A Syntactic and Functional Correspondence between Reduction Semantics and Reduction-Free *Full* Normalisers *

Álvaro García-Pérez
IMDEA Software Institute, and
Universidad Politécnica de Madrid
agarcia@babel.ls.fi.upm.es

Pablo Nogueira
Universidad Politécnica de Madrid
pablo.nogueira@upm.es

Abstract

Olivier Danvy and others have shown the syntactic correspondence between reduction semantics (a small-step semantics) and abstract machines, as well as the functional correspondence between reduction-free normalisers (a big-step semantics) and abstract machines. The correspondences are established by program transformation (so-called interderivation) techniques. A reduction semantics and a reduction-free normaliser are interderivable when the abstract machine obtained from them is the same. However, the correspondences fail when the underlying reduction strategy is hybrid, i.e., relies on another sub-strategy. Hybridisation is an essential structural property of full-reducing and complete strategies. Hybridisation is unproblematic in the functional correspondence. But in the syntactic correspondence the refocusing and inlining-of-iterate-function steps become context sensitive, preventing the refunctionalisation of the abstract machine. We show how to solve the problem and showcase the interderivation of normalisers for normal order, the standard, full-reducing and complete strategy of the pure lambda calculus. Our solution makes it possible to interderive, rather than contrive, full-reducing abstract machines. As expected, the machine we obtain is a variant of Pierre Crégut's full Krivine machine KN.

* This research has been partially funded by the Spanish 'Ministerio de Economía y Competitividad' through project DESAFIOS10 TIN2009-14599, and by Comunidad de Madrid through programme PROMETIDOS P2009/TIC-1465. The first author is supported by Comunidad de Madrid grant CPI/0622/2008 and by IMDEA Software Institute.

1. Introduction

The interderivation techniques [1, 2, 5, 9–12], hereafter cited collectively as [IT], prove the correspondence between 'semantic artefacts' [9] that formally define the operational semantics of higher-order programming languages. Traditionally, a *small-step* operational semantics is given by a single-step reduction relation on terms which is a partial function. Reduction is defined as the reflexive and transitive closure. Figure 1 (right column) shows an example explained at length in Section 3. A syntactic theory or *reduction semantics* [11, 13] is a small-step operational semantics with an explicit representation of the reduction context. Figure 5 (Section 3) shows an example. Reduction is performed by iterating three steps: decomposing a term into a reduction context and a reducible expression ('redex' for short, plural 'redices'), contracting that redex, and plugging the contractum (the result) back into the context. The iteration either terminates (when the term is irreducible) or diverges (loops forever). A reduction semantics must satisfy a unique-decomposition property. A *reduction-based normaliser* is a program implementing a reduction semantics.

A *big-step* operational semantics is given by a partial function that does away with intermediate single steps and delivers, for an input term, the final irreducible term of the reduction sequence, if such term exists, or diverges otherwise. Figure 1 (left column) shows an example. A *reduction-free normaliser* is a program implementing the big-step partial function.[1]

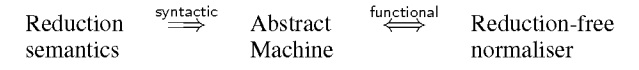
Small-step and big-step operational semantics are based on an underlying *reduction strategy*, informally, a total order in which redices are to be contracted.

Finally, an *abstract machine* is a state transition machine which, unlike a virtual machine, operates directly on terms and therefore has no instruction set and no need for a compiler.

One of the key contributions of [IT] is to show how to obtain an abstract machine from a reduction semantics (the *syntactic correspondence*) and from a reduction-free normaliser (the *functional correspondence*) by program-transformation steps. The functional correspondence consists of CPS-transformation and defunctionalisation steps, which are reversible. The syntactic correspondence consists of refocusing (which optimises the iteration loop), lightweight fusion [10] and inlining-of-iterate-function

[1] We also prefer 'normaliser' to 'reducer' or 'evaluator' because the first suggests that final results are irreducible terms, and 'evaluator' appears in the context of denotational semantics.

steps, which are not reversible in general. The following diagram illustrates the connections:



A reduction semantics and a reduction-free normaliser are said to be *interderivable* if they derive the same abstract machine. From left to right, a reduction-free normaliser can be derived from a reduction semantics. From right to left, a reduction semantics can be postulated whose derived abstract machine is also derived from the reduction-free normaliser. Interderived semantic artefacts are equivalent because the transformation steps are equivalence-preserving. By equivalent we mean they implement the same reduction strategy (contract redices in the same order). Informally, 'they all truly define the same elephant' [12].

In [IT] we find the equivalence of various call-by-name and call-by-value normalisers and their corresponding abstract machines, namely, Krivine's, SECD, CEK, CLS, etc.

However, the interderivation of *full-reducing* strategies, and corresponding abstract machines such as [6, 7, 15], has received scarce attention. The importance of full reduction has been acknowledged long ago [6]. Two applications are program optimisation by partial evaluation and type checking in proof assistants [15]. Full-reducing[2] strategies deliver (full-)normal-forms, as opposed to weak-normal-forms or weak-head-normal-forms. Paradigmatic full-reducing strategies of the *pure* lambda calculus are applicative order and normal order. Applicative order,[3] like call-by-name and call-by-value strategies, is a *uniform* strategy, it does not rely on another strategy and is defined only in terms of itself (see for example [19]). In contrast, normal order is a *hybrid* strategy, it does rely on a *subsidiary* uniform sub-strategy, namely, call-by-name. (We borrow 'uniform' and 'hybrid' terminology from [19].)

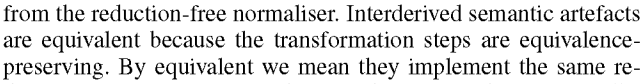
Hybridisation is unproblematic in the functional correspondence. In [2] we find the derivation of a big-step virtual machine from a reduction-free full normaliser for normal order. *However, the syntactic correspondence between reduction semantics and abstract machines cannot be established for hybrid strategies such as normal order using the current interderivation techniques verbatim.*

Hybridisation is important because it is a necessary condition for a strategy to be full-reducing and *complete*. A full-reducing strategy is complete if it delivers the full-normal-form of a term if it exists or diverges otherwise. Normal order is complete, it relies on call-by-name to avoid going prematurely 'under lambda' and discards unneeded potentially divergent subterms [20]. Another important complete hybrid strategy is the counterpart of normal order in the lambda-value calculus [17] (Section 7). In contrast, applicative order is uniform and is not complete: it reduces potentially divergent subterms.

Contributions: We refine the refocusing and inlining-of-iterate-function steps of the syntactic correspondence in order to accommodate hybrid strategies, and showcase the detailed interderivation of small-step and big-step semantic artefacts for normal order. The abstract machine we obtain is, as expected, a variant of the full Krivine machine KN [7], actually, KN is derivable from our machine. Our solution makes it possible to interderive, rather than contrive, full-reducing abstract machines. We use the same programming language (Standard ML) and follow the same steps in the presentation as [9] and [12]. The latter, which appeared in PEPM'11, is an excellent tutorial introduction to [IT]. We assume the reader is familiar with Standard ML, with those papers, and with the lambda calculus. Our code can be downloaded from the following URL:

`http://babel.ls.fi.upm.es/~agarcia/papers/PEPM13`

The following paragraphs summarise the technicalities. Like [12], we start from a search function and derive from it the context-based reduction semantics:



A search function locates the next redex to be contracted or returns the corresponding irreducible form if no contractable redices exist. Starting from a search function is important to discover the correspondence between reduction contexts and defunctionalised continuations. Two semantic artefacts (hybrid and subsidiary) have to be interderived in parallel but a single datatype for defunctionalised continuations is needed. If we apply the techniques literally then the refocus function in the reduction semantics and the iterate function in the 'pre-abstract machine' [11] end up depending on the current defunctionalised continuation to determine which normalisation function (hybrid or subsidiary) must continue. Consequently, after inlining the iterate function, the dispatcher for the abstract machine has to inspect the context stack deeply, i.e., has to look at the arguments of value constructors representing defunctionalised continuations. But since refunctionalisation requires a shallow inspection of the context stack, the machine obtained cannot be refunctionalised into a CPS program.

