivation for this paper.

In the introduction we gave an intuitive definition of uniform/hybrid strategy in terms of in/dependence on less-reducing subsidiary strategies. In [GPN14] we formalised the notion of uniform/hybrid strategy (which we redundantly called in that paper ‘uniform/hybrid strategy in its nature’) using small-step context-based reduction semantics [Fel87, FH92, FFF09]. This operational semantics style works with the notion of a one-step strategy (Appendix A), namely, a total function (a subset of the reduction relation) that takes an input term and locates and contracts one redex. An evaluation sequence is obtained by iteration. A strategy

can be defined extensionally by its set of so-called ‘contexts’ which are terms with one hole where the latter marks the redex position. The extensional definition is captured intensionally by a context-based reduction semantics which basically defines the set of contexts and permissible redexes using context-free grammars. Since we wish to improve on [GPN14], in this section we temporarily leave the big-step style and switch to that small-step definitional device. We provide a quick overview of it in Section 7.2.1. We provide the necessary intuitions for the formal definition of uniform/hybrid strategy in Section 7.2.2 using illustrative examples. We arrive at the formal definition in Section 7.2.3.

7.2.1. Small-step context-based reduction semantics. A *context* is a term with one hole, the latter written \square . For example, $\lambda x.\square x$ is a context. The set of contexts \mathcal{C} is defined by adding a hole to terms: $\mathcal{C} ::= \square \mid \lambda V.C \mid C \Lambda \mid \Lambda C$.¹³ We use $\mathcal{C}_1, \mathcal{C}_2$, etc., for contexts in \mathcal{C} . Given a context \mathcal{C}_1 we use the function application notation $\mathcal{C}_1(R)$ and $\mathcal{C}_1(\mathcal{C}_2)$ respectively for the operation of replacing the hole by a redex R and by another context \mathcal{C}_2 , irrespectively of variable captures. The first operation delivers a term and the second delivers a context. Because the function application notation is uncurried, we also use composition $\mathcal{C}_1 \circ \mathcal{C}_2$. For example, let $\mathcal{C}_1 = \lambda x.\square x$, $\mathcal{C}_2 = y\square$, and R a redex, then $\mathcal{C}_1(R) = \lambda x.Rx$, $\mathcal{C}_2(R) = yR$, $\mathcal{C}_1(\mathcal{C}_2) = \lambda x.y\square x$, $\mathcal{C}_2(\mathcal{C}_1) = y(\lambda x.\square x)$, $(\mathcal{C}_1(\mathcal{C}_2))(R) = (\mathcal{C}_1 \circ \mathcal{C}_2)(R) = \lambda x.yRx$, and $(\mathcal{C}_2(\mathcal{C}_1))(R) = (\mathcal{C}_2 \circ \mathcal{C}_1)(R) = y(\lambda x.Rx)$.

A strategy st is defined by a set of contexts $ST \in \mathcal{C}$ and a set of permissive redexes within holes R , both defined by context-free (in our case, EBNF) grammars with non-terminal axioms ST and R , such that for every term M there is at most one context $\mathcal{C}_1 \in ST$ (derivable from non-terminal ST) such that $M = \mathcal{C}_1(R)$ and $R \in R$ is the next redex to contract. The redex is contracted and the process is repeated to produce an evaluation sequence that stops when there is no derivable context. Spurious strategies like id and strategies that have no standalone hole context ($\mathcal{C}_1 = \square$) for the outermost redex are discarded.

To illustrate, take the following sets of contexts and of redexes for the applicative order strategy:

$$\begin{aligned} AO & ::= \square \mid \lambda V.AO \mid AO \Lambda \mid NF AO \\ R & ::= (\lambda V.NF)NF \end{aligned}$$

The set of contexts is generated by AO , and the set of permissible redexes is generated by R . Reading AO ’s grammar from left to right, a redex can occur in outermost position (standalone hole), under lambda, in the operator, or in the operand if the operator is in NF . Contraction takes place iff the redex has the body and the operand in NF . Given a term M , there is at most one context \mathcal{C}_1 derivable from non-terminal AO such that $M = \mathcal{C}_1(R)$ and $R \in R$. For example, given $(\lambda x.(\lambda y.z)x)N$ with arbitrary N , the uniquely-derivable context is $(\lambda x.\square)N$ with $(\lambda y.z)x \in R$. Contraction takes place delivering the contractum $(\lambda x.z)N$. The process is repeated to produce an evaluation sequence that stops when there is no derivable context.

¹³Recall that throughout the paper we overload non-terminal symbols to denote the sets generated by them, *e.g.* \mathcal{C} also denotes the set generated by that non-terminal.

7.2.2. *Intuition of uniform/hybrid strategy.* To provide the intuition we use the normal order strategy because it is the most well-known strategy of the pure lambda calculus. We reuse the context grammars presented in [GPN14] without redundant productions (*e.g.* redundant holes as discussed in [GPN14]). The details of the grammars are unimportant. The point is to show *alternative* grammars (styles) for the *same* hybrid strategy and illustrate from those examples the notions of dependency and inclusion that makes ‘hybrid’ a style-independent property. We start with a hybrid-style grammar for normal order which explicitly relies on a subsidiary call-by-name grammar. We then show two alternative hybrid-style grammars which ‘hide’ the call-by-name grammar and where the second grammar has a mutual dependency. We finally show a uniform-style grammar for normal order.

First, we start with a hybrid-style grammar of normal order contexts NO that uses subsidiary call-by-name contexts BN explicitly. The NE contexts are for reducing neutrals to NF. The set of redexes is R. We omit it in subsequent grammars because it remains the same throughout the discussion. In [GPN14] we showed that an implementation of the context-based reduction semantics with this grammar derives by program transformation to the normal order (no) evaluator of Figure 4.

$$\begin{aligned} \text{BN} &::= \square \mid \text{BN } \Lambda \\ \text{NO} &::= \text{BN} \mid \lambda V. \text{NO} \mid \text{NE} \\ \text{NE} &::= \text{NE } \Lambda \mid V \{ \text{NF} \}^* \text{NO} \\ \text{R} &::= (\lambda V. \Lambda) \Lambda \end{aligned}$$

Next, we show a hybrid-style grammar that hides BN. The subsidiary is now NW which contains the hole, the call-by-name contexts now represented by NW Λ , and the contexts that evaluate neutrals to WNF.

$$\begin{aligned} \text{NW} &::= \square \mid \text{NW } \Lambda \mid V \{ \text{NF} \}^* \text{NW } \{ \Lambda \}^* \\ \text{NO} &::= \text{NW} \mid \lambda V. \text{NO} \mid V \{ \text{NF} \}^* \text{NO } \{ \Lambda \}^* \end{aligned}$$

Next, we show a hybrid-style grammar which has a *mutual dependency* on a different non-terminal NN that has the hole, the call-by-name contexts, and the contexts that evaluate neutrals to NF. This is the ‘preponed’ grammar in [GPN14, p. 195].

$$\begin{aligned} \text{NN} &::= \square \mid \text{NN } \Lambda \mid V \{ \text{NF} \}^* \text{NO } \{ \Lambda \}^* \\ \text{NO} &::= \text{NN} \mid \lambda V. \text{NO} \end{aligned}$$

Finally, we show a uniform-style grammar, it has only one non-terminal.¹⁴

$$\text{NO} ::= \square \{ \Lambda \}^* \mid \lambda V. \text{NO} \mid V \{ \text{NF} \}^* \text{NO } \{ \Lambda \}^*$$

All the grammars generate the same set of normal order contexts. All BN, NW, and NN contexts are in NO. It is the inclusion (*i.e.* composition) of contexts what determines a dependency, not the expression of the inclusion using explicit non-terminals, which is a matter of style. The dependency is inextricable when contexts are not closed under inclusion. Let us rephrase the argument in [GPN14, Sec. 4]. Take the call-by-name context $\square N$. If we place in $\square N$ another call-by-name context, say $\square M$, then we get a call-by-name context $\square MN$. This is the case for any call-by-name context. Call-by-name is closed under context

¹⁴The one non-terminal grammar cannot be written in context-free grammars like BNF that lack the sequence notation $\{X\}^*$. However, the extra productions would only be for sequences, *i.e.* $A ::= \epsilon \mid XA$. A small-step device related to the one non-terminal grammar is the uniform-style structural operational semantics of normal order given in [Pie02, p. 502].

inclusion. If we place in $\square N$ a normal order context $\lambda x.\square$ then we get $(\lambda x.\square)N$ which is not a call-by-name context. Call-by-name does not include (depend on) normal order.

In contrast, take the context $\lambda x.\square N$ which is a normal order context. If we replace the hole with the normal order context $\lambda y.\square$ then we get $\lambda x.(\lambda y.\square)N$ which is not a normal order context. Normal order is not closed under context inclusion. If we replace the hole with any call-by-name context, say $\square M$, then we get a normal order context $\lambda x.\square MN$. Normal order depends on call-by-name.

NO defines a hybrid strategy that depends on what can be identified (or as we say in [GPN14], ‘unearthed’) as call-by-name contexts. It is made to explicitly depend on BN in the first definiens, on NW in the second, on NN (mutually) in the third, and with no explicit dependency (uniform-style) in the last.

7.2.3. Definition of uniform/hybrid strategy. The following definitions formalise the concepts of uniform and hybrid strategy using closeness under context inclusion (or context composition) and dependency on subsidiary. There may be many possible choices of subsidiaries each of which determines a particular definiens.

Definition 7.1. (i) A strategy ST is uniform iff for any $\mathcal{C}_1, \mathcal{C}_2 \in \mathbf{C}$ it is the case that $\mathcal{C}_1(\mathcal{C}_2) \in \text{ST}$ iff $\mathcal{C}_1 \in \text{ST}$ and $\mathcal{C}_2 \in \text{ST}$.
(ii) A strategy is hybrid iff it is not uniform.

Definition 7.2. A strategy HY *depends on* a different subsidiary strategy SU iff there exists a context $\mathcal{C}_1 \in \text{HY}$ such that for any context $\mathcal{C}_2 \in \mathbf{C}$, it is the case that $\mathcal{C}_1(\mathcal{C}_2) \in \text{HY}$ iff $\mathcal{C}_2 \in \text{SU}$.

These two definitions supersede the definitions in [GPN14, Sec. 4]. Notice that Definition 7.1(ii) implies Definition 7.2.

Proposition 7.3. *Definition 7.1(i) and Definition 7.2 cannot be the case simultaneously.*

Proof. Definition 7.1(i) requires that ST contexts be closed under context inclusion. Definition 7.2 requires that there exist SU contexts included in ST. Because all ST contexts are closed under inclusion then all SU contexts are also closed under inclusion and $\text{SU} = \text{ST}$. \square

As indicated in contribution **C5**, a hybrid strategy uses the subsidiary to evaluate the operator M in applications MN to obtain a redex $(\lambda x.B)N$. A balanced hybrid is either a non-strict hybrid (does not evaluate N) or is a strict hybrid that uses the subsidiary to evaluate N . All the balanced hybrid evaluators discussed in the survey do exactly this. In contrast, the unbalanced hybrid evaluators use themselves recursively on N .

Definition 7.4. A hybrid strategy is *balanced* when it has the same set of permissible redexes than its subsidiary.

7.3. Structuring the uniform space. We structure the strategy space starting with uniform strategies because they are simpler. The following provisos gleaned from the eval-apply instantiation table in Figure 8 characterise uniform-style eval-apply evaluators (‘uniform evaluators’, for short):

(U₁) Uniform evaluators call themselves recursively on the shared op_1 parameter of CON and NEU, and thus call the identity in NEU’s op_2 .

(U₂) Uniform evaluators differ only on the **weakness** (**la**), **strictness** (**ar₁**), and **headness** (**ar₂**) parameters, where they either call themselves recursively or call the identity.

Proposition 7.5. *An evaluator that satisfies the uniform provisos defines a uniform strategy.*

Proof. In all the natural semantics rules, each premise at most calls the evaluator recursively so that every term is evaluated by the same evaluator. There can be no inclusion of two contexts other than those generated by the evaluator. \square

Recall from Section 4.2 and Figure 3 that the set of contexts generated by the evaluator can be obtained from the CON nodes by replacing the contracta with holes in the evaluation sequences obtained from derivation trees.

A uniform evaluator inexorably defines a uniform strategy. In contrast, as shown in Section 7.2.2, a uniform-style small-step context-based reduction semantics can define a hybrid strategy.

To continue the analysis of uniform strategies without explicit reference to the template and instantiation table of Figure 8 (page 37), we introduce a convenient systematic notation for encoding uniform evaluators/strategies as a triple of letters where each letter can either be S for ‘self’ (recursive) or I for ‘identity’. The following table shows the encoding for the uniform strategies in the survey.

Name	la	ar ₁	ar ₂	Systematic notation
Call-by-value (bv)	id	bv	bv	ISS
Call-by-name (bn)	id	id	id	III
Applicative order (ao)	ao	ao	ao	SSS
Head spine (he)	he	id	id	SII
Head applicative order (ho)	ho	ho	id	SSI
IIS	id	id	IIS	IIS
SIS	SIS	id	SIS	SIS
ISI	id	ISI	id	ISI

The last four are novel, and the last three are unnamed. We name them using their encoding. The first one is IIS (weak, non-strict, non-head) which evaluates operands of neutrals that are not abstractions. The ea evaluator of Section 6.4 is exactly IIS. This strategy is interesting because it is complete for WNF in the pure lambda calculus and can be used to evaluate general recursive functions as follows:

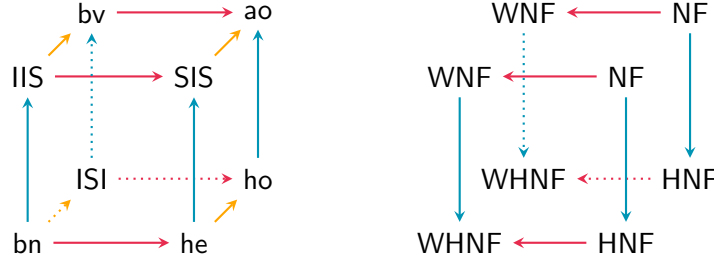
Strategy	Result	Combinator	Style	Thinking
IIS	WNF	Y	direct	none

The remaining two strategies are SIS (non-weak, non-strict, non-head and delivers NFs) and ISI (weak, strict, head and delivers WHNFs) which are uninteresting by themselves but are useful for hybrid and eval-readback evaluator definitions (Section 9).

By considering $I = 0$, $S = 1$, and the obvious partial order on boolean triples, from least reducing III (weak, non-strict, head) to most reducing SSS (non-weak, strict, non-head), the uniform strategy space is neatly structured by *Gibbons’s Beta Cube* lattice shown below left.¹⁵ Non-strict strategies are on the front face and strict strategies are on the rear face.

