Formal Development for a Node Replication Calculus with Abstract Machine Extraction for a Lazy Strategy

Felipe A. Costa[1], **Daniel Ventura**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia, GO – Brazil

felipeac@egresso.ufg.br, ventura@ufg.br

Abstract. *Node-by-node replication is related with the implementation of optimal graph-based reduction for the λ-calculus and its associated substitution mechanism was recently identified as the Curry-Howard interpretation of deep-inference. A lazy (weak) strategy was defined for the node-replication calculus, with full-laziness and an observationally equivalent relation equivalent to the same relation for a (weak) call-by-name strategy. A node-replication calculus with a fully lazy call-by-need reduction strategy is specified in the Coq Proof Assistant (currently known as Rocq), following an approach based on refocusing, allowing the automatic extraction of the corresponding abstract machine.*

1. Introduction

The node-replication calculus in [Kesner et al. 2024] has a fully lazy call-by-need reduction strategy, *i.e.* the number of steps to compute an answer is the same as the shortest path in a (weak) strategy. Reduction strategies define how some term is reduced, corresponding to how a program in the functional paradigm is executed [Diehl et al. 2000]. An abstract machine is a theoretical model that allows for step-by-step execution of programs, where single steps operations on the state of computations can be defined by a rewriting system [Hannan and Miller 1992]. For instance, Haskell is a well-known functional language with its abstract machine implementing a lazy strategy [Jones 1992]. Refocusing allows the extraction of the abstract machine from the calculus specification which includes its reduction strategy (see [Biernacka and Charatonik 2019] and references therein). We present a specification of a node-replication calculus based on [Kesner et al. 2024] in the Coq Proof Assistant (currently known as Rocq)[1]. We follow the approach in [Biernacka and Charatonik 2019], using the refocusing procedure, to extract the corresponding abstract machine.

Contributions To the best of our knowledge, the current development is the first formalisation of a node-by-node replication calculus. Once the properties necessary to allow machine extraction are proved, the following properties hold for the specified calculus: strategy determinism and a characterisation of the strategy normal forms. However, a formal relation between the lazy strategy in [Kesner et al. 2024] and the one introduced in the present work has yet to be investigated.

The document has the following structure: Sec. 2 presents some general background to reduction strategies, the node-replication calculus with related notions, and refocusing; Sec. 3 introduces our node-replication calculus based in [Kesner et al. 2024];

[1]https://rocq-prover.org/

in Sec. 4 key aspects of the Coq development are discussed and finally Sec. 5 concludes with a discussion of future work.

The complete formal development (Coq Version 8.11.1) is available at `https://github.com/felipeagc/generalized_refocusing/blob/master/examples/node_replication.v`.

2. Background

We assume the reader is familiar with the λ-calculus [Barendregt 1985] and with **explicit substitution calculi** [Kesner 2009], where the syntax is extended with terms of the form $t[x\backslash u]$, denoting a term t with a pending substitution, which can also be interpreted as a **sharing device**, where u or any computation from it is shared by all free occurrences of x in t. Terms with no explicit substitutions are called **pure terms**.

Reduction Strategies Reductions can happen at any place in a term in the general theory, *i.e.* any subterm which is a redex can be contracted. However, different **reduction strategies** can be defined, restricting where a reduction can take place[2]. For instance, the **call-by-name** strategy does not reduce an argument before the function application, *i.e.* before the corresponding β-redex contraction, while the **call-by-value** strategy applies a function only to **values**, *i.e.* terms which are either a variable[3] or an abstraction. Functional language implementations need to define a (deterministic) reduction strategy, executed by its abstract machine. **Context-based reductions** is one way to define it. A **context** is defined as a term with a (unique) hole: $\mathtt{C} ::= \Diamond \mid \lambda x.\mathtt{C} \mid \mathtt{C}t \mid t\mathtt{C}$, where $\mathtt{C}\langle t\rangle$ denotes a context $\mathtt{C}$ where its hole is filled with term t. **Elementary contexts** are such that context-variables correspond to $\Diamond$, *e.g.* elementary $\mathtt{C}$-contexts are $\Diamond$, $\lambda x.\Diamond$, $\Diamond t$ and $t\Diamond$. Reductions in the general theory can then be defined by the closure of the β-reduction rule ($\mapsto_\beta$) for $\mathtt{C}$-contexts, *i.e.* a term t is reducible if $t = \mathtt{C}\langle(\lambda x.u)r\rangle$ for some context $\mathtt{C}$. The (weak) call-by-name strategy is defined as the closure of $\mapsto_\beta$ by $\mathtt{D}$-contexts: $\mathtt{D} ::= \Diamond \mid \mathtt{D}\,t \mid n\,\mathtt{D}$, where n denotes a (weak) **neutral normal-form**, *i.e.* a weak normal form which is not an abstraction. The strategy is called **weak** because no reduction under λ-abstractions is allowed. The **call-by-need** strategy [Ariola and Felleisen 1997] is a lazy strategy that combines the advantages of call-by-name and call-by value. Some **memoisation technique** is applied in this strategy, in order to avoid recalculations of partial results used more than once during computations. We use explicit substitutions, also called explicit cuts, as a memory device in the current work.