We observe that there is a dependency between the value constructors of defunctionalised continuations and which normalising function has to continue. This will allow us to rewrite the dispatcher to have the shallow inspection property. From the abstract machine we finally obtain the correct reduction-free normaliser.

2. Preliminaries

The pure untyped lambda calculus ($\lambda\mathbf{K}\beta$ calculus) is described in [4]. We use the traditional syntax of lambda terms as specified by the pseudo-grammar $\Lambda ::= x \mid (\lambda x.\Lambda) \mid (\Lambda\,\Lambda)$, where x, y, etc, range over the elements of a countably infinite set of variables. In words (to refresh terminology), terms consist of *variables*, of *abstractions* (consisting of a bound variable and a term called the abstraction *body*), and of *applications* of an *operator* term to an *operand* term. For example, $((\lambda x.x)y)$ is the identity abstraction applied to variable y. The abstraction $(\lambda x.x)$ is the operator in the application, and the variable y is the operand. We overload Λ for a grammatical non-terminal and for the set of lambda terms. Uppercase, sometimes primed, letters M, N, B, M', etc, range over elements of Λ. We use the standard precedence and association conventions: applications associate to the left and abstraction binds tighter than application. Hence, we write $(\lambda x.x)y$.

We use grammars in Extended Backus-Naur Form to define subsets of Λ. Alphanumeric non-terminal names are written in uppercase sans-serif. For example, $\mathsf{NF} ::= \lambda x.\mathsf{NF} \mid x\,\{\mathsf{NF}\}^*$ defines the set of full-normal-forms. The regular expression $\{\alpha\}^*$ stands for zero or more occurrences of the sentential form α. For example, x, $x\,\mathsf{NF}$, $x\,\mathsf{NF}\,\mathsf{NF}$, etc., are sentential forms of the second production, which respectively associate as x, $(x\,\mathsf{NF})$, $((x\,\mathsf{NF})\,\mathsf{NF})$, etc.

The reader must be familiar with the usual notions of bound and free variable, syntactic equality of terms modulo renaming of bound variables (written $\equiv$), substitution ($[N/x]B$ stands for the term resulting from the capture-avoiding substitution of N for the free occurrences of x in B), the notion of redex (a term of the

[2] Some authors use 'strong-reducing', but that may be confused with 'strong-normalising' which means something different [4].

[3] Applicative order should not be confused with call-by-value! It is a strategy of the pure lambda calculus, not of the pure lambda-value calculus [17]. It reduces operands before applications and realises the idea of 'passing parameters by value' but 'value' here means 'full-normal-form' and not 'non-applications' as in the lambda-value calculus.

$$\frac{}{x \Downarrow_{bn} x}\ (\textsc{Bn-Var}) \qquad \frac{}{\lambda x.B \Downarrow_{bn} \lambda x.B}\ (\textsc{Bn-Abs})$$

$$\frac{M \Downarrow_{bn} M' \quad M' \equiv \lambda x.B \quad [N/x]B \Downarrow_{bn} B'}{M\,N \Downarrow_{bn} B'}\ (\textsc{Bn-Con})$$

$$\frac{M \Downarrow_{bn} M' \quad M' \not\equiv \lambda x.B}{M\,N \Downarrow_{bn} M'\,N}\ (\textsc{Bn-Neu})$$

$$\frac{}{x \Downarrow_{no} x}\ (\textsc{No-Var}) \qquad \frac{B \Downarrow_{no} B'}{\lambda x.B \Downarrow_{no} \lambda x.B'}\ (\textsc{No-Abs})$$

$$\frac{M \Downarrow_{bn} M' \quad M' \equiv \lambda x.B \quad [N/x]B \Downarrow_{no} B'}{M\,N \Downarrow_{no} B'}\ (\textsc{No-Con})$$

$$\frac{M \Downarrow_{bn} M' \quad M' \not\equiv \lambda x.B \quad M' \Downarrow_{no} M'' \quad N \Downarrow_{no} N'}{M\,N \Downarrow_{no} M''\,N'}\ (\textsc{No-Neu})$$

$$\frac{}{(\lambda x.B)N \rightarrow_{no} [N/x]B}\ (\beta) \qquad \frac{M \notin \mathsf{WHNF} \quad M \rightarrow_{no} M'}{M\,N \rightarrow_{no} M'\,N}\ (\mu 1)$$

$$\frac{M \in \mathsf{WHNF} \quad M \not\equiv \lambda x.B \quad M \rightarrow_{no} M'}{M\,N \rightarrow_{no} M'\,N}\ (\mu 2)$$

$$\frac{M \in \mathsf{NF} \quad M \not\equiv \lambda x.B \quad N \rightarrow_{no} N'}{M\,N \rightarrow_{no} M\,N'}\ (\nu) \qquad \frac{B \rightarrow_{no} B'}{\lambda x.B \rightarrow_{no} \lambda x.B'}\ (\xi)$$

$$\begin{array}{rcl} \mathsf{WHNF} & ::= & \lambda x.\Lambda \mid x\,\{\Lambda\}^* \\ \mathsf{NF} & ::= & \lambda x.\mathsf{NF} \mid x\,\{\mathsf{NF}\}^* \end{array}$$

Figure 1. Big-step (left column) and small-step (right column) operational semantics of normal order.

form $(\lambda x.B)N$) and the single-step reduction relation $\rightarrow_\beta$ and its reflexive and transitive closure $\rightarrow_\beta^*$.

An operational semantics is a partial function that is a subrelation of $\rightarrow_\beta^*$. We are interested in operational semantics which realise a reduction strategy by choosing redices in a fixed order. We write $M \rightarrow_s N$ and $M \Downarrow_s N$ respectively for the small-step and big-step semantics of reduction strategy s. We use relational $M \Downarrow_s N$ and functional $\Downarrow_s(M) = N$ notation interchangeably in order to use diagrammatic composition when appropriate, e.g., $(\Downarrow_s; \Downarrow_t)(M) = \Downarrow_t(\Downarrow_s(M))$.

3. Normal order and hybridisation

Normal order is typically defined by the slogan 'contract the leftmost redex first', understanding 'leftmost' as in [8] or 'leftmost-outermost' when referring to the redex's position in the abstract syntax tree of the term. Normal order is complete and is substantiated by the Standardisation Theorem [8]. Figure 1 shows the big-step (left column) and small-step (right column) semantics of normal order.

Let us begin with the big-step (relation $\Downarrow_{no}$). Rule No-Var says normal order on variables is an identity. Rule No-Abs says that normal order recursively 'goes under lambda'. Rules No-Con and No-Neu say in their first premiss that normal order relies on call-by-name $\Downarrow_{bn}$ to reduce operators in applications MN (potential redices). If the result is an abstraction (second premiss of Rule No-Con) then normal order reduces the result of substituting the *unreduced* operand for the free variable in the abstraction body, thus implementing non-strict functional semantics for redices. If the result is not an abstraction (second premiss of Rule No-Neu) then the original application MN is a *neutral term*. (Neutral terms are either single variables or non-redex applications.) In that case normal order fully-reduces the operator and the operand.