¹⁵As mentioned in the introduction, we name the cube after Jeremy Gibbons who suggested to us the use of booleans to construct a lattice.

The ‘more-reducing’ relation is reflected on the inclusion relation of final forms along the **weakness** and **headness** axes, shown on the right. The arrows in the opposite direction indicate the inclusion. For instance, NFs are valid HNFs so the latter include the former.



Notice that no uniform strategy is complete for normal forms. This requires a hybrid strategy (contribution **C6**, Section 7.4, Appendix E, [GPN14]).

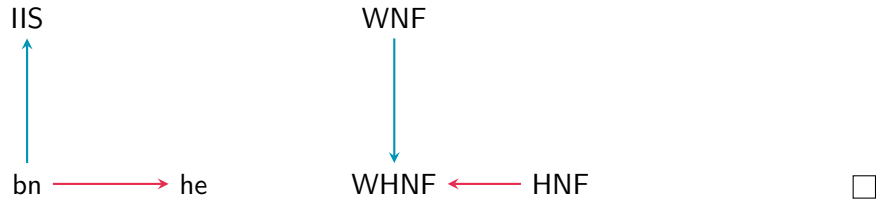
The inclusion relation of final forms suggests the possibility of absorption, which indeed holds for strategies on the left ‘L’ of the front (non-strict) face of the cube.

Definition 7.6 (Absorption). A strategy st_2 absorbs a strategy st_1 iff $st_2 \circ st_1 = st_2$.

Absorption is a big-step property. It is not necessary for st_1 to deliver a prefix of st_2 ’s evaluation sequence. Absorption does not hold when st_2 evaluates less than st_1 .

Proposition 7.7. *Both IIS and head spine (he) each absorb call-by-name (bn).*

Proof. The three strategies are quasi-leftmost, which subsumes the spine and leftmost-outermost strategies (Appendix E), and are complete for their respective final results, which satisfy the inclusions $WNF \subset WHNF$ and $HNF \subset WHNF$. If a term has a WNF/HNF then it also has a $WHNF$. Call-by-name (bn) finds the $WHNF$ and IIS/he finds the WNF/HNF .



Proposition 7.8. *Absorption does not hold with SIS.*

Proof. A counter-example term is $(\lambda k.k\Omega)(\lambda x.y)$. Indeed, SIS is non-weak and non-head, and diverges on that term whereas the other strategies in the ‘L’ are either weak or head and deliver y . □

Proposition 7.9. *Absorption does not hold between strategies that differ in strictness.*

Proof. The counter-example term is $(\lambda x.y)\Omega$ on which a strict strategy diverges but a non-strict strategy converges. □

Proposition 7.10. *Absorption does not hold on the strict face.*

Proof. The counter-examples are easily constructed using weakness and headness.

- The counter-example that **ao** does not absorb **ISI**, nor **bv**, nor **ho**, is the term $(\lambda x.y)(\lambda k.k\Omega)$. This is the counter-example for **SIS** above but with operator and operand flipped around. Clearly, **ao** diverges on the operand with Ω . But **ISI**, **bv**, and **ho** are weak or head and do not evaluate it.

- The counter-example that **bv** does not absorb ISI is $(\lambda x.y)(x\Omega)$.
- The counter-example that **ho** does not absorb ISI is $(\lambda x.y)(\lambda x.\Omega)$. □

7.4. Structuring the hybrid space. The following table, extracted from the instantiation table in Figure 8 (page 37), collects the hybrid evaluators discussed in the survey. The calls to subsidiaries are highlighted with the remaining parameters greyed out.

Name	la	op ₁	ar ₁	op ₂	ar ₂
Normal order (no)	no	bn	id	no	no
Head reduction (hr)	hr	bn	id	hr	id
Strict normalisation (sn)	sn	bv	bv	sn	sn
Hybrid normal order (hn)	hn	he	id	hn	hn
Hybrid applicative order (ha)	ha	bv	<u>ha</u>	ha	ha
Ahead machine (am)	am	bv	bv	am	bv
Spine applicative order (so)	so	ho	<u>so</u>	so	so
Balanced spine applicative order (bs)	bs	ho	ho	bs	bs

The only unbalanced hybrid evaluators in the survey are hybrid applicative order (**ha**) and spine applicative order (**so**), which call themselves recursively in **ar₁** (underlined in the table). Any strategy could be used on any of the five parameters but in practice the same uniform subsidiary is used on some parameters to satisfy the desired properties (completeness, spineness, etc.). Since Definition 7.1(ii) implies Definition 7.2, it is always possible to fix a subsidiary strategy on which a hybrid strategy depends. In particular, the subsidiary evaluator is called to evaluate operators in **op₁**, and the instantiations for the other parameters must be such that the hybrid evaluator evaluates more redexes than the subsidiary. The following provisos gleaned from the table characterise hybrid-style eval-apply evaluators (‘hybrid evaluators’, for short). The provisos assume that the subsidiary uniform evaluator upholds provisos (U_1) and (U_2).

- (H₁) On the shared **op₁** parameter of **CON** and **NEU**, the hybrid evaluator calls the subsidiary evaluator. On **NEU**’s **op₂** parameter, the hybrid evaluator calls itself recursively.
- (H₂) On the **la** and **ar₂** parameters, the hybrid evaluator must evaluate strictly more than the subsidiary evaluator, namely, at least one of the parameters must be a call to the hybrid, and there must be at least one call to the subsidiary or to the hybrid where the homonymous parameter in the subsidiary is the identity.
- (H₃) On the **ar₁** parameter, the hybrid evaluator calls the identity when non-strict, and evaluates at least as much as the subsidiary when strict. For balanced hybrid evaluators, the **ar₁** parameter has the same value as the homonymous parameter of the subsidiary.

Evaluating ‘more’ and ‘at least as much’ is succinctly expressed by $\{\text{id}\} \leq \{\text{id}, \text{su}, \text{hy}\}$ and $\{\text{su}\} \leq \{\text{su}, \text{hy}\}$ where the left sets have the permissible values of an homonymous subsidiary parameter and the right sets have the permissible values of the hybrid parameter given that homonymous subsidiary parameter.

For illustration, proviso (H₂) holds when the hybrid evaluator calls itself on one of **la**, **ar₂** where the subsidiary calls **id**, and on the other parameter it has the same value as the homonymous parameter of the subsidiary. For example, ahead machine (**am**) calls itself on **la** whereas its subsidiary call-by-value (**bv**) calls the identity on **la**, and ahead machine calls

bv on ar_2 exactly as bv does on its ar_2 . Proviso (H_2) also holds when the hybrid calls the subsidiary on one of la , ar_2 where the subsidiary calls id , and on the other parameter it calls itself recursively. For example, normal order (no) calls itself on la whereas its subsidiary call-by-name (bn) calls the identity, and normal order calls itself recursively on ar_2 .

The provisos are for evaluators, that is, for hybrid *style*. As we anticipated in Section 7.2 and will show in Section 7.4.1, a hybrid evaluator that upholds all the provisos can still define a uniform strategy. The aim of the provisos is not to match evaluators and strategies. The provisos express obvious syntactic and reasonable operational restrictions under which a hybrid-style evaluator can define a hybrid strategy. The provisos are gleaned from the evaluators in the survey, with an eye on the connection with the eval-readback provisos to prove one-step equivalence, including the cases when the eval-readback and the hybrid eval-apply evaluator define the same uniform strategy. (We compare the provisos for both styles after the following proposition.)

Proposition 7.11. *Let hy be a hybrid eval-apply evaluator that uses a uniform eval-apply evaluator su that upholds provisos (U_1) (U_2) . The hy evaluator cannot define a hybrid strategy according to Definitions 7.1(ii) and 7.2 if it does not uphold the hybrid provisos (H_1) , (H_2) (H_3) . If the strategy is balanced then hy upholds the ‘balanced’ part of proviso (H_3) .*

Proof. In contrapositive form, if the intended strategy is hybrid then the evaluator upholds the provisos. The su evaluator defines a uniform strategy by Proposition 7.5. By proviso (U_1) , the hybrid strategy depends on su because it is used (at least) on the op_1 parameter, and there is a derivation tree stacked on that premise with only subsidiary contexts. By the context inclusion property of the hybrid strategy, it evaluates more redexes than the subsidiary. This is satisfied by calling the hybrid evaluator in op_2 and by proviso (H_2) and the first part of (H_3) which cover all possible stackings of derivation trees and contexts where hy evaluates more than su . If the hybrid strategy is balanced, the set of permissible redexes must be the same as su ’s. This is satisfied by hy in its use of su in op_1 to uphold proviso (H_1) and by the balanced part of (H_3) . \square

A comparison between the hybrid provisos and the eval-readback provisos of Section 6.3 provides intuitions for the role of the provisos in determining the conditions for one-step equivalence.

- The readback evaluator comes after the eval evaluator, and readback evaluates more than eval. The hybrid evaluator comes after the subsidiary evaluator (on operators), and the hybrid evaluates more than the subsidiary.
- Eval calls itself on op_1 . The subsidiary calls itself on op_1 . Readback calls itself on op_2 . The hybrid calls itself on op_2 .
- The provisos (ER_2) and (H_2) (H_3) are related: a balanced hybrid evaluator uses the same parameter as its subsidiary for ar_1 , and in the other two parameters (la and ar_2) one of the following must be the case:
 - The hybrid calls the identity, or the subsidiary, or itself, if the subsidiary calls the identity in the homonymous parameter.
 - The hybrid calls the subsidiary or itself, if the subsidiary calls itself in the homonymous parameter.

7.4.1. *A systematic notation for hybrid evaluators.* To continue the analysis of hybrid evaluators without explicit reference to the template and instantiation table of Figure 8 (page 37), we introduce a convenient systematic notation $XXX \diamond YYY$ for encoding hybrid evaluators by means of two triples. The right triple YYY is the Beta Cube triple of the uniform evaluator used as a subsidiary (table in Section 7.3, page 43). We may on occasion abbreviate it using the uniform’s name when it has one (*e.g.* bv). The left triple XXX encodes the behaviour of the hybrid evaluator on its la , ar_1 , and ar_2 parameters with the following values: I for call to the identity, S for call to the subsidiary, and H for recursive call to the hybrid. Conveniently, S can be read as ‘subsidiary’ across the encoding because in the right triple it stands for ‘self’ which is a recursive call of the subsidiary. The diagram in Figure 10 summarises. As an aid to the reader, Figure 11 shows how to obtain the natural semantics of a hybrid evaluator from its systematic notation by instantiating the eval-apply evaluation template of Figure 8. The example is for the hybrid evaluator $HIS \diamond SII$ which is not discussed in the survey. We invite readers to take a couple of notations from the above table, obtain the natural semantics in the same way as illustrated in Figure 11, and compare the result (modulo evaluator names) to the natural semantics in the survey.

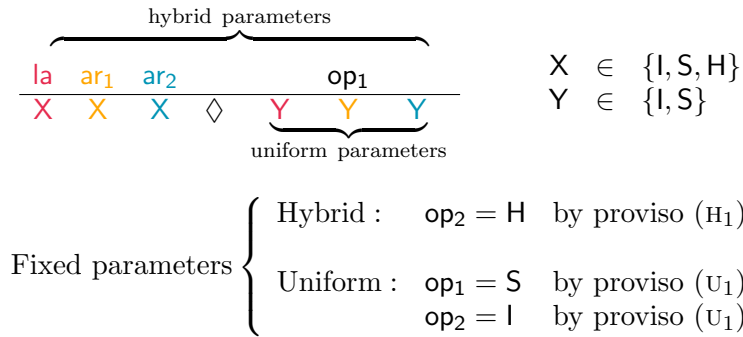


Figure 10: Systematic notation for encoding hybrid evaluators.

The following table lists the encodings for the hybrid evaluators in the survey. We omit the colours.

Name	Systematic	Abbreviated
Normal order (no)	HIH \diamond III	HIH \diamond bn
Head reduction (hr)	HIH \diamond III	HI \diamond bn
Strict normalisation (sn)	HSH \diamond ISS	HSH \diamond bv
Hybrid normal order (hn)	HIH \diamond SII	HIH \diamond he
Hybrid applicative order (ha)	HHH \diamond ISS	HHH \diamond bv
Ahead machine (am)	HSS \diamond ISS	HSS \diamond bv
Spine applicative order (so)	HHH \diamond SSI	HHH \diamond ho
Balanced spine applicative order (bs)	HSH \diamond SSI	HSH \diamond ho

We can use the notation to name hybrid evaluators for strategies not shown in the survey. For example, the following evaluators use the subsidiary on the la parameter unlike the

$$\begin{array}{c}
\text{ABS} \frac{\text{SII}(B) = B'}{\text{SII}(\lambda x.B) = \lambda x.B'} \quad \text{CON} \frac{\text{SII}(M) = \lambda x.B \quad \text{id}(N) = N \quad \text{SII}([N/x](B)) = B'}{\text{SII}(MN) = B'} \\
\text{NEU} \frac{\text{SII}(M) = M' \quad M' \not\equiv \lambda x.B \quad \text{id}(M') = M' \quad \text{id}(N) = N}{\text{SII}(MN) = M'N} \\
\text{ABS} \frac{(\text{HIS} \diamond \text{SII})(B) = B'}{(\text{HIS} \diamond \text{SII})(\lambda x.B) = \lambda x.B'} \quad \text{CON} \frac{\text{SII}(M) = \lambda x.B \quad \text{id}(N) = N \quad (\text{HIS} \diamond \text{SII})([N/x](B)) = B'}{(\text{HIS} \diamond \text{SII})(MN) = B'} \\
\text{NEU} \frac{\text{SII}(M) = M' \quad M' \not\equiv \lambda x.B \quad (\text{HIS} \diamond \text{SII})(M') = M'' \quad \text{SII}(N) = N'}{(\text{HIS} \diamond \text{SII})(MN) = M''N'}
\end{array}$$

Figure 11: Natural semantics obtained by instantiating $\text{HIS} \diamond \text{SII}$ on the eval-apply evaluation template of Figure 8. The instantiations of parameters are coloured according to the colour of the parameter.

evaluators in the survey, which call themselves recursively on that parameter.