Node Replication Node-by-node replication is the Curry-Howard interpretation of deep inference [Kesner et al. 2024], where substitutions are executed constructor by constructor. Node replication was originally introduced to implement optimal graph-based reduction for the λ-calculus [Lamping 1990] and the first Curry-Howard interpretation was the so-called **atomic λ-calculus** [Gundersen et al. 2013]. The atomic λ-calculus is an explicit resource calculus where, besides explicit substitution, both weakening –corresponding to garbage collection– and contraction –dealing with term duplications– are also handled through explicit constructors/devices. Investigation of properties of the calculus and for different reduction strategies –as an application to concrete implementations of programming languages– is very difficult. Therefore, in [Kesner et al. 2024] a **node replication**

[2] each strategy has a corresponding set of normal-forms

[3] some definitions do not consider variables as values

calculus with implicit weakening and contraction was defined, allowing several properties to be established for the node-replications paradigm, including the investigations of two different (weak) reduction strategies: call-by-name and call-by-need. **Full laziness** of the call-by-need strategy for pure terms, when a normal-form is obtained with the same number of β-reduction steps as in the shortest (weak) reduction in the pure λ-calculus, is achieved through an operation called **skeleton extraction**. A **skeleton** of $\lambda x.t$ is the minimal term structure necessary to keep the same binding relation between λx and free occurrences of x in t. For instance, let $t = (\lambda x.x\,x)u \mapsto_{\mathtt{dB}} (x\,x)[x\backslash u]$, where $u = (\lambda z.z(\mathtt{I}\,\mathtt{I}))$ with $\mathtt{I} = \lambda x.x$. Term u cannot be reduced in a weak strategy and replacing both occurrences of x by u will duplicate the redex $(\mathtt{I}\,\mathtt{I})$, both executed in a later stage. Instead, u is split in two components: term $\lambda z.zy$, the skeleton of u, and $\{(\mathtt{I}\,\mathtt{I})\}$, a multiset of terms with no free occurrence of z, called the **maximal free expressions** (MFE). Term $(x\,x)[x\backslash(\lambda z.z(\mathtt{I}\,\mathtt{I}))]$ is then reduced to $((\lambda z.zy)(\lambda z.zy))[y\backslash(\mathtt{I}\,\mathtt{I})]$, where only the skeleton is duplicated while maintaining the redex $(\mathtt{I}\,\mathtt{I})$ shared. Skeleton extraction was already used for full laziness in [Ariola and Felleisen 1997] but extraction was defined as a meta-operation while in [Kesner et al. 2024] the operation was defined as a substrategy in the calculus. Besides explicit substitutions, the syntax of terms is then extended with the **distributors** to deal with shared abstractions: $t[x\backslash\backslash\lambda y.u]$. Explicit substitutions and distributors are called **(explicit) cuts**, where $t[x \triangleleft u]$ denotes both.