Let us look at subsidiary call-by-name $\Downarrow_{bn}$. Rule Bn-Abs says that, unlike normal order, call-by-name does not go under lambda. Rule Bn-Neu says that, unlike normal order, it does not reduce operands in neutral terms. Rule Bn-Con says that, like normal order, it implements non-strict functional semantics for redices. Call-by-name is a uniform strategy recursively defined only in terms of itself. Normal order relies on call-by-name so that the leftmost redex $(\lambda x.B)N$ is contracted next and not a redex within B by going prematurely under lambda.

```
datatype term = IND of int | LAM of term
              | APP of term * term

(*  bn : term -> term *)
fun bn (i as IND n) = i
  | bn (l as LAM b) = l
  | bn (APP (m, n)) =
    let val m' = bn m
    in (case m' of (LAM b) => bn (subst (b, n, 0))
                 | _       => APP (m', n))
    end

(*  no : term -> term *)
fun no (i as IND n) = i
  | no (LAM b)      = LAM (no b)
  | no (APP (m, n)) =
    let val m' = bn m
    in (case m' of (LAM b) => no (subst (b, n, 0))
                 | _       => APP (no m', no n))
    end
```

Figure 2. Canonical substitution-based reduction-free normaliser for normal order.

The reader should not be daunted by the rules. These conform to the format of Hilbert-style logical theories for defining relations [4]. In fact, $\Downarrow_{no}$ and $\Downarrow_{bn}$ are syntax-directed partial functions with a natural semantics interpretation [16]. The Boolean conditions in premisses are non-overlapping so the rules can be applied deterministically and translate directly to a strict functional program in which a term matching the left-hand-side of the conclusion is recursively reduced according to the premisses from left to right, with conditions corresponding to case analysis. The canonical 'substitution-based' (following terminology in [5]) reduction-free normaliser in Figure 2 has been written directly from the rules, save for the de-Bruijn-indices representation [4] of lambda terms. Function `subst` implements capture-avoiding substitution.

Now to the small-step (relation $\rightarrow_{no}$). There is no rule for variables because these are in full-normal-form. Rule (β) reduces redices. Rule (ξ) provides structural compatibility with abstractions (going under lambda). Apart from (β), there are three other rules for applications. Rule $(\mu 1)$ reduces the operator when it is not in weak-

$$\frac{}{x \Downarrow_{rn} x}\ (\text{RN-VAR}) \qquad \frac{B \Downarrow_{bn} B' \quad B' \Downarrow_{rn} B''}{\lambda x.B \Downarrow_{rn} \lambda x.B''}\ (\text{RN-ABS})$$

$$\frac{M \Downarrow_{rn} M' \quad N \Downarrow_{bn} N' \quad N' \Downarrow_{rn} N''}{M\,N \Downarrow_{rn} M'\,N''}\ (\text{RN-NEU})$$

Figure 3. The readback stage of normal order.

$$\frac{B \to^{p}_{\Lambda} B'}{\lambda x.B \to^{p}_{\Lambda} \lambda x.B'}\ (p1)$$

$$\frac{i = \min\{j \leq m | M_j \notin \mathsf{NF}\} \quad M_i \to^{p}_{\Lambda} M_i'}{x\,M_1 \ldots M_m \to^{p}_{\Lambda} x\,M_1 \ldots M_i' \ldots M_m}\ (p2)$$

$$\frac{}{(\lambda x.B)N\,M_1 \ldots M_m \to^{p}_{\Lambda} [N/x]B\,M_1 \ldots M_m}\ (p3)$$

Figure 4. Principal reduction machine for normal order.

head-normal-form (hereafter whnf). Rule $(\mu 2)$ reduces the operator when it is not an abstraction, if it were then Rule (β) would have been applicable instead. Finally, Rule (ν) reduces the operand when the operator is in full-normal-form (hereafter nf). Although a nf is also a whnf, Rules $(\mu 2)$ and (ν) are non-overlapping because the third premiss in $(\mu 2)$ is not the case when $M \in \mathsf{NF}$.

Hybridisation is a structural property. In the big-step semantics, a subsidiary set of rules defines a strategy used by normal order. In the small-step semantics, a subsidiary sub-relation can be identified. More concretely, $\Downarrow_{bn}$ is used on operators by $\Downarrow_{no}$, and (β) and $(\mu 1)$ define small-step call-by-name, exactly function $\to_n$ in Plotkin's λ_N calculus [17]. It is immediate to prove that call-by-name is a left identity of normal order: $\Downarrow_{bn}; \Downarrow_{no} = \Downarrow_{no}$.

Hybridisation is inexorably conspicuous in alternative renditions, for example:

- Normal order can be defined in eval-readback style in a similar fashion to the definition of $N()$, a strict full-reducing strategy for a closed calculus [15]. In our case, the 'eval' stage is $\Downarrow_{bn}$, and the 'readback' stage is $\Downarrow_{rn}$ shown in Figure 3. Normal order is the composition $\Downarrow_{bn}; \Downarrow_{rn}$. The eval stage carries out the reduction steps corresponding to the first premiss in Rule NO-CON (shared with Rule NO-NEU) and the nested call-by-name reduction steps of the third premiss. The distribution of reduction over whnfs is deferred to the readback stage. Notice that $\Downarrow_{rn}$ has no RN-CON rule because it operates on terms in whnf, and a term in whnf cannot have an outermost redex.
- Normal order is obtained by instantiating Δ with Λ, and Δ-nf with NF in the 'principal reduction machine' $\to^{p}_{\Delta}$ of the parametric lambda calculus [18], as shown in Figure 4. The parametric lambda calculus generalises various lambda calculi by adding the premiss $N \in \Delta$ to the beta rule. Different calculi are obtained by choosing particular sets of terms for Δ. The principal reduction machine is simply a small-step reduction strategy that is parametric on Δ and on a notion of normal form Δ-nf. It uses a flattened representation of multiple applications. The hybrid-subsidiary interplay can be observed in the rules of Figure 4. Rule $(p3)$ subsumes rules (β) and $(\mu 1)$ in Figure 1 which correspond to call-by-name. Rules $(p1)$ and $(p2)$ reduce terms in whnf. The left-hand-sides of their conclusions match the definition of WHNF (Figure 1).

Terms:	Λ	$::=$	$x \mid \lambda x.\Lambda \mid \Lambda\,\Lambda$
Normal forms:	NF	$::=$	$\lambda x.\mathsf{NF} \mid \mathsf{NNF}$
	NNF	$::=$	$x \mid \mathsf{NNF}\,\mathsf{NF}$
Contexts:	$\mathsf{C}_{no}[\,]$	$::=$	$\mathsf{C}_{bn}[\,] \mid \lambda x.\mathsf{C}_{no}[\,] \mid \mathsf{C}_{ne}[\,]$
	$\mathsf{C}_{bn}[\,]$	$::=$	$[\,] \mid \mathsf{C}_{bn}[\,]\,\Lambda$
	$\mathsf{C}_{ne}[\,]$	$::=$	$\mathsf{NNF}\,\mathsf{C}_{no}[\,] \mid \mathsf{C}_{ne}[\,]\,\Lambda$
β-rule:	$(\lambda x.B)N \to_{\beta} [N/x]B$		

Figure 5. Reduction semantics for normal order.

Context-based reduction semantics. A context-based reduction semantics [11, 13] is the starting point of a syntactic correspondence. Hybridisation is also conspicuous in the context-based reduction semantics of normal order, shown in Figure 5, which consists of terms, nfs, reduction contexts, and the β-rule. Normal order is the iteration of single reductions consisting of (i) decomposing a term into a reduction context (derived from non-terminal $\mathsf{C}_{no}[\,]$) and a redex within the hole $[\,]$, (ii) contracting the redex and, (iii) plugging the contractum back into its context, recomposing the next reduct in the reduction sequence [11]. The iteration either terminates when the term is a nf or otherwise diverges. Non-terminal NNF defines neutral terms in full-normal-form. Figure 6 shows two single reductions for a particular term. The reduction contexts obtained are shown on the right. The context $\mathsf{C}_{no}[\,]$ satisfies the unique-decomposition property, that is, a term is either a nf or can be decomposed into a unique context and the next redex. The proof that $\mathsf{C}_{no}[\,]$ is uniquely-decomposable proceeds by simple induction on terms.