Systematic	la	op ₁	ar ₁	op ₂	ar ₂
SIH \diamond he	he	he	id	SIH \diamond he	SIH \diamond he
SSH \diamond ho	ho	ho	ho	SSH \diamond ho	SSH \diamond ho

The notation is semantically restricted by the hybrid provisos. When the left triple is identical to the right triple (*e.g.* $\text{SIS} \diamond \text{SIS}$) then proviso (H_2) fails and the evaluator defined is actually uniform (*e.g.* SIS) because the evaluation trees differ only in the evaluator name at the roots. Also, evaluators such as $\text{HIH} \diamond \text{SIS}$, $\text{HSI} \diamond \text{SSI}$, $\text{IHH} \diamond \text{ISS}$, $\text{HHI} \diamond \text{SSI}$, etc., are spurious hybrid evaluators because they do not evaluate more than the subsidiary: the hybrid simply calls itself on the same parameters evaluated by the subsidiary and does not evaluate more on the other parameters.

Proviso (H_3) dictates that balanced hybrid evaluators have the form $\text{XIX} \diamond \text{YIY}$ (non-strict) and $\text{XSX} \diamond \text{YSY}$ (strict), with the other two parameters X and Y restricted by proviso (H_2) .

We now show the example anticipated in Section 7.2 of a hybrid evaluator that actually defines a uniform strategy. The uniform strategy is IIS , defined by the uniform evaluator encoded by the notation. This strategy can be defined by the one-step equivalent hybrid evaluator $\text{IIH} \diamond \text{bn}$. The equivalence is easy to see with the notation: the hybrid evaluator is constructed by moving non-headness from the uniform triple IIS to a hybrid triple IIH , leaving a head uniform subsidiary ($\text{bn} = \text{II}$). This is the same move shown in Section 6.4 with the eval-apply ea evaluator and the eval-readback $\text{rb} \circ \text{bn}$ evaluator. The ea evaluator is exactly IIS , and the eval-readback evaluator is obtained by moving non-headness to the readback stage. Indeed, the hybrid evaluator $\text{IIH} \diamond \text{bn}$ and the eval-readback evaluator $\text{rb} \circ \text{bn}$ of Section 6.4 are one-step equivalent (Figure 14). Both define the uniform strategy IIS which is better defined in uniform style.

Notice that there are shallow hybrid evaluators such as $\text{SIH} \diamond \text{III}$ which uphold the provisos. This evaluator calls its subsidiary bn on la for a shallow non-weak evaluation, and calls itself on ar_2 to perform that very shallow evaluation on operands of neutrals.

Such shallow hybrid evaluators are suitable non-uniform eval candidates for eval-readback evaluators.

7.4.2. (Non-)Equivalences within the style. With the systematic notation we can conveniently show (non-)equivalences among hybrid evaluators. A simple example is the non-equivalence of $\text{HIS} \diamond \text{bn}$ and $\text{HIS} \diamond \text{IIS}$. As their common left triple indicates, both hybrids use the subsidiary to evaluate neutrals (HIS). But bn (= III) is head and IIS is non-head. A counter-example neutral that disproves the equivalence is trivial: $x(xR)$ where R is a redex. This redex is evaluated by $\text{HIS} \diamond \text{IIS}$ but not by $\text{HIS} \diamond \text{bn}$.

An equivalence can be obtained when the hybrid calls itself recursively on the neutral instead of calling the subsidiary, that is, the left triple is not HIS but HIH . The hybrids are $\text{HIH} \diamond \text{bn}$ (normal order) and $\text{HIH} \diamond \text{IIS}$. However, the equivalence is one-step modulo commuting redexes because IIS is weak and non-head, so it evaluates operands of neutrals that are not abstractions. A counter-example neutral is $x(\lambda x.R_1)R_2R_3$ where R_i are redexes. Normal order evaluates the redexes from left to right whereas $\text{HIH} \diamond \text{IIS}$ calls IIS on the operator $x(\lambda x.R_1)R_2$ and thus R_2 is the first evaluated redex.

As another example, we show the non-equivalence of normal order ($\text{HIH} \diamond \text{bn}$) respectively with hybrid normal order ($\text{HIH} \diamond \text{he}$) and head reduction ($\text{HII} \diamond \text{bn}$). In the first case, the encodings differ only on the la value of the uniform triple (III vs SII), which marks the difference between delivering a WHNF or a HNF by the subsidiary. The hybrid triple HIH indicates that evaluation continues (and in this case, completes) with non-weak and non-head evaluation to NF . In the second case of normal order vs head reduction, the common subsidiary delivers a WHNF and the encodings differ only on the ar_2 parameter of the hybrid triple (HIH vs HII), which marks the difference between delivering a NF or a HNF as a final result.

There are no equivalences on the strict space because a strict subsidiary is used by the hybrid to evaluate operators and, being strict, the subsidiary evaluates the operands of redexes within the operators. The weakness and headness is tied-in with the evaluation of those operands. For example, all the hybrids with respective subsidiaries ISI (head) and bv (= ISS) (non-head) are not equivalent. For instance $\text{HSS} \diamond \text{ISI}$ and $\text{HSS} \diamond \text{ISS}$, or $\text{HSH} \diamond \text{ISI}$ and $\text{HSH} \diamond \text{ISS}$ (strict normalisation). A counter-example is easy to find by placing a redex R within a neutral that is the operand of a redex that is the operator first evaluated by the subsidiaries: $((\lambda x.B)(xR))N$.

7.5. Structuring the eval-readback space. The provisos for eval-readback evaluators have been introduced in Section 6.3. We introduce a systematic notation $\text{XX} \circ \text{YYY}$ for encoding eval-readback evaluators by means of a pair and a triple. The right triple YYY is the Beta Cube triple of the uniform evaluator used as eval. We may on occasion abbreviate it using the uniform's name when it has one. The left pair XX encodes the behaviour of the readback on its la and ar_2 parameters with the following values: I for call to the 'identity', E for call to 'eval', R for call to 'readback', and (RE) for the composition of readback after eval. Mind the parenthesis: (RE) is not a pair. Valid pairs are ER , RE , and $(\text{RE})\text{X}$ and $\text{X}(\text{RE})$ for any valid value of X . The following table shows the encodings for the eval-readback

evaluators in Figure 9 (colours omitted).

Name	Systematic
byValue	(RE)R ◦ bv
byName	R(RE) ◦ he
Normal order	(RE)(RE) ◦ bn
Ahead machine	(RE)I ◦ bv
<i>unnamed</i>	ER ◦ bv

The proviso (ER_2) dictates that one element X of the readback pair must be E or (RE) . By unfolding the proviso we obtain some new eval-readback evaluators:

- One element of the pair is (RE) and the other is I . For instance, $I(RE) \circ bn$, $(RE)I \circ bn$, and $(RE)I \circ bv$.
- One element of the pair is E and the other is R when the corresponding premises in eval are I . For instance, $ER \circ IIS$, $RE \circ he$, and $ER \circ bv$.

The $I(RE) \circ bn$ and $(RE)I \circ bn$ evaluators are respectively one-step equivalent to the uniform evaluators IIS and head reduction (hr). The $(RE)I \circ bv$ evaluator is one-step equivalent to the hybrid evaluator for the ahead machine (am). These and more equivalences are collected in Figure 14 (page 62) and follow from the LWF proof given in the following section.

8. EQUIVALENCE PROOF BY LWF TRANSFORMATION

Our equivalence proof uses LWF to fuse the eval and readback stages into the single-function balanced hybrid eval-apply evaluator. The proof does *not* apply LWF on a generic eval-readback evaluator. It applies LWF on a *plain but arbitrary* eval-readback evaluator where its uniform eval evaluator is a fixed-point of generic `gen_eval_apply` (Figure 8, page 37) and its readback evaluator is a fixed-point of generic `gen_readback` (Figure 9, page 39). The arbitrariness is achieved by considering some parameters as ‘free’ (existentially quantified in the sense of logic programming) by giving them a well-typed `undefined` value. The undefined identifiers provide genericity in the form of arbitrariness without the need for the higher-order parametrisation of generic evaluators. The provisos and the LWF steps impose constraints on the possible concrete values that can be passed to the parameters. The result of the transformation is the one-step equivalent *plain* balanced hybrid eval-apply evaluator and a set of constraints that capture the possible substitutions. The subsidiary and the hybrid evaluators are fixed-points of `gen_eval_apply`. The following sections elaborate.

8.1. Plain but arbitrary eval-readback evaluators. Take the plain `byValue` evaluator of Figure 9 (page 39):

```
eval_readback_byValue :: Red
eval_readback_byValue = bodies <=< bv
  where bv      :: Red
        bv      = gen_eval_apply id bv bv id bv
        bodies  :: Red
        bodies  = gen_readback (bodies <=< bv) bodies
```

Both `bv` and `bodies` are locally-defined Haskell identifiers. The first is a fixed-point of `gen_eval_apply` that satisfies the provisos (U_1) and (U_2) for uniform evaluators. The second is a fixed-point of `gen_readback` that satisfies the provisos (ER_1) and (ER_2) for readback evaluators.

We generalise to a plain but arbitrary eval-readback evaluator that implements a strategy `xx` by using type-checked but undefined identifiers for the *actual* parameters of `gen_eval_apply` and `gen_readback` that are not fixed by the provisos. The undefined identifiers are named by combining the parameter's name with a letter `e` and `r` to distinguish between the eval and the readback parameter. They stand for *actual, not formal*, parameters.

```
eval_readback_xx :: Red
eval_readback_xx = reb_xx <=< evl_xx
  where evl_xx :: Red
        evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
        reb_xx :: Red
        reb_xx = gen_readback lar_xx ar2r_xx

lae_xx  :: Red
lae_xx  = undefined
...
ar2r_xx :: Red
ar2r_xx = undefined
```

The original `eval_readback_byValue` evaluator is obtained by editing the code and rewriting `lae_xx` to `id`, `lar_xx` to `(reb_xx <=< evl_xx)`, etc.

The provisos for the uniform evaluator and the readback evaluator constrain the possible values of the actual parameters. Let us transcribe the provisos in code:

- (U_1) The second and fourth actual parameters of `gen_eval_apply` are respectively `evl_xx` and `id`.
- (U_2) The identifiers `lae_xx`, `ar1e_xx`, and `ar2e_xx` stand for either `eval_xx` or `id`.
- (ER_1) Function `gen_readback` only takes two actual parameters: `lar_xx` and `ar2e_xx`.
- (ER_2) In the other two parameters `lar_xx` and `ar2r_xx`:
- (rb-ev) At least one stands for `evl_xx` where the homonymous parameter (`lae_xx` or `ar2e_xx`) stands for `id`.
- (rb-rb) At least one stands for `reb_xx`.

Since `reb_xx` can only be called on a parameter after `evl_xx` has been already called on the term affected by that parameter, readback may call the identity, itself, or the eval-readback composition as specified by the following table:

Eval value	Readback value
<code>id</code>	<code>id</code> or <code>eval_xx</code> or <code>reb_xx <=< evl_xx</code>
<code>evl_xx</code>	<code>id</code> or <code>reb_xx</code>

If one of the parameters has no call to `evl_xx` (resp. `reb_xx`) then the other parameter must contain a call to `evl_xx` (resp. `reb_xx`). These constraints will be elaborated during the LWF transformation.

8.2. Plain but arbitrary balanced hybrid eval-apply evaluators. Take the plain balanced hybrid ‘strict normalisation’ evaluator of Figure 8 (page 37):

```
bal_hybrid_sn :: Red
bal_hybrid_sn = sn
  where bv :: Red
        bv = gen_eval_apply id bv bv id bv
        sn :: Red
        sn = gen_eval_apply sn bv bv sn sn
```

Both `bv` and `sn` are locally-defined Haskell identifiers. The first has the same code as in Section 8.1. The second is a fixed-point of `gen_eval_apply` that satisfies the provisos (H_1) , (H_2) , and (H_3) for balanced hybrid evaluators.

We generalise to a plain but arbitrary balanced hybrid evaluator that implements a strategy `xx` by using type-checked undefined identifiers as before:

```
bal_hybrid_xx :: Red
bal_hybrid_xx = hyb_xx
  where sub_xx :: Red
        sub_xx = gen_eval_apply las_xx sub_xx ar1s_xx id ar2s_xx
        hyb_xx :: Red
        hyb_xx = gen_eval_apply lah_xx sub_xx ar1s_xx hyb_xx ar2h_xx

las_xx :: Red
las_xx = undefined
...
ar2h_xx :: Red
ar2h_xx = undefined
```

The undefined identifiers are named by combining the parameter’s name with a letter `s` and `h` to distinguish between the subsidiary and the hybrid parameter.

Let us transcribe in code the provisos for the uniform evaluator and the balanced hybrid evaluator:

- (U₁) The second and fourth actual parameters of the first `gen_eval_apply` are respectively `sub_xx` and `id`.
- (U₂) The identifiers `las_xx`, `ar1s_xx`, and `ar2s_xx` stand for either `sub_xx` or `id`.
- (H₁) The second and fourth actual parameters of the second `gen_eval_apply` are respectively `sub_xx` and `hyb_xx`.
- (H₂) At least one of `lah_xx` and `ar2h_xx` stand for `hyb_xx`, and one must stand for `sub_xx` or `hyb_xx` where the homonymous parameter (`las_xx` or `ar2s_xx`) stands for `id`.
- (H₃) The third actual parameter of both `gen_eval_apply` functions is `ar1s_xx` which stands for either `id` or `sub_xx`.

Let us unpack (H_2) focusing on `lah_xx`. The same unpacking applies to `ar2h_xx`. Let `lah_xx` stand for `hyb_xx`. If `las_xx` stands for `sub_xx` then `ar2h_xx` may either stand for `sub_xx` or for `hyb_xx`. If `las_xx` stands for `id` then `ar2h_xx` may stand for the same as `ar2s_xx`, or for `sub_xx`, or for `hyb_xx`. In sum, for `lah_xx` and `ar2h_xx` the hybrid may call the identity, the subsidiary, or itself as specified by the following table:

Subsidiary value	Hybrid value
<code>id</code>	<code>id</code> or <code>sub_xx</code> or <code>hyb_xx</code>
<code>sub_xx</code>	<code>sub_xx</code> or <code>hyb_xx</code>

8.3. Statement of one-step equivalence. In Section 8.4 we show the LWF transformation of `eval_readback_xx` into `bal_hybrid_xx` when they uphold their respective provisos in code stated in Sections 8.1 (page 51) and 8.2 (page 52). At the end of the transformation we collect the possible substitutions for the undefined parameters in a list of equations. The equations are also a means to produce one evaluator style given actual code for the other.