Refocusing Introduced as a general approach to extract abstract machines from context-based reduction semantics, refocusing was specified with Coq, with an automated machine extraction from a reduction semantics satisfying some syntactical properties (see [Biernacka and Charatonik 2019] and references therein). Reduction in a context-based calculus depends on the term decomposition in a reduction context and a redex. Refocusing is based on keeping a stack of elementary contexts built while processing the term to identify the current redex, with a continuation of the decomposition/recomposition procedure from its contractum, avoiding a rework while identifying the next redex. For instance, let $t = (\lambda x.\lambda y.y\,x)u\,v\,w\,s$ and the weak call-by-name strategy with the $\mathtt{D}$-contexts defined above. Then we decompose $t = \mathtt{D}\langle r_0\rangle$ where $\mathtt{D} = \Diamond\,v\,w\,s$ and $r_0 = (\lambda x.\lambda y.y\,x)u$; contracts $r_0 \mapsto_\beta \lambda y.y\,u = r_0'$; and recompose $t' = \mathtt{D}\langle r_0'\rangle$. The decomposition stage in the next step holds $t' = \mathtt{D}'\langle r_1\rangle$ where $\mathtt{D}' = \Diamond\,w\,s$ and $r_1 = r_0'\,v$, where the new redex r_1 is contracted. Decomposition can be obtained through an iteration of a process splitting a term in a subterm and an elementary context (e.c.), until a redex or a "stuck term", *i.e.* a normal form, is achieved. Considering t as above, a first iteration would hold the pair $(\lambda x.\lambda y.y\,x)u\,v\,w$ and $\Diamond s$ while the full iteration holds r_0 as before and $[\Diamond v; \Diamond w; \Diamond s]$, a stack of e.c.'s obtained in each iteration. The reduct t' is then obtained plugging back contractum r_0', and the resulting terms, in e.c.'s popped from the stack. Decomposition of t' then holds the redex r_1 and the stack $[\Diamond w; \Diamond s]$. Therefore, instead of the recomposition to obtain t' as described above, and the subsequent decomposition of t', refocusing proceeds from the pair r_0 and $[\Diamond v; \Diamond w; \Diamond s]$ as follows:

1. after $r_0 \mapsto_\beta r_0'$, decomposition continues from r_0', identified as non-decomposable;
2. r_0' is then plugged back in the e.c. in the top of the stack, where the redex r_1 is identified;
3. $r_1 \mapsto_\beta v\,u = r_1'$, with decomposition continuing from r_1'.

Hence, the first two iterations while decomposing both t and t' are executed only once. An axiomatisation of a reduction semantics was proposed and proved sufficient to auto-

mate the extraction of an abstract machine equivalent to the (reduction) evaluator. Those axioms must be satisfied by the calculus specification provided by the user. For example, some syntactic categories must be defined in the specification: terms, values, potential redexes and e.c.'s (then called context frames). Properties about decomposition need to be satisfied, resulting in the following three cases for any term t decomposition: (1) $t = r$, a non-decomposable redex; (2) $t = v$, a non-decomposable value; (3) $t = \mathtt{C}\langle t' \rangle$, with $\mathtt{C}$ an e.c.. As illustrated by the example above, such a decomposition is iterated until a non-decomposable term is reached. If it is a redex, then a contraction is executed and decomposition applied in its result. If it is a value v, then the e.c. stack is analysed where:

1. v is the final answer, if the stack is empty;
2. v is plugged back into the e.c. on the top of the stack, with the resulting term re-analysed with three possible outcomes: a redex, a value or it is decomposable in a pair of a term and a new e.c..

Decomposition and value recomposition functions, as described above, must be provided by the user. Uniqueness of decompositions is a consequence of several properties proved to be satisfied by the provided functions, where strict orders for term and contexts –the latter also provided by the user– are considered. Once all properties are checked by the user, the automatic extraction of the corresponding abstract machine can be applied.

However, the so-called hybrid strategies, such as the normal order evaluation [Barendregt 1985], could not be handled by the approach. Thus, a generalisation of the refocusing procedure was presented and applied in [Biernacka and Charatonik 2019] for both weak and strong call-by-need calculi, the former a uniform strategy while the latter is hybrid. The present work follows this specification approach.

3. Towards a Lazy Node-Replication Machine

The weak call-by-need calculus in [Biernacka and Charatonik 2019] uses two explicit substitution constructors to differentiate when a substitution term needs to be evaluated. Such construct is called an **active** or a **strict substitution**, denoted by $t[\![x \backslash u]\!]$. The original strong call-by-need in [Balabonski et al. 2017] has no such a constructor but the strong strategy defined in [Biernacka and Charatonik 2019] uses the same approach. We follow this approach in the node-replication calculus specification. The **syntax of term expressions** from [Kesner et al. 2024] is extended to include **active cuts** $t[\![x \triangleleft u]\!]$:

(Terms) $t, u \ ::= \ x \mid \lambda x.t \mid tu \mid t[x \backslash u] \mid t[x \backslash\backslash \lambda y.u] \mid t[\![x \backslash u]\!] \mid t[\![x \backslash\backslash \lambda y.u]\!]$

(Term Values) $v \ ::= \ \lambda x.t$ **(List Contexts)** $\mathtt{L} \ ::= \ \Diamond \mid \mathtt{L}[x \backslash u] \mid \mathtt{L}[x \backslash\backslash \lambda y.u]$

with the corresponding **needy** reduction **contexts**: $\mathtt{N} ::= \Diamond \mid \mathtt{N}t \mid \mathtt{N}[x \triangleleft t] \mid \mathtt{N}\langle\langle x \rangle\rangle [\![x \backslash \mathtt{N}]\!]$. **Reduction rules** for the extended syntax is presented below, where the **relation** $\Downarrow^{\theta}$ used for skeleton extraction, with θ a set of variables, is presented in Fig. 1:

$$
\begin{array}{llll}
\mathtt{L}\langle \lambda x.t \rangle u & \mapsto_{\mathtt{dB}} & \mathtt{L}\langle t[x \backslash u] \rangle & \\
\mathtt{N}\langle\langle x \rangle\rangle [x \backslash t] & \mapsto_{\mathtt{Spl}} & \mathtt{N}\langle\langle x \rangle\rangle [\![x \backslash t]\!] & \\
\mathtt{N}\langle\langle x \rangle\rangle [\![x \backslash \mathtt{L}\langle \lambda y.t \rangle]\!] & \mapsto_{\mathtt{SplS}} & \mathtt{L}\langle \mathtt{L}'\langle \mathtt{N}\langle\langle x \rangle\rangle [\![x \backslash\backslash \lambda y.t']\!] \rangle \rangle, & \text{if } t \Downarrow^{\{y\}} \mathtt{L}'\langle t' \rangle \\
\mathtt{N}\langle\langle x \rangle\rangle [x \backslash\backslash v] & \mapsto_{\mathtt{Ls}} & \mathtt{N}\langle\langle x \rangle\rangle [\![x \backslash\backslash v]\!] & \\
\mathtt{N}\langle\langle x \rangle\rangle [\![x \backslash\backslash v]\!] & \mapsto_{\mathtt{LsS}} & \mathtt{N}\langle\langle v \rangle\rangle [x \backslash\backslash v] &
\end{array}
$$

The reduction strategy in [Kesner et al. 2024] is considered on a restricted set of terms

$$\frac{x \text{ fresh}}{p \Downarrow^{\theta} x[x\backslash p]} \quad \text{if } \mathrm{fv}(p) \cap \theta = \varnothing\text{; otherwise:}$$

$$\frac{}{x \Downarrow^{\theta} x} \qquad \frac{p \Downarrow^{\theta \cup \{x\}} \mathtt{L}\langle p' \rangle}{\lambda x.p \Downarrow^{\theta} \mathtt{L}\langle \lambda x.p' \rangle} \qquad \frac{p \Downarrow^{\theta} \mathtt{L}_1\langle p' \rangle \qquad q \Downarrow^{\theta} \mathtt{L}_2\langle q' \rangle}{pq \Downarrow^{\theta} \mathtt{L}_2\langle \mathtt{L}_1\langle p'q' \rangle \rangle}$$

Figure 1. Skeleton Extraction Big-Step Semantics

while here the reduction strategy is considered for any term originated from a reduction starting from a pure term, *i.e.* a term without explicit cuts. Normal-forms are expected to be one of two kinds, as in [Kesner et al. 2024]:

(Needy Terms) $n^x ::= x \mid n^x t \mid n^x[y \triangleleft t] \mid n^y[\![y\backslash n^x]\!]$ **(Answers)** $a ::= v \mid a[x \triangleleft t]$

Note that n^x is a needy term iff exists a needy context $\mathtt{N}$ s.t. $n^x = \mathtt{N}\langle\langle x \rangle\rangle$ and a is an answer iff exists a term value v and a list context $\mathtt{L}$ s.t. $a = \mathtt{L}\langle v \rangle$. Following the refocusing approach, all normal-forms are considered to be **values** in the formal development. Needy terms are used to syntactically restrict active cuts to be of the form $n^x[\![x \triangleleft u]\!]$, then called **strict cuts**. Differently from [Biernacka and Charatonik 2019] where needy terms play the role of intermediate results only, the current strategy considers them as final results since open terms, *i.e.* terms with free variables, are allowed.