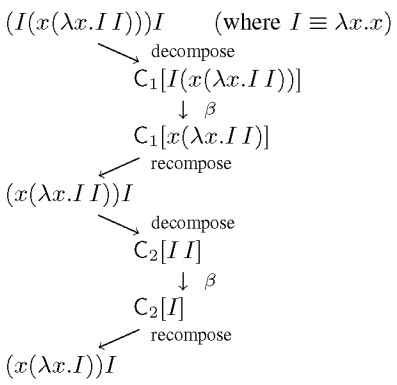
The hybrid-subsidiary interplay can be observed in that $\mathsf{C}_{no}[\,]$ includes the call-by-name reduction context $\mathsf{C}_{bn}[\,]$. The reduction semantics for call-by-name consists of $\mathsf{C}_{bn}[\,]$ and the β-rule. The inclusion is unavoidable: $\mathsf{C}_{no}[\,]$ cannot be defined without $\mathsf{C}_{bn}[\,]$ which has to be used in the operator position when an application is a redex. Notice also that $\mathsf{C}_{no}[\,]$ contexts other than $\mathsf{C}_{bn}[\,]$ precisely match the shape of a whnf. A hole can only occur at the body of an abstraction, $\lambda x.\mathsf{C}_{no}[\,]$, or at the operand of an application with a neutral term in full-normal-form as operator $\mathsf{NNF}\,\mathsf{C}_{no}[\,]$. This application may be possibly applied to additional unreduced arguments, $\mathsf{C}_{ne}[\,]\,\Lambda$. In the latter case, the hole is always at the right of a nf and at the left of arbitrary terms, thus enforcing left reduction: $\mathsf{NNF}\,\mathsf{C}_{no}[\,]\,\Lambda \ldots \Lambda$.

4. Prelude to a reduction semantics

Like [12] we start the syntactic correspondence from a search function that locates the next redex to be contracted and then derive from it the context-based reduction semantics of Figure 5. The transformation of the search function into decomposition functions will shed light on the correspondence between reduction contexts and defunctionalised continuations (Section 4.5).

Two search functions are required, one for the hybrid that searches for a nf or the next redex to be contracted, and another for the subsidiary that searches for a whnf or the next redex in the call-by-name sub-reduction to be contracted. The two search functions are to be transformed into decomposition functions which, additionally to the next redex, give the context where it appears. We construct the search functions from the small-step definition of normal order in Figure 1, observing that rules (μ_2), (ν) and (ξ) are applicable only when a term is already in whnf, and therefore, they are not applied by the subsidiary search function:

- The hybrid search must invoke the subsidiary search over operators in applications in order to check whether they are whnfs or not, correlating with Rules $(\mu 1)$ and $(\mu 2)$. When a non-abstraction whnf is found, the hybrid search must be invoked



Context	Derivation
$\mathsf{C}_1[\,] \equiv [\,]\,I$	$\mathsf{C}_{no}[\,] \Rightarrow$ $\mathsf{C}_{bn}[\,] \Rightarrow$ $\mathsf{C}_{bn}[\,]\,\Lambda \Rightarrow$ $[\,]\,\Lambda \Rightarrow [\,]\,I$
$\mathsf{C}_2[\,] \equiv (x(\lambda x.[\,]))I$	$\mathsf{C}_{no}[\,] \Rightarrow \mathsf{C}_{ne}[\,] \Rightarrow$ $\mathsf{C}_{ne}[\,]\,\Lambda \Rightarrow$ $(\mathsf{NNF}\,\mathsf{C}_{no}[\,])\Lambda \Rightarrow$ $(x\,\mathsf{C}_{no}[\,])\Lambda \Rightarrow$ $(x(\lambda x.\mathsf{C}_{no}[\,]))\Lambda \Rightarrow$ $(x(\lambda x.\mathsf{C}_{bn}[\,]))\Lambda \Rightarrow$ $(x(\lambda x.[\,]))\Lambda \Rightarrow$ $(x(\lambda x.[\,]))I$

Figure 6. Example of normal order reduction sequence using context-based reduction semantics.

over that whnf, correlating with Rule ($\mu 2$). If the result of that search is a nf then the hybrid search must be invoked over the operand, correlating with Rule (ν). Finally, the hybrid search must always be invoked over the bodies of abstractions, correlating with Rule (ξ).

- The subsidiary search, in turn, must invoke itself recursively on operators, correlating with Rule ($\mu 1$). It could be realised as a predicate function that tests if the input term is in whnf, after which hybrid search would continue if it is not. However, hybrid search subsumes such predicate function and returns the next contratable redex when the term is not in whnf. Our subsidiary search realises that subrelation of the hybrid search.

Notice that free variables are trivially returned as nfs or whnfs, respectively, to the hybrid or subsidiary search functions.

4.1 One datatype embedding nfs in whnfs

The datatype `term` has been given in Figure 2. As for nfs and whnfs, we could define a different datatype for each but this complicates the CPS transformation step (there will be different defunctionalised continuation datatypes for subsidiary and hybrid) which would require some sort of two-layer CPS. Rather, we embed nfs into whnfs, reuse most of the datatypes (irreducible forms, redices, defunctionalised continuations, etc) and use a flat CPS. The type `whnf` follows the *function* and *accumulator* representation of [15] and corresponds to non-terminal WHNF in Figure 1 (right column). A whnf is either an abstraction (in de-Bruijn-indices representation) or a neutral term consisting of a variable (an index) applied to zero or more terms. Function `embed` recovers a `term` from a `whnf`, and `apply_acc` appends a `term` operand to the accumulator representation of whnfs.

```
datatype whnf = FUN of term | ACC of int * term list

(*  embed : whnf -> term *)
fun embed (FUN b)       = LAM b
  | embed (ACC (n, ts)) = embed_aux (IND n, ts)

(*  embed_aux : term * term list -> term *)
and embed_aux (t, [])        = t
  | embed_aux (m, (t :: ts)) =
    embed_aux (APP (m, t), ts)

(*  apply_acc : whnf * term -> whnf *)
fun apply_acc (ACC (n, ts), t)
    = ACC (n, ts @ [t])
```

4.2 Search functions

The datatype `redex` represents redices. The datatype `found` consists of a `whnf` (recall it embeds nfs!) or a `redex`. Function `search_whnf` implements the subsidiary search, and `search_nf` the hybrid search.

```
datatype redex = SUB of term * term
datatype found = WHNF of whnf | RED of redex

(*  search_whnf : term -> found *)
fun search_whnf (IND n)      = WHNF (ACC (n, []))
  | search_whnf (LAM b)      = WHNF (FUN b)
  | search_whnf (APP (m, n)) =
    (case search_whnf m
      of (WHNF wm) =>
         (case wm  of (FUN b) => RED (SUB (b, n))
                    | _       => WHNF (apply_acc (wm, n)))
       | (red as RED _) => red)

(*  search_nf : term -> found *)
fun search_nf (IND n)      = WHNF (ACC (n, []))
  | search_nf (LAM b)      =
    (case search_nf b
      of (WHNF wb)      => WHNF (FUN (embed wb))
       | (red as RED _) => red)
  | search_nf (APP (m, n)) =
    (case search_whnf m
      of (WHNF wm)      =>
         (case wm of (FUN b) => RED (SUB (b, n))
                   | _       =>
            (case search_nf (embed wm)
               of (WHNF nm)      =>
                  (case search_nf n
                    of (WHNF nn)      =>
                       WHNF (apply_acc (nm, embed nn))
                     | (red as RED _) => red)
                | (red as RED _) => red))
       | (red as RED _) => red)

(*  search : term -> found *)
fun search t = search_nf t
```