Proposition 8.1. *Let $\mathbf{rb} \leq \mathbf{ev}$ be a concrete eval-readback evaluator defined by concretising the arbitrary parameters of `eval_readback_xx` such that \mathbf{ev} and \mathbf{rb} uphold their respective provisos in code (page 51). Let \mathbf{hy} be a concrete balanced hybrid eval-apply evaluator defined by concretising the arbitrary parameters of `bal_hybrid_xx` such that \mathbf{su} and \mathbf{hy} uphold their respective provisos in code (page 52).*

The evaluators $\mathbf{rb} \leq \mathbf{ev}$ and \mathbf{hy} are one-step equivalent (modulo commuting redexes when `ar2e_xx` is not the identity) when the following equations hold for the concrete values provided for the undefined parameters in the equations:

$$\begin{aligned}
 \mathbf{lae_xx} &= \mathbf{las_xx} \\
 \mathbf{ar1e_xx} &= \mathbf{ar1s_xx} \\
 \mathbf{ar2e_xx} &= \mathbf{ar2s_xx} \\
 \mathbf{lar_xx} \leq \mathbf{lae_xx} &= \mathbf{lah_xx} \\
 \mathbf{ar2r_xx} \leq \mathbf{ar2e_xx} &= \mathbf{ar2h_xx}
 \end{aligned}$$

Figure 12 shows the instantiations of the equations, and thus the evaluator equivalences, for some example strategies.

Corollary 8.2. $\mathbf{rb} \leq \mathbf{ev} = \mathbf{hy}$.

Corollary 8.3. *The implemented strategies uphold $\mathbf{ev} = \mathbf{su}$ and $\mathbf{rb} \circ \mathbf{ev} = \mathbf{hy}$.*

Corollary 8.4. $\mathbf{hy} \circ \mathbf{ev} = \mathbf{hy}$ (*hy absorbs ev*)

Proof.

$$\begin{aligned}
 &\mathbf{hy} \circ \mathbf{ev} = \mathbf{hy} \\
 &\Leftrightarrow \{ \mathbf{rb} \circ \mathbf{ev} = \mathbf{hy} \} \\
 &(\mathbf{rb} \circ \mathbf{ev}) \circ \mathbf{ev} = \mathbf{rb} \circ \mathbf{ev} \\
 &\Leftrightarrow \{ \text{associativity} \} \\
 &\mathbf{rb} \circ (\mathbf{ev} \circ \mathbf{ev}) = \mathbf{rb} \circ \mathbf{ev} \\
 &\Leftrightarrow \{ \mathbf{ev} \circ \mathbf{ev} = \mathbf{ev} \} \\
 &\mathbf{rb} \circ \mathbf{ev} = \mathbf{rb} \circ \mathbf{ev} \\
 &\Leftrightarrow \{ \text{reflexivity} \} \\
 &\text{true}
 \end{aligned}$$

□

8.4. Proof by LWF transformation. Starting with `eval_readback_xx` we fuse `ev1_xx` and `reb_xx` into a single function to arrive at `bal_hybrid_xx` according to the following LWF-steps [OS07].

Normal order:

$$\begin{array}{l} (\text{RE})(\text{RE}) \circ \text{III} \quad \text{HIH} \diamond \text{III} \\ \text{id} = \text{id} \\ \text{id} = \text{id} \\ \text{id} = \text{id} \\ (\text{reb} \leq\leq \text{evl}) \leq\leq \text{id} = \text{hyb} \\ (\text{reb} \leq\leq \text{evl}) \leq\leq \text{id} = \text{hyb} \end{array}$$

$(\text{RE})(\text{RE}) \circ \text{III}$ is one-step equivalent to $\text{HIH} \diamond \text{III}$ as (RE) after I is equal to H .

Hybrid normal order:

$$\begin{array}{l} \text{R}(\text{RE}) \circ \text{SII} \quad \text{HIH} \diamond \text{SII} \\ \text{evl} = \text{sub} \\ \text{id} = \text{id} \\ \text{id} = \text{id} \\ \text{reb} \leq\leq \text{evl} = \text{hyb} \\ (\text{reb} \leq\leq \text{evl}) \leq\leq \text{id} = \text{hyb} \end{array}$$

$\text{R}(\text{RE}) \circ \text{SII}$ is one-step equivalent to $\text{HIH} \diamond \text{SII}$ as both R after S and RE after I are equal to H .

Head reduction:

$$\begin{array}{l} (\text{RE})\text{I} \circ \text{III} \quad \text{HII} \diamond \text{III} \\ \text{id} = \text{id} \\ \text{id} = \text{id} \\ \text{id} = \text{id} \\ \text{reb} \leq\leq \text{evl} = \text{hyb} \\ \text{id} \leq\leq \text{id} = \text{id} \end{array}$$

$(\text{RE})\text{I} \circ \text{III}$ is one-step equivalent to $\text{HII} \diamond \text{III}$ as RE is equal to H .

Strict normalisation:

$$\begin{array}{l} (\text{RE})\text{R} \circ \text{ISS} \quad \text{HSH} \diamond \text{ISS} \\ \text{id} = \text{id} \\ \text{evl} = \text{sub} \\ \text{evl} = \text{sub} \\ (\text{reb} \leq\leq \text{evl}) \leq\leq \text{id} = \text{hyb} \\ \text{reb} \leq\leq \text{evl} = \text{hyb} \end{array}$$

$(\text{RE})\text{R} \circ \text{ISS}$ is one-step equivalent to $\text{HSH} \diamond \text{ISS}$ as both RE after I and R after S are equal to H .

Ahead machine:

$$\begin{array}{l} (\text{RE})\text{I} \circ \text{ISS} \quad \text{HSS} \diamond \text{ISS} \\ \text{id} = \text{id} \\ \text{evl} = \text{sub} \\ \text{evl} = \text{sub} \\ (\text{reb} \leq\leq \text{evl}) \leq\leq \text{id} = \text{hyb} \\ \text{id} \leq\leq \text{evl} = \text{sub} \end{array}$$

$(\text{RE})\text{I} \circ \text{ISS}$ is one-step equivalent to $\text{HSS} \diamond \text{ISS}$ as RE after I is equal to H , and I after S is equal to S .

Figure 12: Equation instantiations for some example strategies.

(1) Start with `eval_readback_xx`:

```
eval_readback_xx :: Red
eval_readback_xx = reb_xx <=< evl_xx
  where evl_xx :: Red
```

```

evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
reb_xx :: Red
reb_xx = gen_readback lar_xx ar2r_xx

```

- (2) Inline `evl_xx` in `reb_xx` \ll `evl_xx` and beta-convert the composition attending to the cases of an input term `t`. The result is `eval_readback_xx1` below where the calls to `reb_xx` are all in tail position:

```

eval_readback_xx1 :: Red
eval_readback_xx1 t = case t of
  v@(Var _) -> return v
  (Lam s b) -> do b' <- lae_xx b
                l <- return (Lam s b')
                reb_xx l
  (App m n) -> do m' <- evl_xx m
                case m' of (Lam s b) -> do n' <- ar1e_xx n
                                           s <- evl_xx (subst n' s b)
                                           reb_xx s
                - -> do m'' <- id m'
                       n' <- ar2e_xx n
                       a <- return (App m'' n')
                       reb_xx a

where evl_xx :: Red
      evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
      reb_xx :: Red
      reb_xx = gen_readback lar_xx ar2r_xx

```

- (3) Inline the first and last occurrences of `reb_xx` in `eval_readback_xx1`. The result is `eval_readback_xx2` below:

```

eval_readback_xx2 :: Red
eval_readback_xx2 t = case t of
  v@(Var _) -> return v
  (Lam s b) -> do b' <- lae_xx b
                b'' <- lar_xx b'
                return (Lam s b'')
  (App m n) -> do m' <- evl_xx m
                case m' of (Lam s b) -> do n' <- ar1e_xx n
                                           s <- evl_xx (subst n' s b)
                                           reb_xx s
                - -> do m'' <- id m'
                       n' <- ar2e_xx n
                       m''' <- reb_xx m''
                       n'' <- ar2r_xx n'
                       return (App m''' n'')

where evl_xx :: Red
      evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
      reb_xx :: Red
      reb_xx = gen_readback lar_xx ar2r_xx

```


- (4) In `eval_readback_xx2`, swap the lines

```
n'   <- ar2e_xx n
m''  <- reb_xx m''
```

This swapping is required by the LWF transformation and alters the evaluation order in neutrals: the operator is evaluated entirely and then the operand entirely, instead of alternatingly. One-step equivalence is preserved except for strategies where `ar2e_xx` is not the identity. In that case one-step equivalence is modulo commuting redexes. We identify such strategies in Section 9.

After swapping, replace the binding `m'' <- id m'` by the binding `m'' <- evl_xx m'` which has no effect because `m'` is the term delivered by `evl_xx` on the neutral's operator `m`, and certainly `id ∘ evl_xx = evl_xx ∘ evl_xx`.

Finally, simplify the `do` notation by applying the right identity of monadic `bind` and also Kleisli composition. The result is `eval_readback_xx3` below:

```
eval_readback_xx3 :: Red
eval_readback_xx3 t = case t of
  v@(Var _) -> return v
  (Lam s b) -> do b' <- (lar_xx <=< lae_xx) b
                return (Lam s b')
  (App m n) -> do m' <- evl_xx m
                case m' of
                  (Lam s b) -> do n' <- ar1e_xx n
                                (reb_xx <=< evl_xx) (subst n' s b)
                  _         -> do m'' <- (reb_xx <=< evl_xx) m'
                                n'   <- (ar2r_xx <=< ar2e_xx) n
                                return (App m'' n')
```

```
where evl_xx :: Red
      evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
      reb_xx :: Red
      reb_xx = gen_readback lar_xx ar2r_xx
```

- (5) Extrude the body of `eval_readback_xx3` by adding a new binding named `template` that takes three parameters `la`, `ar1` and `ar2`. The result is `eval_readback_xx4` below:

```
eval_readback_xx4 :: Red
eval_readback_xx4 =
  template (lar_xx <=< lae_xx) ar1e_xx (ar2r_xx <=< ar2e_xx)
  where
    evl_xx :: Red
    evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
    reb_xx :: Red
    reb_xx = gen_readback lar_xx ar2r_xx
    template :: Red -> Red -> Red -> Red
    template la ar1 ar2 t = case t of
      v@(Var _) -> return v
      (Lam s b) -> do b' <- la b
                    return (Lam s b')
      (App m n) -> do m' <- evl_xx m
                    case m' of
```

```

(Lam s b) -> do n' <- ar1 n
            (reb_xx <=< evl_xx) (subst n' s b)
_         -> do m'' <- (reb_xx <=< evl_xx) m'
            n' <- ar2 n
            return (App m'' n')

```

- (6) Replace the two calls to `(reb_xx <=< evl_xx)` in `template` by a call to a new binding `this` that is bound to the top-level call to `template`. The result is `eval_readback_xx5` below:

```

eval_readback_xx5 :: Red
eval_readback_xx5 = this
  where evl_xx :: Red
        evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
        reb_xx :: Red
        reb_xx = gen_readback lar_xx ar2r_xx
        template :: Red -> Red -> Red -> Red
        template la ar1 ar2 t = case t of
          v@(Var _) -> return v
          (Lam s b) -> do b' <- la b
                        return (Lam s b')
          (App m n) -> do m' <- evl_xx m
                        case m' of
                          (Lam s b) -> do n' <- ar1 n
                                          this (subst n' s b)
                          _         -> do m'' <- this m'
                                          n' <- ar2 n
                                          return (App m'' n')

        this :: Red
        this = template (lar_xx <=< lae_xx) ar1e_xx (ar2r_xx <=< ar2e_xx)

```

- (7) Promote `template` to a global function that keeps the local bindings for `evl_xx`, `reb_xx`, and `this`. The result is `eval_readback_xx6` below:

```

template :: Red -> Red -> Red -> Red
template la ar1 ar2 t = case t of
  v@(Var _) -> return v
  (Lam s b) -> do b' <- la b
                return (Lam s b')
  (App m n) -> do m' <- evl_xx m
                case m' of (Lam s b) -> do n' <- ar1 n
                                        this (subst n' s b)
                            _         -> do m'' <- this m'
                                        n' <- ar2 n
                                        return (App m'' n')

where evl_xx :: Red
      evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
      reb_xx :: Red
      reb_xx = gen_readback lar_xx ar2r_xx
      this   :: Red

```

```

    this = template (lar_xx <=< lae_xx) ar1e_xx (ar2r_xx <=< ar2e_xx)

eval_readback_xx6 :: Red
eval_readback_xx6 = this
  where evl_xx :: Red
        evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
        reb_xx :: Red
        reb_xx = gen_readback lar_xx ar2r_xx
        this   :: Red
        this   = template (lar_xx <=< lae_xx) ar1e_xx (ar2r_xx <=< ar2e_xx)

```

- (8) Add formal parameters `op1` and `op2` to `template` so that it has the same parameters as `gen_eval_apply`, and rename `template` to `template1`. In the call to `template1` instantiate `op1` to `evl_xx` and `op2` to `this`. Remove from `template1` the local bindings in the `where` clause which are now unused. The result is `eval_readback_xx7` below:

```

template1 :: Red -> Red -> Red -> Red -> Red -> Red
template1 la op1 ar1 op2 ar2 t = case t of
  v@(Var _) -> return v
  (Lam s b) -> do b' <- la b
                return (Lam s b')
  (App m n) -> do m' <- op1 m
                 case m' of (Lam s b) -> do n' <- ar1 n
                                           this (subst n' s b)
                             _         -> do m'' <- op2 m'
                                           n' <- ar2 n
                                           return (App m'' n')

  where this :: Red
        this = template1 la op1 ar1 op2 ar2

eval_readback_xx7 :: Red
eval_readback_xx7 = this
  where
    evl_xx :: Red
    evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
    reb_xx :: Red
    reb_xx = gen_readback lar_xx ar2r_xx
    this   :: Red
    this   =
      template1
        (lar_xx <=< lae_xx) evl_xx ar1e_xx this (ar2r_xx <=< ar2e_xx)

```

- (9) Replace `template1` by `gen_eval_apply` because they have identical definitions. The result is `eval_readback_xx8` below:

```

eval_readback_xx8 :: Red
eval_readback_xx8 = this
  where
    evl_xx :: Red
    evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx

```

```

reb_xx :: Red
reb_xx = gen_readback lar_xx ar2r_xx
this   :: Red
this   =
  gen_eval_apply
    (lar_xx <=< lae_xx) evl_xx ar1e_xx this (ar2r_xx <=< ar2e_xx)

```

The provisos in code for `eval_readback_xx` (page 51) constrain the possible values for `lar_xx` which can only stand for:

- (a) The identity `id`. In this case, the parameter `(lar_xx <=< lae_xx)` either stands for `(id <=< id)` or for `(id <=< evl_xx)`, each equivalent respectively to `id` and to `evl_xx`.
- (b) The value `reb_xx` if `lae_xx` stands for `evl_xx`, or the value `(reb_xx <=< evl_xx)` if `lae_xx` stands for `id`. Then, the parameter `(lar_xx <=< lae_xx)` stands for `(reb_xx <=< evl_xx)` or for `((reb_xx <=< evl_xx) <=< id)`, both equivalent to `(reb_xx <=< evl_xx)` and thus equivalent to a recursive call to `this`.