As noted in Sec. 2, refocusing uses a stack of elementary contexts (e.c.) to avoid recalculations. We now define the two functions used as inputs in the procedure.

Definition 3.1 (⇓/⇑ Functions)**.** *Down (⇓) and Up (⇑) functions –where* $\mathtt{V}_{\_}$ *and* $\mathtt{R}$ *indicates a value and a redex, respectively– are defined by:*

$$\begin{aligned}
x &\Downarrow \mathtt{V}_n \\
\lambda x.t &\Downarrow \mathtt{V}_a \\
t_1 t_2 &\Downarrow (t_1, \Diamond\, t_2) \\
t_1[x \triangleleft t_2] &\Downarrow (t_1, \Diamond\, [x \triangleleft t_2]) \\
t_1[\![x\backslash t_2]\!] &\Downarrow (t_2, t_1[\![x\backslash \Diamond]\!]) \\
t_1[\![x\backslash\backslash t_2]\!] &\Downarrow \mathtt{R}
\end{aligned}
\qquad
\begin{aligned}
\langle n^- \rangle t &\Uparrow \mathtt{V}_n \\
\langle a \rangle t &\Uparrow \mathtt{R} \\
\langle a \rangle [x \triangleleft t] &\Uparrow \mathtt{V}_a \\
\langle n^x \rangle [x \triangleleft t] &\Uparrow \mathtt{R} \\
\langle n^y \rangle [x \triangleleft t] &\Uparrow \mathtt{V}_n, \textit{if } x \neq y \\
n^x[\![x\backslash \langle a \rangle]\!] &\Uparrow \mathtt{R} \\
n^x[\![x\backslash \langle n^- \rangle]\!] &\Uparrow \mathtt{V}_n
\end{aligned}$$

Values can either be an answer ($\mathtt{V}_a$) or a needy term ($\mathtt{V}_n$). Intuitively, decomposition is achieved iterating ⇓ until either a redex or a value is obtained, the latter triggering the application of ⇑, recomposing values in order to identify the next redex.

Example 3.1 (Refocusing Procedure)**.** *Let* $t = (\lambda x.x)rs$ *then:* $t \Downarrow ((\lambda x.x)r, \Diamond\, s)$*;* $(\lambda x.x)r \Downarrow (\lambda x.x, \Diamond\, r)$*;* $\lambda x.x \Downarrow \mathtt{V}_a$*, i.e. the decomposition procedure holds a pair* $(\lambda x.x, \Diamond\, r; \Diamond\, s)$ *of a term, already identified as an answer* a*, and a stack of e.c..* ⇑ *is then applied to the recomposed term:* $\langle \lambda x.x \rangle r \Uparrow \mathtt{R}$*. Once the redex is identified, the corresponding reduction rule is applied:* $(\lambda x.x)r \mapsto_{\mathtt{dB}} x[x\backslash r]$*. Decomposition restarts from* $x[x\backslash r]$*, holding* $(x, \Diamond\, [x\backslash r]; \Diamond\, s)$ *with* $x \Downarrow \mathtt{V}_n$*, triggering* ⇑ *once again.*

4. Coq Formal Development

The formal development for the calculus in Sec. 3, allowing an extraction of the corresponding abstract machine, is based on the weak call-by-need specification in

[Biernacka and Charatonik 2019][4]. The original codebase provides a framework for specifying and extracting the abstract machine utilizing the so-called generalised refocusing procedure. It was implemented using Coq version 8.11.1, also used in the present work[5].

The implementation is split into two modules, one implementing the reduction semantics (satisfying the `PRE_REF_SEM` module signature), and one implementing the lower-level reduction strategy (satisfying the signature returned by `REF_STRATEGY`). Starting with the module defining the reduction strategy –called `Lam_cbnd_PreRefSem`–, first thing to be defined are the kinds of contexts considered in the strategy and, since the current strategy is uniform, the only kind of context is `N`. Among the adaptations necessary to implement the node replication strategy, we extended the `expr` definition to include explicit distributors, along with their strict variants. The same was done to the `needy` type definition, where a needy term n^x has the dependent type `needy x`. For instance, term $y[y\backslash u]$ can be encoded as `ExpSubst (Var Y) Y U` where `Y = (Id 2)`, an encoding of y as a name, and `U` denotes the encoding of term u. Moreover, $y[\![y\backslash u]\!]$ is encoded as `ExpSubstS Y (nVar Y) U`. Note that variable y as the body of an explicit cut is encoded differently in each case, being a term — with type `expr` — in the former and a needy variable — with type `needy Y` — in the latter.