4.3 The search functions in continuation-passing style

```
(*  search_whnf_cps : term * (found -> 'a) -> 'a *)
fun search_whnf_cps (IND n, k)      = k (WHNF (ACC (n, [])))
  | search_whnf_cps (LAM b, k)      = k (WHNF (FUN b))
  | search_whnf_cps (APP (m, n), k) =
    search_whnf_cps (m,
     fn (WHNF wm)      =>
        (case wm of (FUN b) => k (RED (SUB (b, n)))
```

```
                | _       => k (WHNF (apply_acc (wm, n))))
    | (red as RED _) => k red)

(*  search_nf_cps : term * (found -> 'a) -> 'a *)
fun search_nf_cps (IND n, k)       = k (WHNF (ACC (n, [])))
  | search_nf_cps (LAM b, k)       =
    search_nf_cps (b,
     fn (WHNF wb)      => k (WHNF (FUN (embed wb)))
      | (red as RED _) => k red)
  | search_nf_cps (APP (m, n), k) =
    search_whnf_cps (m,
     fn (WHNF wm)      =>
        (case wm
          of (FUN b) => k (RED (SUB (b, n)))
           | _       =>
             search_nf_cps (embed wm,
              fn (WHNF nm)      =>
                 search_nf_cps (n,
                  fn (WHNF nn)      =>
                     k (WHNF (apply_acc (nm, embed nn)))
                   | (red as RED _) => k red)
               | (red as RED _) => k red))
    | (red as RED _) => k red)

(*  search1 : term -> found *)
fun search1 t = search_nf_cps (t, fn f => f)
```

4.4 Simplifying the CPS-transformed search functions

The CPS-transformed search functions are simplified by returning a result only when a redex or a whnf is found. The simplification rests on an isomorphism between the type signatures for continuations [3]. Datatype `found` is the disjoint sum of `whnf` and `redex`, and type `(whnf + redex -> 'a)` is isomorphic to type `(whnf -> 'a) * (redex -> 'a)`. The `(redex -> 'a)` continuations are always the identity and can be optimised away. Because the end result is always of type `found`, we can instantiate `'a` to `found`, leaving `term * (whnf -> found) -> found` as the type signature for the CPS-transformed search functions.

```
(*  search_whnf_sim : term * (whnf -> found) -> found *)
fun search_whnf_sim (IND n, k)       = k (ACC (n, []))
  | search_whnf_sim (LAM b, k)       = k (FUN b)
  | search_whnf_sim (APP (m, n), k) =
    search_whnf_sim (m,
     fn wm =>
        (case wm of (FUN b) => RED (SUB (b, n))
                  | _       => k (apply_acc (wm, n))))

(*  search_nf_sim : term * (whnf -> found) -> found *)
fun search_nf_sim (IND n, k)       = k (ACC (n, []))
  | search_nf_sim (LAM b, k)       =
    search_nf_sim (b, fn wb => k (FUN (embed wb)))
  | search_nf_sim (APP (m, n), k)  =
    search_whnf_sim (m,
     fn wm =>
        (case wm
          of (FUN b) => RED (SUB (b, n))
           | _       => search_nf_sim (embed wm,
              fn nm =>
                 search_nf_sim (n,
                  fn nn =>
                     k (apply_acc (nm, embed nn))))))

(*  search2 : term -> found *)
fun search2 t = search_nf_sim (t, fn v => WHNF v)
```

4.5 Defunctionalising continuations

Continuations are defunctionalised by enumerating the inhabitants of the function space and introducing the `apply_cont` function that dispatches on them. The type `continuation` is an explicit representation of the context in which a search takes place.

```
datatype continuation = C0
                      | C1 of term * continuation
                      | C2 of continuation
                      | C3 of term * continuation
                      | C4 of term * continuation
                      | C5 of continuation * whnf

(*  apply_cont : continuation * whnf -> found *)
fun apply_cont (C0, w)          = WHNF w
  | apply_cont (C1 (n, k), wm)  =
    (case wm of (FUN b) => RED (SUB (b, n))
              | _       => apply_cont (k, apply_acc (wm, n)))
  | apply_cont (C2 k, wb)       =
    apply_cont (k, FUN (embed wb))
  | apply_cont (C3 (n, k), wm)  =
    (case wm
      of (FUN b) => RED (SUB (b, n))
       | _       => search_nf_cont (embed wm, C4 (n, k)))
  | apply_cont (C4 (n, k), nm)  =
    search_nf_cont (n, C5 (k, nm))
  | apply_cont (C5 (k, nm), nn) =
    apply_cont (k, apply_acc (nm, embed nn))

(*  search_whnf_cont : term * continuation -> found *)
and search_whnf_cont (IND n, k)      =
    apply_cont (k, ACC (n, []))
  | search_whnf_cont (LAM b, k)      =
    apply_cont (k, (FUN b))
  | search_whnf_cont (APP (m, n), k) =
    search_whnf_cont (m, C1 (n, k))

(*  search_nf_cont : term * continuation -> found *)
and search_nf_cont (IND n, k)      =
    apply_cont (k, ACC (n, []))
  | search_nf_cont (LAM b, k)      = search_nf_cont (b, C2 k)
  | search_nf_cont (APP (m, n), k) =
    search_whnf_cont (m, C3 (n, k))

(*  search3 : term -> found *)
fun search3 t = search_nf_cont (t, C0)
```

Constructor `C0` stands for the initial continuation. Constructor `C1` stands for the continuation in `search_whnf_sim`, while `C2`, `C3`, `C4` and `C5` stand for the continuations in `search_nf_sim`. Constructors `C1` and `C3` are analogous, they both correspond to a context with the hole inside the operator. However, we cannot subsume them into a single constructor because they inform about which search function is to continue: the dispatching function `apply_cont` acts differently on them when it encounters neutral terms. It applies the parameter continuation `k` to the whole term in the case of `C1` but applies `search_nf_cont` over the operator with a new `C4` continuation in the case of `C3`. Thus, the datatype `continuation` carries more information than just the position of the hole in the term. Observe that looking at the context stack shallowly is not enough to know whether we are within the hybrid or the subsidiary, because `C3` appears in both cases.

As expected, defunctionalised continuations correspond to reduction contexts (Figure 5) with some subtle appreciations:

- `C0` stands for the empty context: $[\,]$.
- `C1` stands for a call-by-name context in the operator, which is not derived from a normal order context: $\mathsf{C}_{bn}[\,]\,\Lambda$.
- `C2` stands for a context in the body of a lambda: $\lambda x.\mathsf{C}_{no}[\,]$.
- `C3` stands for a call-by-name context in the operator, but this time derived from a normal order context: $\mathsf{C}_{bn}[\,]\,\Lambda$.
- `C4` stands for a context in the operator of a neutral term: $\mathsf{C}_{ne}[\,]\,\Lambda$.
- `C5` stands for a context in the operand of a neutral term in nf: $\mathsf{NNF}\;\mathsf{C}_{no}[\,]$.

There is no need to distinguish between initial continuations (holes) which are derived from a normal order context or not, because there is nothing to do after the initial continuation. However, an implementation with two initial continuations (respectively for hybrid and subsidiary) would also be correct. Nevertheless, for the purpose of full reduction, the initial continuation which corresponds to call-by-name would be unused. Only a stand-alone normaliser for the subsidiary call-by-name would use it.