It is therefore safe to replace `lar_xx` in `eval_readback_xx8` by a new parameter `laer_xx` that stands for

- `id` when both `lar_xx` and `lae_xx` are `id`,
 - `evl_xx` when `lar_xx` is `id` and `lae_xx` is `evl_xx`,
 - `this` when `lar_xx` is either `reb_xx` or `(reb_xx <=< evl_xx)`,
- which meets the provisos for `bal_hybrid_xx` (page 52).

The same discussion applies to `(ar2r_xx <=< ar2e_xx)`. It is therefore also safe to replace `ar2e_xx` by a new parameter `ar2er_xx`.

- (10) By the discussion on the parameters in Step (9), replace `lar_xx` and `ar2r_xx` with `laer_xx` and `ar2er_xx` respectively. Remove the binding for `reb_xx` which is now unused. The result is `eval_readback_xx9` below.

```

eval_readback_xx9 :: Red
eval_readback_xx9 = this
  where evl_xx :: Red
        evl_xx = gen_eval_apply lae_xx evl_xx ar1e_xx id ar2e_xx
        this   :: Red
        this   = gen_eval_apply laer_xx evl_xx ar1e_xx this ar2er_xx

```

Compare it to `bal_hybrid_xx`. We copy-paste the definition here for convenience:

```

bal_hybrid_xx :: Red
bal_hybrid_xx = hyb_xx
  where sub_xx :: Red
        sub_xx = gen_eval_apply las_xx sub_xx ar1s_xx id ar2s_xx
        hyb_xx :: Red
        hyb_xx = gen_eval_apply lah_xx sub_xx ar1s_xx hyb_xx ar2h_xx

```

The code for `eval_readback_xx9` and `bal_hybrid_xx` coincide when:

- The identifiers `ar1e_xx` and `ar1s_xx` both stand for `id`, or for `eval_xx`, the latter the same evaluator defined by the hybrid's subsidiary evaluator `sub_xx`.

- Each of the remaining eval-readback parameters (`lae_xx`, `ar2e_xx`, `laer_xx` and `ar2er_xx`) has the same value as its homonymous hybrid parameters (`las_xx`, `ar2s_xx`, `lah_xx` and `ar2h_xx`) as explained in Step (9).
- The local variables `sub_xx` and `hyb_xx` are substituted respectively for `eval_xx` and `this`. This is a mere syntactic substitution.

The possible substitutions are collected in the following equations:

$$\begin{aligned}
 \text{lae_xx} &= \text{las_xx} \\
 \text{ar1e_xx} &= \text{ar1s_xx} \\
 \text{ar2e_xx} &= \text{ar2s_xx} \\
 \text{lar_xx} \leq\leq \text{lae_xx} &= \text{lah_xx} \\
 \text{ar2r_xx} \leq\leq \text{ar2e_xx} &= \text{ar2h_xx}
 \end{aligned}$$

The equations are not a part of the LWF steps, they collect the required substitutions of arbitrary parameters in Step 9. When instantiating the arbitrary evaluators with concrete parameters, the equations must be checked considering `reb_xx` $\leq\leq$ `evl_xx` and `hyb_xx` are one-step equivalent modulo commuting redexes (recall the examples in Figure 12).

9. SUMMARY AND CONCLUSIONS

Figure 13 (page 61) provides an extensive list of uniform and hybrid eval-apply evaluators. Figure 14 (page 62) provides an extensive list of eval-readback evaluators and their LWF-equivalent balanced hybrid eval-apply evaluators where ‘mcr’ abbreviates ‘modulo commuting redexes’.

There are 8 uniform eval-apply evaluators and 33 (four times more) hybrid eval-apply evaluators. Of the latter, one evaluator defines a uniform strategy ($\text{IIIH} \diamond \text{bn} = \text{IIS}$), and two evaluators (`no` and $\text{IIIH} \diamond \text{IIS}$) define equivalent strategies modulo commuting redexes (mcr). We find 4 uniform evaluators/strategies in the literature (`bn`, `he`, `bv`, `ao`). The remaining 4 are novel (`ho`, `IIS`, `SIS`, `ISI`) where the first two have interesting properties and uses. We find 6 hybrid evaluators in the literature that define 6 hybrid strategies (`hr`, `no`, `hn`, `am`, `sn`, `ha`). The remaining 26 evaluators are novel. Some define strategies with interesting properties and uses. In particular, `bs` and `so` can be used for eager evaluation of general recursive functions using delimited CPS and thunking protecting-by-variable. The other evaluators and strategies remain to be explored.

There are 22 eval-readback evaluators that are LWF-equivalent to 22 balanced hybrid eval-apply evaluators, 6 of them mcr. (There are more hybrid evaluators than eval-readback evaluators.) The equivalences mcr inform the equivalences mcr within the eval-readback style. For instance, $(\text{RE})\text{R} \circ \text{IIS}$ is equivalent mcr to $\text{IIIH} \diamond \text{IIS}$ and the latter to `no`, thus $(\text{RE})\text{R} \circ \text{IIS}$ is equivalent mcr to $(\text{RE})(\text{RE}) \circ \text{bn}$. One eval-readback evaluator $\text{I}(\text{RE}) \circ \text{bn}$ defines the uniform strategy `IIS`. We find 4 eval-readback evaluators in the literature (head reduction, normal order, `byName` / hybrid normal order, `byValue` / strong reduction). We have not found any of the remaining 18 eval-readback evaluators in the literature, but we have not searched exhaustively.

Figure 15 illustrates the position of our one-step equivalence proof within the wider context of correspondences between operational semantics devices. The top diagram in the figure is based on the diagrams in [DM09, p.545–547] and shows the addition of the space of big-step eval-readback evaluators that LWF-transform into the subspace of big-step balanced hybrid eval-apply evaluators. The so-called ‘functional correspondence’

Name	Systematic	Equivalent	Strictness	Classif.	Result
Call-by-name (bn)	III	IIH \diamond bn	non-strict	uniform	WHNF
	IIS		non-strict	uniform	WNF
Head spine (he)	SII	SIS	non-strict	uniform	HNF
	SIS		non-strict	uniform	NF
Call-by-value (bv)	ISI	ISS	strict	uniform	WHNF
	ISS		strict	uniform	WNF
Head applicative order (ho)	SSI	SSS	strict	uniform	HNF
Applicative order (ao)	SSS		strict	uniform	NF
Head reduction (hr)	IIH \diamond bn	IIS	non-strict	uniform	WNF
	SIH \diamond bn		non-strict	hybrid bal.	WNF
	HII \diamond bn	no (mcr)	non-strict	hybrid bal.	HNF
	HIS \diamond bn		non-strict	hybrid bal.	VHNF
Normal order (no)	HIH \diamond bn	HIH \diamond IIS (mcr)	non-strict	hybrid bal.	NF
	SIH \diamond IIS	no (mcr)	non-strict	hybrid bal.	WNF
Hybrid normal order (hn)	HIS \diamond IIS		no (mcr)	non-strict	hybrid bal.
	HIH \diamond IIS	non-strict		hybrid bal.	NF
	SIH \diamond he	no (mcr)	non-strict	hybrid bal.	WNF
	HIS \diamond he		non-strict	hybrid bal.	HNF
Hybrid normal order (hn)	HIH \diamond he	no (mcr)	non-strict	hybrid bal.	NF
	ISH \diamond ISI		strict	hybrid bal.	WNF
Ahead machine (am)	SSH \diamond ISI	no (mcr)	strict	hybrid bal.	WNF
	HSI \diamond ISI		strict	hybrid bal.	HNF
	HSS \diamond ISI		strict	hybrid bal.	VHNF
	HSH \diamond ISI		strict	hybrid bal.	NF
Strict normalisation (sn)	SSH \diamond bv	no (mcr)	strict	hybrid bal.	WNF
	HSS \diamond bv		strict	hybrid bal.	VHNF
Balanced spine applicative order (bs)	HSH \diamond bv	no (mcr)	strict	hybrid bal.	NF
	SSH \diamond ho		strict	hybrid bal.	WNF
	HSS \diamond ho		strict	hybrid bal.	HNF
Hybrid applicative order (ha)	HSH \diamond ho	no (mcr)	strict	hybrid bal.	NF
	IHH \diamond ISI		strict	hybrid	WNF
	SHH \diamond ISI		strict	hybrid	WNF
	HHI \diamond ISI		strict	hybrid	HNF
	HHS \diamond ISI		strict	hybrid	VHNF
Hybrid applicative order (ha)	HHH \diamond ISI	no (mcr)	strict	hybrid	NF
	SHH \diamond bv		strict	hybrid	WNF
	HHS \diamond bv		strict	hybrid	VHNF
Spine applicative order (so)	HHH \diamond bv	no (mcr)	strict	hybrid	NF
	SHH \diamond ho		strict	hybrid	WNF
Spine applicative order (so)	HHS \diamond ho	no (mcr)	strict	hybrid	HNF
	HHH \diamond ho		strict	hybrid	NF

Figure 13: List of uniform and hybrid eval-apply evaluators.

[Rey72,ABDM03,ADM05,DM09,DJZ11] begins at the space of direct-style big-step eval-apply evaluators (called ‘reduction-free normalisers’ in [DM09,Dan05]) which can be transformed into abstract machines by CPS transformation and defunctionalisation steps. Both steps are invertible by refunctionalisation and direct-style transformation. The abstract machines are transformed (compiled) to unstructured programs using GOTO statements to implement the control structures of the structured abstract machines.

The functional correspondence applies to arbitrary direct-style eval-apply evaluators, including uniform and unbalanced hybrid evaluators, which are located within the grey ring. The transformations in the diagram inject a source space into a target subspace, *e.g.*

Name	Systematic	LWF-equivalent eval-apply	Intermediate	Result
Head reduction	$I(RE) \circ bn$	$IIH \diamond bn (= IIS)$	WHNF	WNF
	$E(RE) \circ bn$	$SIH \diamond bn$	WHNF	WNF
	$(RE)I \circ bn$	hr	WHNF	HNF
	$(RE)E \circ bn$	$HIS \diamond bn$	WHNF	VHNF
	$(RE)(RE) \circ bn$	no	WHNF	NF
Normal order	$ER \circ IIS$	$SIH \diamond IIS (mcr)$	WNF	WNF
	$(RE)I \circ IIS$	$HIS \diamond IIS (mcr)$	WNF	VHNF
	$(RE)R \circ IIS$	$HIH \diamond IIS (mcr)$	WNF	NF
byName	$I(RE) \circ he$	$SIH \diamond he$	HNF	WNF
	$RE \circ he$	$HIS \diamond he$	HNF	HNF
	$R(RE) \circ he$	hn	HNF	NF
Ahead machine byValue	$I(RE) \circ ISI$	$ISH \diamond ISI$	WHNF	WNF
	$E(RE) \circ ISI$	$SSH \diamond ISI$	WHNF	WNF
	$(RE)I \circ ISI$	$HSI \diamond ISI$	WHNF	HNF
	$(RE)E \circ ISI$	$HSS \diamond ISI$	WHNF	VHNF
	$(RE)(RE) \circ ISI$	$HSH \diamond ISI$	WHNF	NF
Balanced spine applicative order	$ER \circ bv$	$SSH \diamond bv (mcr)$	WNF	WNF
	$(RE)I \circ bv$	am (mcr)	WNF	VHNF
	$(RE)R \circ bv$	sn (mcr)	WNF	NF
Balanced spine applicative order	$I(RE) \circ ho$	$SSH \diamond ho$	HNF	WNF
	$RE \circ ho$	$HSS \diamond ho$	HNF	HNF
	$R(RE) \circ ho$	bs	HNF	NF

Figure 14: List of eval-readback evaluators and their one-step equivalent eval-apply evaluators.

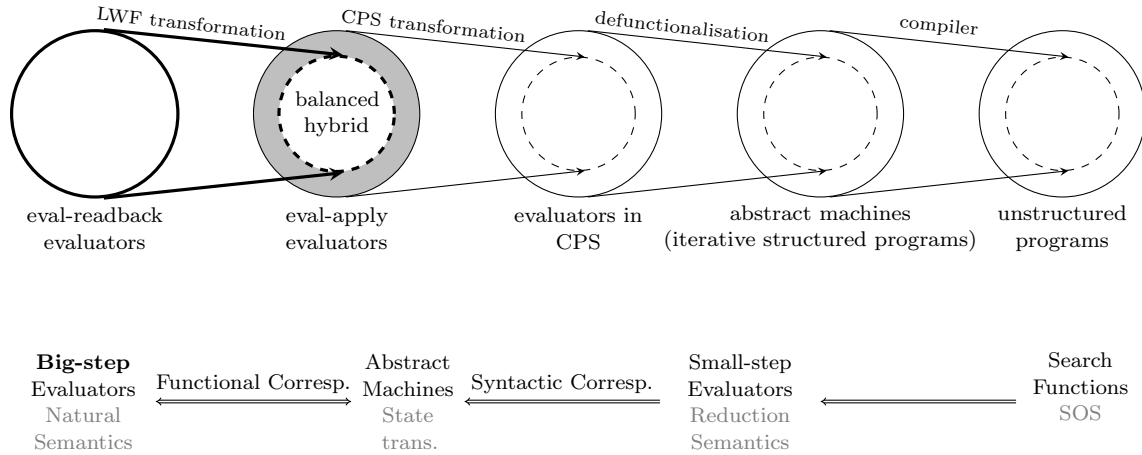


Figure 15: *Top*: Functional correspondence diagram based on [DM09, p. 547] and prefixed with the LWF transformation of eval-readback evaluators into balanced hybrid eval-apply evaluators. *Bottom*: Correspondence diagram between operational semantics devices in [GPN14, p. 177]. The one-step equivalence proof presented in this paper extends the functional correspondence to the left for big-step eval-readback evaluators.

not every evaluator in CPS functionally-corresponds with a source direct-style eval-apply evaluator, but those that do can be transformed back and forth.