The `value` (dependent) type is one of the must-do definitions, based here on term values with type `val N`, answers as defined in Sec. 3 and needy terms:

```
Inductive val : ckind -> Type :=
  | vLam : forall {k}, var -> term -> val k.

Inductive answer : ckind -> Type :=
  | ansVal : val N -> sub -> answer N
  | ansNd : forall x, needy x -> answer N.

Definition value := answer.
```

A needy term m^y is encoded as `ansNd Y M` and an answer $a = \mathtt{L}\langle\lambda x.t\rangle$ is encoded as `ansVal (vLam X T) L`, where `L` –with type `sub`– is the encoding of `L` as a list of explicit cuts. Parametrisation of `answer` and thus `value` by context kinds is due to the possibility of different values to be considered in different (sub-)strategies in a hybrid setting. A notable difference from the original implementation is our use of lists of cuts instead of the context-like `ansCtx` definition in the original implementation. The main difference is that `ansCtx` type is also parametrised by context kinds while the type `sub` of lists of cuts is not. Redices are defined as in Sec. 3:

```
Inductive red : ckind -> Type :=
  | rApp  : forall {k}, val k -> sub -> term -> red k
  | rSplS : forall {k} x, needy x -> val k -> sub -> red k
  | rSpl  : forall {k} x, needy x -> term -> red k
  | rLsS  : forall {k} x, needy x -> val k -> red k
  | rLs   : forall {k} x, needy x -> val k -> red k.

Definition redex := red.
```

The type definition for e.c. `eck` was also adapted to match the definition of e.c. in Sec. 3:

```
Inductive eck : ckind -> ckind -> Type :=
| eckApp : forall {k1 k2}, term -> eck k1 k2
| eckSubst : forall {k1 k2}, var -> term -> eck k1 k2
| eckDist : forall {k1 k2}, var -> var -> term -> eck k1 k2
| eckPlugSubst : forall {k1 k2} x, needy x -> eck k1 k2.
```

[4] see `https://bitbucket.org/pl-uwr/generalized_refocusing`

[5] Full development at `https://github.com/felipeagc/generalized_refocusing/blob/master/examples/node_replication.v`.

The key departure from [Biernacka and Charatonik 2019] is the implementation of the `skl_extract` function, which encodes the skeleton extraction defined in Fig. 1. We needed auxiliary functions to implement this step, such as `fv` for obtaining the set of free variables in an expression, as well as `fresh_ind` to obtain an index corresponding to a fresh variable for an expression. The contraction rules are defined as shown in Sec. 3, with the skeleton extraction being applied in the `rSplS` case:

```
Definition contract {k} (r : redex k) : option term :=
match r with
| rApp (vLam x t) l u => Some (sub_to_term l (ExpSubst t x u))
| rSplS x nx (vLam (Id y) t) l =>
    let '(l', p', _) :=
      skl_extract t (S.singleton y) (1 + (fresh_ind t))
    in
      Some (sub_to_term l (sub_to_term l' (ExpDistS x nx (Id y) p')))
| rSpl x nx t => Some (ExpSubstS x nx t)
| rLsS x nx (vLam y p) =>
    Some (ExpDist (subst_needy x nx (@vLam k y p)) x y p)
| rLs x nx (vLam y p) =>
    Some (ExpDistS x nx y p)
end.
```

Now we define the second module –called `Lam_cbn_Strategy`–, which satisfies the signature returned by `REF_STRATEGY` with module `Lam_cbnd_PreRefSem` passed as a parameter. For instance, definition `dec_term` specifies the $\Downarrow$ function:

```
Definition dec_term t k : elem_dec k :=
    match k with N =>
      match t with
      | App t1 t2 => ed_dec N t1 (eckApp t2)
      | Var x     => ed_val (ansNd _ (nVar x))
      | Lam x t1  => ed_val (ansVal (vLam x t1) subEmpty)
      | ExpSubst t1 x t2 =>  ed_dec N t1 (eckSubst x t2)
      | ExpSubstS x nx t =>  ed_dec N t (eckPlugSubst x nx)
      | ExpDist t x y u =>  ed_dec N t (eckDist x y u)
      | ExpDistS x nx y u => ed_red (rLsS x nx (vLam y u))
      end
    end.
```