4.6 From search to decomposition

We turn the search into a decomposition. The result is the found redex (if any) together with the reduction context where it appears.

```
datatype whnf_or_decomposition = WHNF of whnf
                               | DEC of redex * continuation

(*  decompose_cont : continuation * whnf
                     -> whnf_or_decomposition *)
fun decompose_cont (C0, w)          = WHNF w
  | decompose_cont (C1 (n, k), wm)  =
    (case wm
      of (FUN b) => DEC (SUB (b, n), k)
       | _       => decompose_cont (k, apply_acc (wm, n)))
  | decompose_cont (C2 k, wb)       =
    decompose_cont (k, FUN (embed wb))
  | decompose_cont (C3 (n, k), wm)  =
    (case wm
      of   (FUN b) => DEC (SUB (b, n), k)
         | _       => decompose_nf (embed wm, C4 (n, k)))
  | decompose_cont (C4 (n, k), nm)  =
    decompose_nf (n, C5 (k, nm))
  | decompose_cont (C5 (k, nm), nn) =
    decompose_cont (k, apply_acc (nm, embed nn))

(*  decompose_whnf : term * continuation
                     -> whnf_or_decomposition *)
and decompose_whnf (IND n, k)      =
    decompose_cont (k, ACC (n, []))
  | decompose_whnf (LAM b, k)      =
    decompose_cont (k, FUN b)
  | decompose_whnf (APP (m, n), k) =
    decompose_whnf (m, C1 (n, k))

(*  decompose_nf : term * continuation
                   -> whnf_or_decomposition *)
and decompose_nf (IND n, k)      =
    decompose_cont (k, ACC (n, []))
  | decompose_nf (LAM b, k)      = decompose_nf (b, C2 k)
  | decompose_nf (APP (m, n), k) =
    decompose_whnf (m, C3 (n, k))

(*  decompose : term -> whnf_or_decomposition *)
fun decompose t = decompose_nf (t, C0)

(*  recompose : continuation * term -> term *)
fun recompose (C0, t)         = t
  | recompose (C1 (n, k), m)  = recompose (k, APP (m, n))
  | recompose (C2 k, b)       = recompose (k, LAM b)
  | recompose (C3 (n, k), m)  = recompose (k, APP (m, n))
  | recompose (C4 (n, k), m)  = recompose (k, APP (m, n))
  | recompose (C5 (k, nm), n) =
    recompose (k, APP (embed nm, n))
```

5. A reduction semantics

5.1 Reduction-based normalisation

The reduction-based approach iterates decomposition, contraction and recomposition until a nf is found:

```
datatype contractum = CONTRACTUM of term
                    | ERROR of string

(*  contract : redex -> contractum *)
fun contract (SUB (b, n)) = CONTRACTUM (subst (b, n, 0))

datatype result_or_wrong = RESULT of whnf
                         | WRONG of string

(*  refocus : term * continuation
              -> whnf_or_decomposition *)
fun refocus con = (decompose (recompose con))

(*  iterate : whnf_or_decomposition -> result_or_wrong *)
fun iterate (WHNF w)       = RESULT w
  | iterate (DEC (red, k)) =
    (case contract red
      of (CONTRACTUM t) => iterate (refocus (k, t))
       | (ERROR s)      => WRONG s)

(*  normalise : term -> result_or_wrong *)
fun normalise t = iterate (decompose t)
```

The normaliser implements a small-step state transition machine in *trampolined style* [14], where configurations (states) coincide with decompositions (`whnf_or_decomposition`). Discrete transitions steps are implemented by the composition of `contract`, `recompose` and `decompose`. The last two constitute extensional refocusing [11, 12] (function `refocus`). The `iterate` function is the trampoline, taking a decomposition, contracting the redex and then recursively invoking itself over the refocused contractum until the decomposition consists of a `whnf` (again, recall it embeds nfs).

5.2 Refocusing intensionally

We deforest recomposition and decomposition into an intensional refocus function [9]. The reduction-free iterate-and-refocus normaliser does away with intermediate reducts:

```
(*  refocus1 : term * continuation
               -> whnf_or_decomposition *)
fun refocus1 (t, k) =
    (case k of (C1 (_, _) | C3 (_, _)) =>
                 decompose_whnf (t, k)
             | _                       =>
                 decompose_nf (t, k))

(*  iterate1 : whnf_or_decomposition -> result_or_wrong *)
fun iterate1 (WHNF w)       = RESULT w
  | iterate1 (DEC (red, k)) =
    (case contract red of (CONTRACTUM t) =>
                            iterate1 (refocus1 (t, k))
                        | (ERROR s)      => WRONG s)

(*  normalise1 : term -> result_or_wrong *)
fun normalise1 t = iterate1 (refocus1 (t, C0))
```

In `refocus1`, we inspect the current continuation `k` to decide on which decomposition function to continue. The only continuations on which function `decompose_whnf` (Section 4.6) is invoked are `C1` and `C3`. The case expression in `refocus1` takes care of that.

5.3 Pre-abstract machine

The contraction function `contract` can be inlined, and consequently the `ERROR` case disappears because `contract` does not consider execution errors. A result (if any) can only be a nf, otherwise the iteration diverges.

```
(* iterate2 : whnf_or_decomposition -> result_or_wrong *)
fun iterate2 (WHNF w)              = RESULT w
  | iterate2 (DEC (SUB (b, n), k)) =
    iterate2 (refocus1 (subst (b, n, 0), k))

(*  normalise2 : term -> result_or_wrong *)
fun normalise2 t = iterate2 (refocus1 (t, C0))
```

This is a *pre-abstract machine* [11], where the transition function uses the intensional `refocus1` and the trampoline `iterate2` schedules the transition function until a result is obtained.

5.4 Lightweight fusion by fixed-point promotion

Functions `iterate2` and `refocus1` are lightweight-fused [10]:

```
(*  normalise3_cont : continuation * whnf
                      -> result_or_wrong *)
fun normalise3_cont (C0, w)          = iterate3 (WHNF w)
  | normalise3_cont (C1 (n, k), wm)  =
    (case wm of (FUN b) => iterate3 (DEC (SUB (b, n), k))
              | _       =>
                normalise3_cont (k, apply_acc (wm, n)))
  | normalise3_cont (C2 k, wb)       =
    normalise3_cont (k, FUN (embed wb))
  | normalise3_cont (C3 (n, k), wm)  =
    (case wm of (FUN b) => iterate3 (DEC (SUB (b, n), k))
              | _       =>
                normalise3_nf (embed wm, C4 (n, k)))
  | normalise3_cont (C4 (n, k), nm)  =
    normalise3_nf (n, C5 (k, nm))
  | normalise3_cont (C5 (k, nm), nn) =
    normalise3_cont (k, apply_acc (nm, embed nn))

(*  normalise3_whnf : term * continuation
                      -> result_or_wrong *)
and normalise3_whnf (IND n, k)       =
    normalise3_cont (k, ACC (n, []))
  | normalise3_whnf (LAM b, k)       =
    normalise3_cont (k, FUN b)
  | normalise3_whnf (APP (m, n), k) =
    normalise3_whnf (m, C1 (n, k))

(*  normalise3_nf : term * continuation
                    -> result_or_wrong *)
and normalise3_nf (IND n, k)       =
    normalise3_cont (k, ACC (n, []))
  | normalise3_nf (LAM b, k)       =
    normalise3_nf (b, C2 k)
  | normalise3_nf (APP (m, n), k) =
    normalise3_whnf (m, C3 (n, k))

(*  iterate3 : whnf_or_decomposition -> result_or_wrong *)
and iterate3 (WHNF w)              = RESULT w
  | iterate3 (DEC (SUB (b, n), k)) =
    (case k of (C1 (_, _) | C3 (_, _)) =>
               normalise3_whnf (subst (b, n, 0), k)
             | _                       =>
               normalise3_nf (subst (b, n, 0), k))

(*  normalise3 : term -> result_or_wrong *)
fun normalise3 t = normalise3_nf (t, C0)
```

The result is an optimised normaliser where adjacent `iterate3` and `refocus1` have been fused. The normaliser is now closer to a big-step tail-recursive implementation of an abstract machine.