As elegantly argued in [DM09, pp. 545–547], straying from the image of the transformation (staying within the rings of more expressive power) with good reasons is a clear indication that a useful control structure or operator may be missing in the source space. For instance, programmers write evaluators in CPS with non-tail calls ‘thereby introducing the notion of delimited control [...]; this effect can be obtained in direct style [in the source space] by adding [...] delimited-control operators [...]’ [DM09, p. 546]. Similarly in the space of structured programs, the case against the GOTO statement in [Dij68] is not that it is unconditionally harmful but that ‘straying with good reason is a clear indication that a useful control construct is missing in the source language. For example, C and Pascal programmers condone the use of GOTO for error cases because these languages lack an exception mechanism’ [DM09, p. 546].

There may be good reasons for straying from the image of the LWF transformation in the case of unbalanced hybrid strategies, such as hybrid applicative order (ha) and spine applicative order (so). Readback is a selective-iteration-of-eval function. An unbalanced strategy requires more work than eval on operands of redexes beyond distributing eval on abstractions and neutrals. There may be a missing control structure X that enables the equivalence between eval-readback-with- X and arbitrary hybrid eval-apply evaluators.

The bottom diagram in Figure 15 shows a wider correspondence diagram between operational semantics devices [GPN14, p. 177]. The transformation steps employed by the correspondences are semantics-preserving and the implementation of the devices in code define the same evaluation strategy. The figure shows on the left the functional correspondence zoomed out. On the right it shows the so-called ‘syntactic correspondence’ to abstract machines from small-step evaluators that implement context-based reduction semantics [Dan05, BD07, DM08, DJZ11]. The small-step evaluator (called a ‘reduction-based normaliser’ in [Dan05]) is a loop that iterates the following steps: (i) the decomposition of the term into a unique context with the permissible redex within the hole, (ii) the contraction, and (iii) the recomposition of the contractum into a term. The evaluator transforms to an abstract machine by the following non-invertible steps: refocusing (which optimises the iteration loop), LWF to fuse decomposition and recomposition, and inlining of iteration. Notice the different use of LWF to which we have already pointed to in the introduction. There is a further unnamed transformation from small-step structural operational semantics (SOS) to small-step context-based reduction semantics witnessed by the ‘search function’ that implements the compatibility rules of the SOS that traverse a term to locate the redex. An implementation of a search function transforms to the small-step evaluator by the following non-invertible steps: CPS-transformation, simplification, defunctionalisation, and turning the search function into a decomposition function which, additionally to the next redex, delivers the context. The simplification and defunctionalisation of the search function reveals the continuation stack, which is not revealed by a whole implementation of a SOS that searches the input term, contracts the redex, and delivers the next reduct.

Syntactic correspondences have been shown for specific uniform and hybrid strategies, weak-reducing and full-reducing, and connected through the functional correspondence with big-step eval-apply evaluators [Mun08, GPN13, DMM13, GPNM13, DJ13, GPN14, BCZ17]. The further connection with eval-readback evaluators is provided by our one-step equivalence proof. The eval-readback evaluators are obtained from balanced hybrid eval-apply evaluators by instantiating the equations (Proposition 8.1).

10. RELATED WORK

We have discussed related work throughout the paper, especially in the introduction and Sections 3, 5, 7.2, and Appendix B. Big-step evaluators are ubiquitous in the literature on lambda calculus and functional programming. We have not found higher-order generic evaluators like those of Section 7.1 in the literature, but we believe they must have been considered before.

The survey and structuring of the strategy space presented in this paper significantly emend and amend the seminal survey and structuring presented in [GPNG10, Gar14] which in turn extend the survey and structuring in [Ses02]. The LWF transformation presented in this paper on plain arbitrary evaluators that instantiate generic evaluators is simpler and more direct than the LWF transformation in [Gar14] on generic evaluators that work with boolean triples and the hybridisation operation (\diamond).

A recent work [BCD22] presents a strategy space (called a zoo) that includes call-by-name, call-by-value (call-by-WNF and lambda-value's), head and full-reducing versions, and non-deterministic and right-to-left versions. The work focuses on small-step context-based reduction semantics to define strategies, using defunctionalised continuations that include contexts and other useful information. It connects the uniform-style property (grammars with one non-terminal, *i.e.* one continuation stack) with determinism, and it structures the space by building strategies from (or decomposing strategies into) combinations of basic fine-grained sets of contexts (weak contexts, head contexts, left or right contexts, etc.) and subsets of specific terms (intermediate and final results). A hybrid strategy as presented in this paper is such that it has contexts in one (maybe more) of its building fine-grained contexts not closed under inclusion. The idea of combining fine-grained contexts is related to the notion of dependence [GPN14]. Despite the differences between our approach and [BCD22], both find and structure many identical left-to-right deterministic strategies: all the well-known strategies and the novel uniform ones presented in [GPNG10, Gar14], *e.g.* IIS and ISI are respectively *low* and *cbwh* in [BCD22]. The zoo does not consider many hybrid strategies (*am*, *ha*, *so*, *bs*, and the unnamed hybrids in Figure 13), but both strategy spaces might be unified or superseded by extending the catalogue of basic fine-grained contexts with those required to define, through their combination, the full hybrid space. We take this as a confirmation that the notion of uniform/hybrid strategy is a fruitful structuring criterion.

Hybrid strategies abound. They are essential to completeness and ‘completeness for’ in the non-strict and strict spaces. For the non-strict strategies, the underlying explanatory technical notion is ‘needed reduction’ (Appendix E) which can be extended to ‘completeness for’ other kinds of final result than normal forms. We summarise the completeness for some uniform and balanced hybrid strategies in Figure 6 (page 30):

- *bn*, *IIS*, *hr*, and *no* are respectively complete for *WHNF*, *WNF*, *HNF*, and *NF* by the Quasi-Leftmost Reduction Theorem (Appendix E).
- *he* and *hn* are respectively complete for *HNF* and *NF*, with redexes in $(\lambda x.HNF)\Lambda$ by the Needed Reduction Theorem (Appendix E).¹⁶

Completeness results can be given for strict strategies (which are incomplete in the pure lambda calculus) using notions of reduction other than the β -rule, namely, using calculi with

¹⁶By the properties of *HNF*, it turns out that restricting the redexes to $(\lambda x.HNF)\Lambda$ does not alter the completeness of *he* and *hn* with unrestricted redexes.

notions of reduction with permissible redexes where the operand and/or the operator has to be in some final form. Concretely, we conjecture the following:

- **bv**, **am**, and **sn** are respectively complete for **WNF**, **VHNF**, and **NF** with redexes in $(\lambda x.\Lambda)\text{WNF}$.
- **ho** and **bs** are respectively complete for **HNF** and **NF** with redexes in $(\lambda x.\text{HNF})\text{HNF}$.

The conjectures rest on these principles:

- (1) The strategies above are outermost when taking into account the corresponding notion of permissible redex, *i.e.* an outermost non-permissible redex may become the next permissible redex to be contracted iff contracting some inner permissible redex makes the outer one a permissible redex.
- (2) All the notions of permissible redexes are stable under contraction, *i.e.* a permissible redex remains so when any of its permissible sub-redexes is contracted.
- (3) Contraction of permissible redexes is confluent, *i.e.* given a term with two permissible redexes, the terms resulting after the contraction of the two permissible redexes in different order reduce to a common term.

We finally discuss the connection between hybridity, outermost, completeness-for, ‘refocusing’, and ‘backward overlaps’ [DJ13]. A set of reduction rules is backward overlapping if a subterm of the left hand side of one rule unifies with the right hand side of the same or a different rule. Intuitively, this means that a potential redex (the term that will unify with the left hand side of the first rule) will turn into an actual redex after contracting some sub-redex (the redex whose contractum is the right hand side of the second rule) of the former one. As pointed out in [DJ13], in the presence of backward overlaps the ‘refocusing’ techniques that deforest the recomposition and decomposition of a contractum in a context-based reduction semantics may miss the outer redex when this context-based reduction semantics implements an outermost strategy (*i.e.* refocusing will carry on decomposing the term at the scope of the inner redex, instead of recomposing the term up to the scope of the outer redex). In order to prevent this, [DJ13] propose a backtracking step that will recompose the term just enough to reach the scope of the outer redex.

All the strategies discussed in the paragraph above about ‘completeness-for’ are outermost, and the notions of reduction induced by the corresponding permissible redexes are backward overlapping. (An outer non-permissible redex may turn into a permissible redex by contracting some inner permissible redex.) Some of the strategies are uniform, because their permissible redexes do not contain any permissible sub-redexes to be contracted according to the strategy, and the problem of missing an outer redex by decomposing at the scope of the inner redex never occurs. This is easy to see in the non-strict case, where the operator $(\lambda x.\Lambda)$ in **WHNF** is irreducible for **bn** and **IIS**, and the operator $(\lambda x.\text{HNF})$ in **HNF** is irreducible for **he**, and where the operands of a permissible redex are never reduced. It can also be seen in the strict case, where the operator $(\lambda x.\Lambda)$ in **WNF** and the operand in **WNF** are irreducible for **bv**, and where the operator $(\lambda x.\text{HNF})$ in **HNF** and the operand in **HNF** are irreducible for **ho**. The remaining outermost and complete-for strategies are inexorably hybrid because their permissible redexes may contain permissible sub-redexes. In this case, a less-reducing subsidiary strategy must be called at the scope of the potential redex (at the operator in the non-strict case, and at the operator and the operand in the strict case) in order to find the inner redexes that will turn the outermost potential redex into an actual one, but without reducing any further. In [GPN13], a refocusing solution based on the shape invariant of the continuation stack used in the decomposition step is shown to be applicable to such hybrid

strategies. This solution is also applicable to the outermost strategy presented in [DJ13], which is hybrid.

11. FUTURE WORK

We list a few of many possibilities for future work. To analyse the strategies that stray from the image of the LWF transformation and their possible relation with control operators mentioned in the conclusion. To investigate one-step equivalence when the subsidiary evaluator implements a shallow hybrid strategy (Section 7.4.1). To investigate the relationship between, on the one side, non-strict leftmost-outermost and spine strategies, and on the other side, strict leftmost-innermost and spine-ish strategies, especially in the wider context of ‘needed reduction’ (Appendix E). To (dis)prove the duality conjecture (Section 5.14). To transcribe the structuring of the strategy space to the search-function side of the syntactic correspondence in order to derive plain but arbitrary small-step evaluators and abstract machines that connect to the functional correspondence. To investigate the connection with the small-step framework in [BCD22]. To mechanise the LWF proof involving constraints on parameters. To develop the functional correspondence from parametric eval-apply evaluators into parametric abstract machines. To apply the principles and criteria presented in this paper to structuring the strategy space of calculi with explicit substitutions.

ACKNOWLEDGEMENTS

Our interest in strategies and evaluators was stimulated by the textbooks and papers cited in the introduction. The following were particularly influential: [FH88, Chp. 9], [Rea89, Chp. 13], [Mac90, Chps. 11–12], and [McC60, Lan64, Rey72, Plo75, Ses02, GL02, ABDM03]. We thank Emilio J. Gallego Arias for contributing to [GPNG10]. We thank the following people for their feedback and encouragement at talks and presentations on the survey and structuring material in [GPNG10, Gar14]: Olivier Danvy, Jan Midtgaard, Ian Zerny, Peter Sestoft, Ruy Ley-Wild, Noam Zeilberger, Richard Bird, Jeremy Gibbons, Ralf Hinze, Geraint Jones, Bruno Oliveira, Meng Wang, and Nicolas Wu. We specially thank Jeremy Gibbons for permission to name the Beta Cube after him. We thank Tomasz Drab for a discussion on the strategy space in [BCD22] and the strategy space presented in this paper. We finally thank the three anonymous reviewers of *LMCS* for accepting to review such a long paper and for their insightful criticism and suggestions which have led us to make important corrections, clarifications, and additions.

APPENDIX A. PURE LAMBDA CALCULUS CONCEPTS AND NOTATION

The set Λ of pure lambda calculus terms consists of variables, abstractions, and applications. Uppercase, possibly primed, letters (*e.g.* B, M, N, M', M'' , etc.) range over Λ . Lowercase, also possibly primed, letters (*e.g.* x, y, z, x', x'' , etc.) range over a countably-infinite set of variables V . Lowercase variables act also as *meta-variables* ranging over V when used in term forms. For example, the concrete term $\lambda y.zy$ with parameter y is of the form $\lambda x.B$ where x is a meta-variable. We say x is the parameter and B is the *body*. A *free variable* is not a parameter of an abstraction. Terms with at least one free variable are *open terms*. Terms without free variables are *closed terms*. In an application MN we say M is the *operator* and N is the *operand*. The application operation is denoted by tiny

whitespace or by juxtaposition. Abstractions associate to the right. Applications associate to the left and bind stronger than abstractions. A few example terms: x , xy , xx , $\lambda x.x$, $\lambda x.\lambda y.yx$, $x(\lambda x.x)$, $xy(\lambda x.x)$, $(\lambda y.y)(xy)$, and $(\lambda x.y)(xyz)(\lambda x.\lambda y.xy)$. Uppercase boldface letters stand for concrete terms. For example, **I** abbreviates the identity term $\lambda x.x$, and **Y** abbreviates the non-strict fixed-point combinator $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. For more fixed-point combinators and their explanation see [Ros50, p. 130], [CF58, p. 178], [HS08, p. 36], and [Bar84, p. 131].

Every term of Λ has the form $\lambda x_1 \dots \lambda x_n.HN_1 \dots N_m$ with $n, m \geq 0$. The head term H is either a variable (the ‘head variable’) or a redex (the ‘head redex’). A *redex* is an application of the form $(\lambda x.B)N$. We write an arrow ‘ \rightarrow ’ to denote one-step beta-reduction. We write $[N/x](B)$ for the *capture-avoiding substitution* of the operand N for the free occurrences (if any) of parameter x in the abstraction body B . The resulting term is called a *contractum*. The function notation $[N/x](B)$ that wraps the subject of the substitution B in parentheses comes from [Plo75]. This notation is itself based on the original notation $[N/x]B$ without parenthesis of [CF58]. Both notations flow in English when read from left to right: ‘substitute N for free x in B ’. The action is substitute-for, not replace-by. Other notations are, of course, valid.

Normal forms, collected in set **NF** (Figure 1), are terms without redexes. They consist of variables, and abstractions and applications in normal form. *Neutrals* have been introduced in Section 1.3. Examples of neutrals are xy , $x(\lambda x.y)$, and $x((\lambda x.y)z)(y(\lambda x.x))$. The last neutral has a neutral $y(\lambda x.x)$ as its second operand. A neutral cannot evaluate to a redex unless a suitable operand can be substituted for the head variable. For this, it is necessary for the neutral to occur within an abstraction that has the head variable for parameter. For example, in the neutral xN the head variable is free and cannot be substituted. In $(\lambda x.xN)(yM)$, the abstraction body xN is a neutral, but the substitution of yM for x delivers yMN which is itself a neutral, not a redex. In contrast, in $(\lambda x.xN)(\lambda y.y)$ the contractum $(\lambda y.y)N$ is a redex.

