UNIVERSIDADE FEDERAL DE GOIÁS
INSITUTO DE INFORMÁTICA

FELIPE AGUIAR COSTA

Formalização de um Cálculo de Replicação de Nós com Extração da Máquina Abstrata para uma Avaliação Preguiçosa

Goiânia
2024

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TRABALHO DE CONCLUSÃO DE CURSO DE GRADUAÇÃO NO REPOSITÓRIO INSTITUCIONAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio do Repositório Institucional (RI/UFG), regulamentado pela Resolução CEPEC no 1240/2014, sem ressarcimento dos direitos autorais, de acordo com a Lei no 9.610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

O conteúdo dos Trabalhos de Conclusão dos Cursos de Graduação disponibilizado no RI/UFG é de responsabilidade exclusiva dos autores. Ao encaminhar(em) o produto final, o(s) autor(a)(es)(as) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do Trabalho de Conclusão de Curso de Graduação (TCCG)

Nome(s) completo(s) do(a)(s) autor(a)(es)(as): Felipe Aguiar Costa

Título do trabalho: Formalização de um Cálculo de Replicação de Nós com Extração da Máquina Abstrata para uma Avaliação Preguiçosa

2. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador) Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante: a) consulta ao(à)(s) autor(a)(es)(as) e ao(à) orientador(a); b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo do TCCG. O documento não será disponibilizado durante o período de embargo.

Casos de embargo:

- Solicitação de registro de patente;

- Submissão de artigo em revista científica;

- Publicação como capítulo de livro.

Obs.: Este termo deve ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **Felipe Aguiar Costa**, **Usuário Externo**, em 04/06/2025, às 18:17, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Daniel Lima Ventura**, **Professor do Magistério Superior**, em 04/06/2025, às 18:52, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5417382** e o código CRC **D7773906**.

Referência: Processo nº 23070.062620/2024-90 SEI nº 5417382

FELIPE AGUIAR COSTA

Formalização de um Cálculo de Replicação de Nós com Extração da Máquina Abstrata para uma Avaliação Preguiçosa

Trabalho de Conclusão apresentado à Coordenação do Curso de Ciência da Computação do Insituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Daniel Ventura

Goiânia
2024

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Costa, Felipe Aguiar
Formalização de um Cálculo de Replicação de Nós com Extração da Máquina Abstrata para uma Avaliação Preguiçosa [manuscrito] / Felipe Aguiar Costa. - 2024.
47 f.

Orientador: Prof. Dr. Daniel Lima Ventura.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Goiás, Instituto de Informática (INF), Ciência da Computação, Goiânia, 2024.
Bibliografia. Apêndice.

1. máquina abstrata. 2. replicação nó-a-nó. 3. avaliação preguiçosa. 4. refocusing. I. Ventura, Daniel Lima, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

BANCA EXAMINADORA

FELIPE AGUIAR COSTA

FORMALIZAÇÃO DE UM CÁLCULO DE REPLICAÇÃO DE NÓS COM EXTRAÇÃO DA MÁQUINA ABSTRATA PARA UMA AVALIAÇÃO PREGUIÇOSA

Trabalho de conclusão de curso apresentado à Universidade Federal de Goiás como parte dos requisitos para a obtenção do título de Bacharel em Ciência da computação.
Orientador: Prof. Dr.: Daniel Lima Ventura

Aprovado em 13/12/2024.

Examinadores:

Prof. Dr. Daniel Lima Ventura
Universidade Federal de Goiás
Instituto de Informática

Prof. Dr. Bruno Oliveira Silvestre
Universidade Federal de Goiás
Instituto de Informática

Documento assinado eletronicamente por **Daniel Lima Ventura**, **Professor do Magistério Superior**, em 13/12/2024, às 16:26, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Bruno Oliveira Silvestre**, **Professor do Magistério Superior**, em 13/12/2024, às 16:27, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5024436** e o código CRC **154BDE12**.

Referência: Processo nº 23070.062620/2024-90 SEI nº 5024436

Costa, Felipe Aguiar. **Formalização de um Cálculo de Replicação de Nós com Extração da Máquina Abstrata para uma Avaliação Preguiçosa**. Goiânia, 2024. 46p. Trabalho de Conclusão de Curso. Insituto de Informática, Universidade Federal de Goiás.