5.5 Corridor transitions and inlining-of-iterate-function

Some configurations (states) have only one possible transition. We cut-and-paste the transition functions above, renaming their indices from 3 to 4:

```
normalise4_cont (C0, w)
= (* inlining normalise4_cont *)
iterate4 (WHNF w)
= (* inlining iterate4 *)
RESULT w
```

We must contract this corridor transition, i.e., inline `normalise4_cont` and `iterate4`. And then, we must inline `iterate4` inside `normalise4_cont`.

```
(*  normalise4_cont : continuation * whnf
                      -> result_or_wrong *)
fun normalise4_cont (C0, w)          = RESULT w
  | normalise4_cont (C1 (n, k), wm)  =
    (case wm
      of (FUN b) =>
         (case k of (C1 (_, _) | C3 (_, _)) =>
                    normalise4_whnf (subst (b, n, 0), k)
                  | _                       =>
                    normalise4_nf (subst (b, n, 0), k))
       | _       =>
         normalise4_cont (k, apply_acc (wm, n)))
  | normalise4_cont (C2 k, wb)       =
    normalise4_cont (k, FUN (embed wb))
  | normalise4_cont (C3 (n, k), wm)  =
    (case wm
      of (FUN b) =>
         (case k of (C1 (_, _) | C3 (_, _)) =>
                    normalise4_whnf (subst (b, n, 0), k)
                  | _                       =>
                    normalise4_nf (subst (b, n, 0), k))
       | _       =>
         normalise4_nf (embed wm, C4 (n, k)))
  | normalise4_cont (C4 (n, k), nm)  =
    normalise4_nf (n, C5 (k, nm))
  | normalise4_cont (C5 (k, nm), nn) =
    normalise4_cont (k, apply_acc (nm, embed nn))

(*  normalise4_whnf : term * continuation
                      -> result_or_wrong *)
and normalise4_whnf (IND n, k)       =
    normalise4_cont (k, ACC (n, []))
  | normalise4_whnf (LAM b, k)       =
    normalise4_cont (k, FUN b)
  | normalise4_whnf (APP (m, n), k) =
    normalise4_whnf (m, C1 (n, k))

(*  normalise4_nf : term * continuation
                    -> result_or_wrong *)
and normalise4_nf (IND n, k)       =
    normalise4_cont (k, ACC (n, []))
  | normalise4_nf (LAM b, k)       =
    normalise4_nf (b, C2 k)
  | normalise4_nf (APP (m, n), k) =
    normalise4_whnf (m, C3 (n, k))
```

Now the iterate function is unused. Observe that the case-expression introduced in the refocusing step (Section 5.2) is inlined twice in `normalise4_cont`. Consequently, the artefact obtained does not have the shallow inspection property because `normalise4_cont` pattern-matches on `k` at the inlining point.

5.6 Recovering the shallow inspection property

To pick the right normalising functions at the inlining points we have to find out which function is invoked with that `k` as the current continuation. Note that `C1` and `C3` are only pushed onto the stack by `normalise4_whnf` and `normalise4_nf` respectively. In both cases, `k` is the current continuation (last line of each function). No other functions can produce defunctionalised continuations `C1` and `C3`, and therefore the case expression can be safely removed and the appropriate normalising functions invoked: `normalise4_whnf` in the second clause of `normalise4_cont`, and `normalise4_nf` in the fourth clause.

```
(*  normalise5_cont : continuation * whnf
                      -> result_or_wrong *)
fun normalise5_cont (C0, w)          = RESULT w
  | normalise5_cont (C1 (n, k), wm) =
    (case wm
      of (FUN b) => normalise5_whnf (subst (b, n, 0), k)
       | _       => normalise5_cont (k, apply_acc (wm, n)))
```

T	$\rightarrow_{init}$	$(T, \mathbf{n}, \mathsf{C}_0)$
$(x, \mathbf{n}, S)$	$\rightarrow_{nf}$	$(x, \mathbf{c}, S)$
$(\lambda x.B, \mathbf{n}, S)$	$\rightarrow_{nf}$	$(B, \mathbf{n}, \mathsf{C}_2(x) :: S)$
$(M\,N, \mathbf{n}, S)$	$\rightarrow_{nf}$	$(M, \mathbf{w}, \mathsf{C}_3(N) :: S)$
$(x, \mathbf{w}, S)$	$\rightarrow_{whnf}$	$(x, \mathbf{c}, S)$
$(\lambda x.B, \mathbf{w}, S)$	$\rightarrow_{whnf}$	$(\lambda x.B, \mathbf{c}, S)$
$(M\,N, \mathbf{w}, S)$	$\rightarrow_{whnf}$	$(M, \mathbf{w}, \mathsf{C}_1(N) :: S)$
$(\lambda x.B, \mathbf{c}, \mathsf{C}_1(N) :: S)$	$\rightarrow_{cont}$	$([N/x]B, \mathbf{w}, S)$
$(M, \mathbf{c}, \mathsf{C}_1(N) :: S)$	$\rightarrow_{cont}$	$(M\,N, \mathbf{c}, S)$
$(B, \mathbf{c}, \mathsf{C}_2(x) :: S)$	$\rightarrow_{cont}$	$(\lambda x.B, \mathbf{c}, S)$
$(\lambda x.B, \mathbf{c}, \mathsf{C}_3(N) :: S)$	$\rightarrow_{cont}$	$([N/x]B, \mathbf{n}, S)$
$(M, \mathbf{c}, \mathsf{C}_3(N) :: S)$	$\rightarrow_{cont}$	$(M, \mathbf{n}, \mathsf{C}_4(N) :: S)$
$(M, \mathbf{c}, \mathsf{C}_4(N) :: S)$	$\rightarrow_{cont}$	$(N, \mathbf{n}, \mathsf{C}_5(M) :: S)$
$(N, \mathbf{c}, \mathsf{C}_5(M) :: S)$	$\rightarrow_{cont}$	$(M\,N, \mathbf{c}, S)$
$(T, \mathbf{c}, \mathsf{C}_0)$	$\rightarrow_{cont}$	T

Figure 7. Normal order abstract machine.

```
  | normalise5_cont (C2 k, wb)      =
    normalise5_cont (k, FUN (embed wb))
  | normalise5_cont (C3 (n, k), wm) =
    (case wm
      of (FUN b) => normalise5_nf (subst (b, n, 0), k)
       | _       => normalise5_nf (embed wm, C4 (n, k)))
  | normalise5_cont (C4 (n, k), nm) =
    normalise5_nf (n, C5 (k, nm))
  | normalise5_cont (C5 (k, nm), nn)=
    normalise5_cont (k, apply_acc (nm, embed nn))

(*  normalise5_whnf : term * continuation
                      -> result_or_wrong *)
and normalise5_whnf (IND n, k)      =
    normalise5_cont (k, ACC (n, []))
  | normalise5_whnf (LAM b, k)      =
    normalise5_cont (k, FUN b)
  | normalise5_whnf (APP (m, n), k) =
    normalise5_whnf (m, C1 (n, k))

(*  normalise5_nf : term * continuation
                    -> result_or_wrong *)
and normalise5_nf (IND n, k)      =
    normalise5_cont (k, ACC (n, []))
  | normalise5_nf (LAM b, k)      = normalise5_nf (b, C2 k)
  | normalise5_nf (APP (m, n), k) =
    normalise5_whnf (m, C3 (n, k))

(*  normalise5 : term -> result_or_wrong *)
fun normalise5 t = normalise5_nf (t, C0)
```

The shallow inspection property is recovered. The resulting normaliser is a big-step tail-recursive implementation of the abstract machine in Figure 7. We use a nameful representation for the machine, storing the formal parameter of an unapplied abstraction inside continuation C_2. We use the control characters **n**, **w** and **c** to indicate whether the configuration is *nf* (hybrid), *whnf* (subsidiary) or *cont* (dispatcher) respectively.