An evaluation *strategy* is defined generically as a function subset of the reduction relation, e.g. [Bar84, p. 53] [TeR03, p. 130]. For instance, a one-step strategy **st** is defined as a total function on terms such that either $M \equiv \text{st}(M)$ for all $M \in \mathbf{F}$ where \mathbf{F} is a set of final forms, or $M \rightarrow \text{st}(M)$ for all $M \notin \mathbf{F}$. An iterative multiple-step definition is obtained by transitive closure: $M \rightarrow^+ \text{st}^n(M)$ for $n > 0$. For each input term the strategy determines a unique *evaluation sequence* up to, if finite, a final result in \mathbf{F} . The strategy *converges* (opposite, *diverges*) when the evaluation sequence is finite (opposite, infinite). The term is said to converge (opposite, diverge) under the strategy. The big-step notation, $\text{st}(M) = N$, can be translated in terms of one-step **st** as follows: either $M \in \mathbf{F}$ and $\text{st}(M) \equiv M \equiv N$, or $M \notin \mathbf{F}$ and $M \rightarrow^+ \text{st}^n(M)$ and $\text{st}^n(M) \equiv N$ for $n > 0$ and $N \in \mathbf{F}$.

A generic definition of a strategy is uninformative because the particular redex contracted in the reduction is unspecified. We illustrate the evaluation sequence of a strategy by underlining the chosen redex: $(\lambda x.\lambda y.xy)(\lambda x.z)x \rightarrow (\lambda y.(\lambda x.z)y)x \rightarrow (\lambda x.z)x \rightarrow z$. The second term in the sequence has another redex, $(\lambda y.(\lambda x.z)y)x$, which is not chosen by the strategy. In this example a final result is a normal form. Operational semantics styles [Plo81, Lan64, Kah87, FH92] define strategies precisely by stipulating how the chosen redex is located.

In the pure lambda calculus it is undecidable whether an arbitrary term can be evaluated to normal form. A strategy that aims at delivering normal forms is *full-reducing*. A strategy that delivers a normal form when the latter exists is *normalising*. A strategy that diverges

for terms for which a normal form exists is *not normalising*. The archetypal term without a normal form is $(\lambda x.xx)(\lambda x.xx)$ which is abbreviated as Ω . The word *complete* (opposite, *incomplete*) is also used in the literature as a synonym of ‘normalising’. We have not been able to trace the genesis of ‘complete’. We think it comes from the idea that the strategy’s evaluation sequence completes to (reaches) the final form, or perhaps from extending to the strategy the ‘complete’ property of the reduction relation which obtains when it is confluent and terminating [TeR03, p. 13]. A strategy determines a unique evaluation sequence so, if it converges in a final form, then it may be called complete by extension. We prefer ‘complete’ because of its generality: final forms need not be exclusively normal forms. Following an anonymous reviewer’s suggestion, we say that a strategy is ‘complete for’ the particular set of terms.

APPENDIX B. CALL-BY-VALUE IN THE LAMBDA-VALUE CALCULUS

The motivation for the lambda-value calculus of [Plo75] was to prove that the SECD abstract machine of the ISWIM language [Lan65] was correct with respect to a calculus with open and divergent terms and a reduction relation with blocking, and provide a standardisation theorem for a complete strategy with blocking that delivers values as results. The blocking of neutrals guarantees the confluence of the reduction relation.

The correctness of SECD is proven with respect to the big-step evaluator ‘eval_V’ [Plo75, p. 130] and the small-step evaluator ‘→_V’ [Plo75, p. 136] by (1) a standardisation theorem [Plo75, Thm. 3, p. 137] that proves that ‘→_V’ is complete for values, (2) by the one-step equivalence between ‘eval_V’ and ‘→_V’ (proven on paper [Plo75, Thm. 4, p. 142], not by program transformation), and (3) by the one-step equivalence of ‘eval_V’ and SECD (also proven on paper [Plo75, Thm. 1, p. 131], not by program transformation). The proofs of (2) and (3) assume *closed input terms*. Indeed, the evaluators do not correspond for open input terms because ‘eval_V’ ignores neutrals whereas ‘→_V’ evaluates the first operand of a neutral in its 4th clause. Stuck terms do not arise with closed input terms. Under this assumption, ‘eval_V’ and ‘→_V’ implement call-by-WNF (Section 3). With open terms, call-by-WNF differs from ‘→_V’ and cannot even be compared to ‘eval_V’ because the latter does not consider neutrals.

Discussions about the pure version of lambda-value, its operational and denotational theory, and its full-reducing strategies can be found among others in [EHR91, EHR92, PR99, RP04, AP12, CG14, GPN16, KMR21, AG22]. Normal forms with stuck terms and a standardisation theorem for a full-reducing and complete strategy were not considered in [Plo75]. The issues have been addressed among others in [PR99, RP04, AP12, CG14, GPN16, KMR21, AG22]. A standardisation theorem for a full-reducing complete strategy that delivers lambda-value normal forms is given in [RP04, p. 70]. A standardisation theorem for a variant full-reducing strategy that evaluates blocks left-to-right is given in [GPN16, Sec. 7.1] along with a standard theory based on the operational relevance (‘solvability’) and sequentiality of some open neutrals and stuck normal forms. The relation between lambda-value and the sequentiality of the linear lambda calculus was first studied in [MOTW95]. For the most complete up-to-date work at the time of writing on lambda-value solvability see [AG22], and the references on standardisation there cited.

APPENDIX C. OVERVIEW OF LWF

The LWF optimisation is outlined in [OS07, p. 144] using code written in the maths font we have used in this paper for lambda calculus terms. To avoid confusion, in this appendix we use the same code symbols as [OS07] but in typewriter font, which is appropriate because LWF is applied to code.

Fusion is a program-transformation optimisation that combines a composition $f \circ g$ into a single function h . LWF is a general fusion optimisation where f and g can be general recursive functions, of any algebraic type, and with any number of curried or uncurried parameters. LWF is lightweight because it consists of a few simple, terminating, and automatable steps that perform local syntactic transformations without the need for heuristics. Thus, LWF can be embedded in the inlining optimisation of a compiler.

Given two general recursive functions $f = \text{fix } f.\lambda x.E_f$ and $g = \text{fix } g.\lambda x.E_g$, the composition $f \circ g$ is fused by promoting f through g 's fixed-point operator fix in the following steps.

- (1) Inline g in $f \circ g$ to obtain $f \circ (\lambda x.E_g)$ and then beta-convert that to $\lambda x.f E_g$.
- (2) Transform the application $f E_g$ to E'_g by distributing f to tail position in E_g .
- (3) Inline f once in E'_g and simplify to obtain $E_{f,g}$.
- (4) Replace the occurrences of $f \circ g$ in $E_{f,g}$ by a new function name h and generate a new binding $h = \text{fix } h.\lambda x.E_h$.

Because the first steps pull out g 's case expression, the transformation requires f to be strict (to pattern-match) on its first argument.

The soundness of LWF is proven in [OS07, Sec. 3] assuming a ‘standard call-by-name’ denotational semantics for a small core functional programming language where f and g are least fixed points in the space of continuous functions. We believe that semantics is assumed for theoretical simplicity and reasons of space in a conference paper because LWF is used on code written in *both* lazy (non-strict, call-by-need, *e.g.* Haskell) and eager (strict, call-by-value, *e.g.* Standard ML) functional programming languages. Because a denotational soundness proof for the latter would depend on the particular call-by-value theory of choice (*e.g.* Appendix B), the authors left that proof as future work and suggested one possible choice in [OS07, Sec. 9].

An alternative would have been to prove the call-by-value operational (rather than denotational) soundness of the steps, but that would have also enlarged the paper. At any rate, such correctness is assumed by LWF’s syntactic simplicity and by the empirical evidence provided by the correct results obtained in the ‘expected typical cases’ [OS07, p. 144], where the fused program h is syntactically identical to a known existing correct solution h' . The code examples with correctly fused results in [OS07] are written in strict Standard ML which certainly has no call-by-name denotational semantics. Applications of LWF in strict Standard ML presented in subsequent work [Dan05, DM08, GPN14] confirm the soundness of the transformation by producing fused programs which are syntactically identical to the expected correct known ones.

The only caveat on soundness, discussed in [OS07, Sec. 6.3], occurs when there is an expression in between a binding $x = g e$ and the application $f x$. These two must be placed together to fuse them into $h e$, and this may require swapping the in-between expression, changing the evaluation order. For pure code, whether in lazy or eager functional programming languages, the swapping may affect divergence order and performance in particular cases [OS07, p. 152]. When the swapped expressions have side-effects, $f \circ g$ and h

would differ in side-effect order. It is for this reason that [OS07, p. 151, 154] state that LWF is unsound for *strict and impure languages*. However, the problem affects *impure code* in any language (the authors seem to consider only the pure subset of Haskell without impure features [OS07, p. 151]). When there are no in-between bindings or the impact of the change in evaluation order (divergence and side-effects) is factored in and taken into account, the LWF transformation is unproblematically applied in lazy and eager languages for strict and non-strict functions.

In our proof by LWF (Section 8.4) a change of evaluation order does occur in the swapping of bindings for neutral terms in Step (4). This change only affects the evaluation order of redexes in operands N_i of neutrals $xN_1 \cdots N_n$. Such redexes are non-overlapping (commuting) and the change in evaluation order means the one-step equivalence is ‘modulo commuting redexes’ for some strategies. The use of impure `I0` in the Haskell code is consistent with this because the result term is printed as a side-effect, showing an equivalence modulo commuting those redexes.

The LWF transformation does not involve eta-conversion, $(\lambda x. f x) \rightarrow f$, nor eta-expansion, $f \rightarrow (\lambda x. f x)$, which is fortunate, as the first is sensitive to types [JG95] and the second introduces the equation $\perp = \lambda x. \perp$.

APPENDIX D. HASKELL’S OPERATIONAL SEMANTICS

Haskell was deliberately conceived as a research language that should avoid ‘success’. This included avoiding standardisation and a formal semantics that would either tie up the language or would have to keep up with its fast evolution [HHPW07]. However, there is a Haskell 2010 standard, *The Haskell 2010 Language Report* (available online), that includes a static semantics and also an operational semantics, as we clarify below. The latest version of the language is known as ‘GHC Haskell’, namely, the latest version compiled by the Glorious Haskell Compiler which is the *de facto* compiler. The static and operational semantics of GHC Haskell are scattered in research papers, some of them listed on the Haskell website and wiki pages.

Haskell 2010 has a standard operational semantics. More precisely, any version of Haskell has an implementation-based operational semantics. Haskell has a stable small core language, System FC, which is a direct-style (in opposition to CPS-style) version of the well-known explicitly-typed System F with extra support for type casts and coercions [Eis20]. Haskell code is statically type-checked and desugared into System FC code on which GHC performs the optimisations and generates target code for the STG abstract machine [Pey92]. This code is, in turn, finally compiled to C or to native machine code.¹⁷ System FC has an operational semantics given by its compilation scheme and execution on the STG machine. Whatever the version of Haskell used, what needs to be explained is the static semantics and the translation to System FC. These are provided for Haskell 2010 by *The Haskell 2010 Language Report* and the GHC implementation of the 2010 standard. In this sense, Haskell 2010 has a well-defined operational semantics.

With the exception of the ‘arbitrary rank polymorphism’ (`RankNTypes`) extension, the Haskell code in this paper conforms to the Haskell 2010 standard. Type classes for ad-hoc polymorphism are supported by the standard, with the `Monad` type class and the `I0` instance shipped with the Haskell 2010 standard library. Type classes have a well-known

¹⁷See Chp. 10 of the GHC Users Guide (visited: July 2023).

and simple dictionary semantics [HHPW96]. The `RankNTypes` extension enables GHC to type-check the type of our plain and generic evaluators which have a `forall` bounded on a monad type parameter. This extension has no run-time effect as it only allows the code to type-check. The Haskell 2010 standard uses a type-inference algorithm that cannot infer rank-polymorphic types whose inhabitants are representable in System FC. Such types are innocuous, type-checkable, and translatable to System FC. At any rate, in our case the extension is entirely optional and can be dropped if desired.

The use of strict monads for strict evaluation does require some elaboration. Haskell evaluates expressions to a weak head normal form. The 2010 standard provides features for strict evaluation to head normal form: `seq`, strict function application (`$!`), and strict value constructor application (or strict data-types), all of them discussed in Sections 4.2.1 and 6.2 of *The Haskell 2010 Language Report*. These features are partly reconcilable with parametricity theorems and the correctness of some fusion transformations [JV04]. GHC also offers extensions for evaluation to normal form (`deepseq` and fully evaluated data-types). However, we have used a non-strict data-type `Term` and the strict `IO` monad to fully evaluate Haskell expressions (evaluator calls) to print terms.

APPENDIX E. COMPLETENESS OF LEFTMOST AND SPINE STRATEGIES

The completeness and ‘completeness for’ of the leftmost strategies is supported by the Standardisation Theorem [CF58] which states that a strategy that chooses the leftmost redex (the leftmost-outermost redex when considering the abstract syntax tree of the term) is complete. Strict strategies are thus incomplete because they evaluate the operand of a leftmost-outermost redex.

The Standardisation Theorem is relaxed by the Quasi-Leftmost-Reduction Theorem [HS08, p. 40] which states that a strategy that *eventually* chooses the leftmost-outermost redex is complete. Such strategies are called *quasi-leftmost*. They subsume the leftmost and spine strategies, whose completeness and completeness-for is supported by the theorem.

The Standardisation and Quasi-Leftmost-Reduction theorems are further relaxed by the ‘needed reduction’ theorem which states that a strategy that eventually chooses the ‘needed’ redexes is complete [BKKS87, p. 208]. A needed redex is a redex that is reduced in every reduction sequence to normal form when the latter exists. Needed reduction eventually reduces all the needed redexes (or its residuals). Intuitively, the critical point is in applications MN where the reduction of M or N may diverge. The operator M should be reduced enough to deliver an abstraction. The operand N should not be reduced because it could be discarded. Only terms that eventually appear in operator position need be evaluated, and just enough (to head normal form) to contract the redex.

REFERENCES

- [ABDM03] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 8–19. ACM Press, 2003. doi:10.1145/888251.888254.
- [ABM14] B. Accattoli, P. Barenbaum, and D. Mazza. Distilling abstract machines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 363–376. ACM Press, 2014. doi:10.1145/2628136.2628154.