Resumo

COSTA, Felipe Aguiar. Formalização de um cálculo de replicação de nós com extração da máquina abstrata para uma avaliação preguiçosa. 2024. 46 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Instituto de Informática, Goiânia, 2024.

A replicação nó-a-nó está relacionada às computações ótimas para reduções baseadas em grafos do λ-calculus e recentemente a substituição associada foi identificada como a interpretação Curry-Howard da Lógica com Inferência Profunda. A investigação para o cálculo com replicação de nós estabelece propriedades baseadas em sua semântica operacional, definida por um sistema de regras de redução baseadas em contextos, e em sua semântica denotacional, com um sistema de tipos quantitativo. Uma estratégia de avaliação (fraca) preguiçosa é definida para o cálculo, satisfazendo as propriedades de *full-laziness* e com uma relação de equivalência observacional equivalente a relação para a estratégia de avaliação (fraca) de chamada-por-nome. Neste trabalho é apresentada uma especificação no assistente de provas Coq de um cálculo com replicação de nós com uma estratégia de chamada-por-necessidade, com uma abordagem baseada em *refocusing*, permitindo que a máquina abstrata associada seja extraída automaticamente a partir da especificação.

Palavras–chave

máquina abstrata, replicação nó-a-nó, avaliação preguiçosa, *refocusing*.

Abstract

COSTA, Felipe Aguiar. Formalisation of a node replication calculus with abstract machine extration for a lazy evaluation. 2024. 46 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Instituto de Informática, Goiânia, 2024.

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 is 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, following an approach based on a refocusing procedure, allowing the automatic extraction of the corresponding abstract machine.

Keywords

node-replication, full-laziness, refocusing, abstract machine.

Sumário

CAPÍTULO 1

Introduction

The node-replication calculus in [Kesner, Peyrot e Ventura 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 by its abstract machine [Diehl, Hartel e Sestoft 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 e Miller 1992]. They are abstract because they ignore many aspects of physical machines, providing an intermediate level towards the language implementation. For instance, Haskell is a well-known functional language with its abstract machine implementing a lazy strategy [Jones 1992]. Refocusing was specified in [Biernacka, Charatonik e Zielinska 2017], allowing the extraction of the abstract machine from the calculus specification, including its reduction strategy. We present a specification of a node-replication calculus based on [Kesner, Peyrot e Ventura 2024] in the Coq Proof Assistant[1]. We follow the approach in [Biernacka e Charatonik 2019], using the refocusing procedure, to extract the corresponding abstract machine.

Node Replication Node-by-node replication is the Curry-Howard interpretation of deep inference [Gundersen, Heijltjes e Parigot 2013, Kesner, Peyrot e Ventura 2024], where substitutions are executed constructor by constructor. For instance, $(x\,x)[x\backslash(u\,r)]$ reduces in one step to $((y\,z)\,(y\,z))[y\backslash u][z\backslash r]$, where only the term application constructor replaces (all) free occurrences of x while both terms in the original application are kept shared. 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, Heijltjes e Parigot 2013]. The atomic λ-calculus is an explicit resource calculus where, besides explicit substitution, both weakening –corresponding to garbage collection– and contraction –dealing with

[1] https://coq.inria.fr/

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, Peyrot e Ventura 2024] a **node replication 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. **Observational equivalence** for the two strategies –where two terms are observational equivalent when they have the same behaviour in any context, *i.e.* they either compute to the same normal-form or they both diverge– were proved to be equivalent. **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 the free occurrences of x in t. Skeleton extraction was already used for full laziness in [Ariola e Felleisen 1997] but extraction was defined as a meta-operation while in [Kesner, Peyrot e Ventura 2024] the operation was defined as a substrategy in the calculus.