Our machine is a variant of the full-reducing Krivine machine KN [7]. KN is restricted to closed terms, is environment-based (following terminology in [5]), and proceeds with full reduction as soon as an accumulator is reached. Our machine admits open terms, is substitution-based, and reconstructs intermediate whnfs.

KN can be derived from our normal order machine! As follows: from the reduction-free normaliser (the one in Figure 2 to which we shall arrive at in Section 6) restricted to closed terms, apply closure conversion [5], adopt de Bruijn indices for term variables and de Bruijn levels for formal parameters in environments, like [7], and subsume substitution, evaluation and normalisation under one normaliser with explicit control. A detailed discussion of the derivation requires considerable space. We plan to include the derivation in an extended version of this paper. We refer the interested to the URL given in the introduction where the code with explanatory comments can be found.

6. From abstract machine to reduction-free normaliser

6.1 Refunctionalisation

The abstract machine in Section 5.5 is an instance of a defunctionalised CPS program, with a configuration for dispatching on the continuations (`normalise4_cont`) and two configurations for the hybrid and the subsidiary strategy (`normalise4_nf` and `normalise4_whnf`). By refunctionalising it we obtain a reduction-free normaliser in CPS.

```
(*  normalise6_whnf : term * (whnf -> 'a) -> 'a *)
fun normalise6_whnf (IND n, k)      = k (ACC (n, []))
  | normalise6_whnf (LAM b, k)      = k (FUN b)
  | normalise6_whnf (APP (m, n), k) =
    normalise6_whnf (m,
     fn wm => (case wm
                 of (FUN b) =>
                    normalise6_whnf (subst (b, n, 0), k)
                  | _       => k (apply_acc (wm, n))))

(*  normalise6_nf : term * (whnf -> 'a) -> 'a *)
and normalise6_nf (IND n, k)      = k (ACC (n, []))
  | normalise6_nf (LAM b, k)      =
    normalise6_whnf (b, fn wb => k (FUN (embed wb)))
  | normalise6_nf (APP (m, n), k) =
    normalise6_whnf (m,
     fn wm =>
        (case wm
          of (FUN b) => normalise6_nf (subst (b, n, 0), k)
           | _       =>
             normalise6_nf (m,
              fn nm =>
                 normalise6_nf (n,
                  fn nn
                     => k (apply_acc(nm, embed nn))))))

(*  normalise6 : term -> result_or_wrong *)
fun normalise6 t = normalise6_nf (t, fn s => RESULT s)
```

6.2 Back to direct style by inverse CPS transformation

```
(*  normalise7_whnf : term -> result_or_wrong *)
fun normalise7_whnf (IND n)      = ACC (n, [])
  | normalise7_whnf (LAM b)      = FUN b
  | normalise7_whnf (APP (m, n)) =
    let val wm = normalise7_whnf m
    in (case wm
         of (FUN b) => normalise7_whnf (subst (b, n, 0))
          | _       => apply_acc (wm, n))
    end

(*  normalise7_nf : term -> result_or_wrong *)
fun normalise7_nf (IND n)     = ACC (n, [])
  | normalise7_nf (LAM b)     =
    let val wb = normalise7_nf b
    in FUN (embed wb)
    end
  | normalise7_nf (APP (m, n)) =
    let val wm = normalise7_whnf m
    in (case wm
         of (FUN b) => normalise7_nf (subst (b, n, 0))
          | _       =>
            let val nm = normalise7_nf (embed wm)
                val nn = normalise7_nf (n)
```

```
            in apply_acc (nm, embed nn)
            end)
    end

(*  normalise7_term : term -> result_or_wrong *)
fun normalise7 t = RESULT (normalise7_nf t)
```

Save for the ancillary `result_or_wrong` datatype and the init function `normalise6`, this is the canonical reduction-free normaliser in Figure 2, with `normalise6_whnf` corresponding to `bn` and `normalise6_nf` corresponding to `no`. This establishes the correspondence.

7. Related and future work

We have already commented at length on [IT] throughout the paper and particularly in the contributions. As discussed in Section 5.5, we have derived from our normal order machine (Figure 7) the full-reducing Krivine machine KN [7]. A detailed discussion of the derivation requires considerable space. We plan to include the derivation in an extended version of this paper. But the code with explanatory comments can be downloaded from the URL given in the introduction.

In [15], a full-reducing hybrid strategy $N()$ is specified in eval-readback style. The subsidiary strategy $V()$ is implemented by an optimised, pre-compiled abstract machine. This machine has been contrived, not derived. This paper opens up the possibility of deriving machines for $N()$. A question to answer is whether optimisations can be incorporated by program transformation.

We have applied the techniques in this paper to the interderivation of small-step and big-step artefacts for full-reducing strategies of the lambda-value calculus, in particular the counterpart of normal order in that calculus. We hope to publish this results elsewhere.

8. Conclusions

The interderivation techniques [IT] can be refined to accommodate hybrid strategies, of which the full-reducing (and their machines) are the most interesting. The insight is to use a single datatype for hybrid and subsidiary artefacts, and to notice the dependency between the value constructors of defunctionalised continuations and which normalising function has to continue. By definition, a hybrid strategy 'contains' a subsidiary strategy. Small-step-wise, the subsidiary is a subrelation. Big-step-wise, the subsidiary is a left identity. Therefore, it is in general possible to use a single datatype for hybrid and subsidiary artefacts, and to embed the hybrid's irreducible forms in the subsidiary's.

Acknowledgments

We are grateful to Oliver Danvy for his hospitality and advice, and to Aarhus University for hosting the first author for a three-month stay during which he attended Olivier's excellent and inspiring dTFP course on interderivation techniques. We are also grateful to Jan Midtgaard and to the anonymous reviewers for valuable feedback on the contents of this paper. The expert reviewer in particular made some important points that we hope to have addressed properly.

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of International Conference on Principles and Practice of Declarative Programming*, pages 8–19, 2003.

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report RS-03-14, BRICS, Department of Computer Science, Aarhus University, Denmark, Mar. 2003.

[3] 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, sep 2005.

[4] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1984.

[5] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Log*, 9(1):6:1–6:29, Dec. 2007.

[6] P. Crégut. An abstract machine for lambda-terms normalization. In *Proceedings of LISP and Functional Programming*, pages 333–340, 1990.

[7] P. Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, Sept. 2007.

[8] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[9] O. Danvy. From reduction-based to reduction-free normalization. *Electr. Notes. Theor. Comput. Sci*, 124(2):79–100, 2005.

[10] O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Inf. Process. Lett*, 106(3):100–109, 2008.

[11] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Technical Report RS-04-26, BRICS, Department of Computer Science, Aarhus University, Denmark, Nov. 2004.

[12] O. Danvy, J. Johannsen, and I. Zerny. A walk in the semantic park. In *Proceedings of Workshop on Partial Evaluation and Program Manipulation*, pages 1–12, 2011.

[13] M. Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, 1987.

[14] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Proceedings of International Conference on Functional Programming*, pages 18–27, 1999.

[15] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of International Conference on Functional Programming*, pages 235–246, 2002.

[16] G. Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[17] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[18] S. Ronchi Della Rocca and L. Paolini. *The Parametric Lambda Calculus*. Springer Verlag, Feb. 2004.

[19] P. Sestoft. Demonstrating lambda calculus reduction. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 420–435. Springer, 2002.

[20] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.