- [ABM15] B. Accattoli, P. Barenbaum, and D. Mazza. A strong distillery. In *Proceedings of the 13th Asian Symposium on Programming and Systems*, volume 9458 of *LNCS*, pages 231–250. Springer, 2015. doi:10.1007/978-3-319-26529-2_13.
- [Abr90] S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, 1990.
- [ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. doi:10.1017/S0956796800000186.
- [ACP⁺08] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–15. ACM Press, 2008. doi:10.1145/1328438.1328443.
- [ADM05] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. doi:10.1016/j.tcs.2005.06.008.
- [AG98] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45. Cambridge University Press, 1998.
- [AG22] B. Accattoli and G. Guerrieri. The theory of call-by-value solvability. *Proceedings of the ACM on Programming Languages*, 6(ICFP):855–885, 2022. doi:10.1145/3547652.
- [AL16] B. Accattoli and U. Dal Lago. (Leftmost-outermost) beta reduction is invariant, indeed. *Logical Methods in Computer Science*, 12(1), 2016. doi:10.2168/LMCS-12(1:4)2016.
- [All78] J.R. Allen. *Anatomy of Lisp*. McGraw-Hill, 1978.
- [ALV21] B. Accattoli, U. Dal Lago, and G. Vanoni. The (in) efficiency of interaction. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–33, 2021. doi:10.1145/3434332.
- [AP12] B. Accattoli and L. Paolini. Call-by-value solvability, revisited. In *Proceedings of the 11th International Symposium on Functional and Logic Programming*, volume 7294 of *LNCS*, pages 4–16, 2012. doi:10.1007/978-3-642-29822-6_4.
- [ASS85] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1st edition, 1985.
- [Bar84] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1984.
- [Bar91] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991. doi:10.1017/S0956796800020025.
- [BBCD20] M. Biernacka, D. Biernacki, W. Charatonik, and T. Drab. An abstract machine for strong call by value. In *Proceedings of the 18th Asian Symposium on Programming Languages and Systems*, volume 12470 of *LNCS*, pages 147–166. Springer, 2020. doi:10.1007/978-3-030-64437-6_8.
- [BBD05] M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2), 2005. doi:10.2168/LMCS-1(2:5)2005.
- [BC19] M. Biernacka and W. Charatonik. Deriving an abstract machine for strong call by need. In *Proceedings of the 4th International Conference on Formal Structures for Computation and Deduction*, volume 131, pages 8:1–8:20. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.8.
- [BCD21] M. Biernacka, W. Charatonik, and T. Drab. A derived reasonable abstract machine for strong call by value. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming*, pages 1–14. ACM Press, 2021. doi:10.1145/3479394.3479401.
- [BCD22] M. Biernacka, W. Charatonik, and T. Drab. The zoo of lambda-calculus reduction strategies, and Coq. In *Proceedings of the 13th International Conference on Interactive Theorem Proving*, pages 7:1–7:19, 2022. doi:10.4230/LIPIcs.ITP.2022.7.
- [BCZ17] M. Biernacka, W. Charatonik, and K. Zielinska. Generalized refocusing: from hybrid strategies to abstract machines. In *Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction*, volume 84, pages 10:1–10:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.FSCD.2017.10.
- [BD07] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):6:1–6:30, December 2007. doi:10.1145/1297658.1297664.
- [BG⁺92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In *IFIP TC10/WG10.2 International*

- Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland/Elsevier, 1992.
- [BKKS87] H. P. Barendregt, J. R. Kennaway, J. W. Klop, and M. R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75:191–231, 1987. doi:10.1016/0890-5401(87)90001-0.
- [BS91] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ -calculus. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, 1991. doi:10.1109/LICS.1991.151645.
- [Car97] L. Cardelli. Type systems. In *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997. Revised 2004.
- [CCM84] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 50–64. Springer, 1984. doi:10.1007/3-540-15975-4_29.
- [CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.
- [CG14] A. Carraro and G. Guerrieri. A semantical and operational account of call-by-value solvability. In *Proceedings of the 17th International Conference on Foundations of Software Science and Computation Structures*, pages 103–118. Springer, 2014. doi:10.1007/978-3-642-54830-7_7.
- [CH00] P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, volume 35(9) of *ACM SIGPLAN Notices*, pages 233–243. ACM Press, 2000. doi:10.1145/351240.351262.
- [Cré07] P. Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, September 2007. doi:10.1007/s10990-007-9015-z.
- [Cur91] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, May 1991. doi:10.1016/0304-3975(91)90230-Y.
- [Dan05] O. Danvy. From reduction-based to reduction-free normalization. *Electronic Notes on Theoretical Computer Science*, 124(2):79–100, 2005. doi:10.1016/j.entcs.2005.01.007.
- [DH92] O. Danvy and J. Hatcliff. Thunks (continued). In *Proceedings of the Workshop on Static Analysis*, pages 3–11, 1992.
- [DHS00] S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000. doi:10.1016/S0167-739X(99)00088-6.
- [Dij68] E. W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11:147–148, 1968. doi:10.1145/362929.362947.
- [DJ04] N. A. Danielsson and P. Jansson. Chasing bottoms: a case study in program verification in the presence of partial and infinite values. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 85–109. Springer, 2004. doi:10.1007/978-3-540-27764-4_6.
- [DJ13] O. Danvy and J. Johannsen. From outermost reduction semantics to abstract machine. In *Proceedings of the 23rd International Symposium in Logic-Based Program Synthesis and Transformation*, volume 8901 of *LNCS*, pages 91–108. Springer, 2013. doi:10.1007/978-3-319-14125-1_6.
- [DJZ11] O. Danvy, J. Johannsen, and I. Zerny. A walk in the semantic park. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 1–12, 2011. doi:10.1145/1929501.1929503.
- [DM08] O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008. doi:10.1016/j.ipl.2007.10.010.
- [DM09] O. Danvy and K. Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. doi:10.1016/j.scico.2007.10.007.
- [DMM13] O. Danvy, K. Milikin, and J. Munk. A correspondence between full normalization by reduction and full normalization by evaluation, 2013. A scientific meeting in honor of Pierre-Louis Curien.
- [DR04] V. Danos and L. Regnier. Head linear reduction. Unpublished, 2004.
- [EHR91] L. Egidi, F. Honsell, and S. Ronchi della Rocca. The lazy call-by-value lambda-calculus. In *Proceedings of Mathematical Foundations of Computer Science*, pages 161–169, 1991. doi:10.1007/3-540-54345-7_59.

- [EHR92] L. Egidi, F. Honsell, and S. Ronchi Della Rocca. Operational, denotational and logical descriptions: a case study. *Fundamenta Informaticae*, 16(1):149–169, 1992. doi:10.3233/FI-1992-16205.
- [Eis20] R. A. Eisenberg. System FC, as implemented in GHC. Technical report, MS-CIS-15-09, University of Pennsylvania, 2020.
- [Fel87] M. Felleisen. *The Calculi of Lambda- ν -CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, 1987.
- [FF86] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *Formal Description of Programming Concepts III*, pages 193–217, 1986.
- [FFF09] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.
- [FH88] A. J. Field and P. G. Harrison. *Functional Programming*. International Series in Computer Science. Addison-Wesley, 1988. Reprinted 1993.
- [FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. doi:10.1016/0304-3975(92)90014-7.
- [GAL92] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26. ACM Press, 1992. doi:10.1145/143165.143172.
- [Gar14] Á. García-Pérez. *Operational Aspects of Full Reduction in Lambda Calculi*. PhD thesis, ETSI Informáticos, Universidad Politécnica de Madrid, 2014. doi:10.20868/UPM.thesis.32851.
- [GH11] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 2–14. ACM Press, 2011. doi:10.1145/2034773.2034777.
- [GL02] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of the 7th International ACM International Conference on Functional Programming*, volume 37(9), pages 235–246. ACM Press, 2002. doi:10.1145/581478.581501.
- [GPN13] Á. García-Pérez and P. Nogueira. A syntactic and functional correspondence between reduction semantics and reduction-free full normalisers. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 107–116. ACM Press, 2013. doi:10.1145/2426890.2426911.
- [GPN14] Á. García-Pérez and P. Nogueira. On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. *Science of Computer Programming*, 95 Part 2, 2014. doi:10.1016/j.scico.2014.05.011.
- [GPN16] Á. García-Pérez and P. Nogueira. No solvable lambda-value term left behind. *Logical Methods in Computer Science*, 12(2):1–43, 2016. doi:10.2168/LMCS-12(2:12)2016.
- [GPN19] Á. García-Pérez and P. Nogueira. The full-reducing Krivine abstract machine KN simulates pure normal-order reduction in lockstep: A proof via corresponding calculus. *Journal of Functional Programming*, 29:e7, 2019. doi:10.1017/S0956796819000017.
- [GPNG10] Á. García-Pérez, P. Nogueira, and E. J. Gallego Arias. The beta cube (extended abstract). In *Proceedings of the 1st International Workshop on Strategies in Rewriting, Proving, and Programming*, pages 3–7, 2010.
- [GPNM13] Á. García-Pérez, P. Nogueira, and J. J. Moreno-Navarro. Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order. In *Proceedings of the 15th International Symposium on Principles and Practice of Declarative Programming*, pages 85–96. ACM Press, 2013. doi:10.1145/2505879.2505887.
- [GPNS14] Á. García-Pérez, P. Nogueira, and I. Sergey. Deriving interpretations of the gradually-typed lambda calculus. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation*. ACM Press, 2014. doi:10.1145/2543728.2543742.
- [HD94] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 458–471, 1994. doi:10.1145/174675.178053.
- [Hen87] M. C. Henson. *Elements of Functional Languages*. Blackwell Scientific, 1987.

- [HHPW96] C. V. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996. doi:10.1145/227699.227700.
- [HHPW07] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, pages 12–1–12–55. ACM Press, 2007. doi:10.1145/1238844.1238856.
- [HS08] J. R. Hindley and J. P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, 2008.
- [Ing61] P. Z. Ingerman. Thanks: a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961. doi:10.1145/366062.366084.
- [JG95] C. B. Jay and N. Ghani. The virtues of eta-expansion. *Journal of Functional Programming*, 5(2):135–154, 1995. doi:10.1017/S0956796800001301.
- [JV04] P. Johann and J. Voigtländer. Free theorems in the presence of *seq*. In *Proceedings of 31st Symposium on Principles of Programming Languages*, pages 99–110, 2004. doi:10.1145/964001.964010.
- [Kah87] G. Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer, 1987. doi:10.1007/BFb0039592.
- [KMR21] A. Kerinec, G. Manzonetto, and S. Ronchi Della Rocca. Call-by-value, again! In *Proceedings of the 6th International Conference on Formal Structures for Computation and Deduction*, 2021. doi:10.4230/LIPIcs.FSCD.2021.7.
- [Kri07] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007. doi:10.1007/s10990-007-9018-9.
- [Lam89] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–30. ACM Press, 1989. doi:10.1145/96709.96711.
- [Lan64] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964. doi:10.1093/comjnl/6.4.308.
- [Lan65] P. Landin. A correspondence between ALGOL 60 and Church’s lambda-notation. *Communications of the ACM*, 8(2–3):89–101, 158–165, 1965. doi:10.1145/363791.363804.
- [Ler91] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1991.
- [Lév78] J.-J. Lévy. Réductions correctes et optimales dans le λ -calcul. Thèse d’Etat, Université de Paris VII, 1978.
- [Lév80] J.-J. Lévy. Optimal reductions in the lambda-calculus. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.
- [Mac90] B. J. MacLennan. *Functional Programming, Practice and Theory*. Addison-Wesley, 1990.
- [McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3:184–195, 1960. doi:10.1145/367177.367199.
- [McG70] C. L. McGowan. The correctness of a modified SECD machine. In *Proceedings of the 2nd ACM Symposium on Theory of Computing*, pages 149–157, 1970. doi:10.1145/800161.805160.
- [Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.
- [MOTW95] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Electronic Notes on Theoretical Computer Science*, 1:370–392, 1995. doi:10.1016/S1571-0661(04)00022-2.
- [Mun08] J. Munk. A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state. Master’s thesis, BRICS, Aarhus University, Denmark, April 2008. doi:10.7146/brics.v15i3.21938.
- [OS07] A. Othori and I. Sasano. Lightweight fusion by fixed point promotion. In *Proceedings of 34th Symposium on Principles of Programming Languages*, pages 143–154, 2007. doi:10.1145/1190216.1190241.
- [Pau96] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

- [Pey92] S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992. doi:10.1017/S0956796800000319.
- [Pie02] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [Plo75] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975. doi:10.1016/0304-3975(75)90017-1.
- [Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.
- [PR99] L. Paolini and S. Ronchi Della Rocca. Call-by-value solvability. *Theoretical Informatics and Applications*, 33(6):507–534, 1999. doi:10.1051/ita:1999130.
- [Rea89] C. Reade. *Elements of Functional Programming*. International Series in Computer Science. Addison-Wesley, 1989.
- [Rey72] J. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM Press, 1972. Reprinted in *Higher-Order and Symbolic Computation* 11(4), 1998. doi:10.1023/A:1010027404223.
- [Ros50] P. C. Rosenbloom. *The Elements of Mathematical Logic*. Dover Publications, 1950.
- [RP04] S. Ronchi Della Rocca and L. Paolini. *The Parametric Lambda Calculus*. Springer, 2004. doi:10.1007/978-3-662-10394-4.
- [Ses02] P. Sestoft. Demonstrating lambda calculus reduction. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 420–435. Springer, 2002. doi:10.1007/3-540-36377-7_19.
- [SPB23] F. Sieczkowski, M. Pyzik, and D. Biernacki. A general fine-grained reduction theory for effect handlers. *Proceedings of the ACM on Programming Languages*, 7(ICFP):511–540, 2023. doi:10.1145/3607848.
- [SR15] G. Scherer and D. Rémy. Full reduction in the face of absurdity. In *Proceedings of Programming Languages and Systems, 24th European Symposium on Programming*, pages 685–709. Springer, 2015. doi:10.1007/978-3-662-46669-8_28.
- [SS78] G. L. Steele, Jr. and G. J. Sussman. The art of the interpreter, or the modularity complex (parts zero, one, and two). AI Memo 453, MIT AI Laboratory, 1978.
- [Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [TeR03] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
- [Wad76] C. Wadsworth. The relation between computational and denotational properties for Scott’s D_∞ -models of the lambda calculus. *SIAM Journal of Computing*, 5(3):488–521, 1976. doi:10.1137/0205036.
- [Wad92a] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. doi:10.1017/S0960129500001560.
- [Wad92b] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, 1992. doi:10.1145/143165.143169.
- [Wad03] P. Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 189–201. ACM Press, 2003. doi:10.1145/944705.944723.