The automated abstract machine extraction is based on calculi satisfying some properties, as briefly discussed in Sec 2. Therefore, properties such as a value cannot be a redex (Lemma `value_redex`), a redex does not contain another redex when considering the reduction strategy (Lemma `redex_trivial1`), and some properties which guarantee uniqueness of a term decomposition –resulting that the strategy is deterministic– (*e.g.* Lemmas `search_order_comp_if` and `dec_context_term_next`), must be proved in order to be able to execute the extraction procedure.

Once both `Lam_cbnd_PreRefSem` and `Lam_cbn_Strategy` modules are defined, we can produce the abstract machine. First, by passing these modules to the functor `RedRefSem`, we produce a module defining the refocusable semantics. More specifically, it automatically proves crucial decomposition lemmas required by refocusing. The functor `RefEvalApplyMachine` is then applied, producing the abstract machine.

5. Conclusion

A node-replication calculus based in [Kesner et al. 2024] was specified in the Coq Proof Assistant (currently known as Rocq), following the refocusing procedure, allowing the extraction of the corresponding abstract machine. Our formalisation was implemented using

Coq version 8.11.1, with the framework presented in [Biernacka and Charatonik 2019] [6]. To the best of our knowledge, the current development is the first formalisation of a node-by-node replication calculus. The calculus strategy is proved to be deterministic, and normal forms are characterised by the needy terms and answers as defined in Sec. 3.

As future work, there are several properties to be checked in order to establish a formal relation between the $\rightarrow_{\texttt{flneed}}$ strategy in [Kesner et al. 2024] and the one in Sec. 3. One property to be checked is if our calculus is closed to the restricted class of the U-terms in [Kesner et al. 2024], even without syntax restrictions present in the reduction rules definitions for the $\rightarrow_{\texttt{flneed}}$ strategy. One important detail that needs further investigation is the generation of fresh variables considered in the function computing the skeleton extraction. Freshness is guaranteed locally, and one has to check if this local freshness is sufficient to guarantee the calculus consistency. Yet another consideration about skeleton extraction, a function implementing its big step semantics was used in the current development, differently from the extraction considered in $\rightarrow_{\texttt{flneed}}$, where a small-step semantics in the calculus is used to implement the extraction. It remains to be checked if our calculus can be specified with skeleton extraction as a substrategy.

References

Ariola, Z. M. and Felleisen, M. (1997). The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301.

Balabonski, T., Barenbaum, P., Bonelli, E., and Kesner, D. (2017). Foundations of strong call by need. *Proc. ACM Program. Lang.*, 1(ICFP):20:1–20:29.

Barendregt, H. P. (1985). *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland.

Biernacka, M. and Charatonik, W. (2019). Deriving an abstract machine for strong call by need. In *FSCD*, volume 131 of *LIPIcs*, pages 8:1–8:20. Schloss Dagstuhl - LZI.

Diehl, S., Hartel, P. H., and Sestoft, P. (2000). Abstract machines for programming language implementation. *Future Gener. Comput. Syst.*, 16(7):739–751.

Gundersen, T., Heijltjes, W., and Parigot, M. (2013). Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In *LICS*, pages 311–320. IEEE CS.

Hannan, J. and Miller, D. (1992). From operational semantics for abstract machines. *Math. Struct. Comput. Sci.*, 2(4):415–459.

Jones, S. L. P. (1992). Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. Funct. Program.*, 2(2):127–202.

Kesner, D. (2009). A theory of explicit substitutions with safe and full composition. *Log. Methods Comput. Sci.*, 5(3).

Kesner, D., Peyrot, L., and Ventura, D. (2024). Node replication: Theory and practice. *Log. Methods Comput. Sci.*, 20(1).

Lamping, J. (1990). An algorithm for optimal lambda calculus reduction. In *POPL*, pages 16–30. ACM Press.

[6] see `https://bitbucket.org/pl-uwr/generalized_refocusing`