Refocusing Introduced as a general approach to extract abstract machines from context-based reduction semantics [Danvy e Nielsen 2004], refocusing was specified with Coq in [Sieczkowski, Biernacka e Biernacki 2010], with an automated machine extraction from a reduction semantics satisfying some syntactical properties. 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 in order to identify the next redex, with a continuation of the decomposition/recomposition procedure from its contractum, avoiding a rework while identifying the next redex. However, the so-called hybrid strategies, such as the normal order evaluation [Barendregt 1985], could not be handled by the approach in [Danvy e Nielsen 2004, Sieczkowski, Biernacka e Biernacki 2010]. Thus, a generalisation of the refocusing procedure was presented in [Biernacka, Charatonik e Zielinska 2017]. The generalised procedure was applied in [Biernacka e 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 the specification approach in [Biernacka e Charatonik 2019].

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, Peyrot e Ventura 2024] and the one introduced in the present work has not yet been investigated.

The document has the following structure: Chapter 2 presents some general background regarding context-based calculi, Chapter 3 presents the call-by-need strategy for the node-replication calculus and related notions linked to the present work, Chapter 4 presents refocusing as approached in [Sieczkowski, Biernacka e Biernacki 2010, Biernacka, Charatonik e Zielinska 2017], Chapter 5 introduces our node-replication calculus based in [Kesner, Peyrot e Ventura 2024], in Chapter 6 key aspects of the development in Coq are discussed and finally in Chapter 7 a discussion of future and related work concludes the current 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` and the Coq code — proofs omitted — is included in Appendix A.

CAPÍTULO 2

Background

We present some notions about the λ-calculus –the theoretical foundation for the functional programming paradigm [Barendregt 1985]– relevant for the present work.

The **syntax of λ-terms** is defined by: $t, u, r ::= x \mid \lambda x.t \mid t\,u$; where $x \in \mathcal{X}$ –$\mathcal{X}$ an infinite set of variables–, **abstractions** are terms of the form $\lambda x.t$ –which can be seen as functions $t(x)$– and $t\,u$ are called **applications** where u is the application **argument**. Applications are defined to be left associative *i.e.* $t\,u\,r = (t\,u)\,r$. The **set of free variable** of a term t, denoted by $\mathsf{fv}(t)$, is inductively defined by: $\mathsf{fv}(x) = \{x\}$; $\mathsf{fv}(t\,u) = \mathsf{fv}(t) \cup \mathsf{fv}(u)$; and $\mathsf{fv}(\lambda x.t) = \mathsf{fv}(t) \setminus \{x\}$[1]. Computations are executed by the so-called **β-reduction rule**: $(\lambda x.t)u \mapsto_\beta t\{x \backslash u\}$, where $t\{x \backslash u\}$ denotes the result of replacing each free occurrence of x in t by u. For instance, $(\lambda x.\lambda y.x(xy))tu \mapsto_\beta (\lambda y.t(ty))u \mapsto_\beta t(tu)$. A term of the form $(\lambda x.t)u$ is called a (β-)**redex** and $t\{x \backslash u\}$ its (β-)**contractum**. A (β-)**normal-form** is a term with no further reductions.

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, based on the λ-calculus, need to define a (deterministic) reduction strategy, executed by its abstract machine. **Context-based reductions** is one way to define a strategy. 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 the context $\mathtt{C}$ with the term t plugged into its hole. **Elementary contexts** are contexts where the context-variable corresponds to $\Diamond$. For instance, 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 for $\mathtt{C}$-contexts, *i.e.* a term t is

[1] a binder λx links all free-occurrences of x in t in the abstraction $\lambda x.t$

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

[3] some definitions do not consider variables as values

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 the $\mathtt{D}$-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 is allowed under λ-abstractions. The **call-by-need** strategy [Ariola et al. 1995, Ariola e Felleisen 1997] is a lazy strategy that combines the advantages of call-by-name and call-by value. Some **memoisation technique** are applied in this strategy, in order to avoid recalculations of partial results used more than once during computations. We use explicit cuts as a memory device in the current work and we now introduce the notion of explicit substitutions.

Explicit Substitutions Explicit substitutions calculi [Kesner 2009] include the substitution operation in the calculus. In other words, instead of substitution as a meta-operation, where $t\{x\backslash u\}$ denotes only the substitution result defined in the meta-level, a **calculus with explicit substitutions** extends the syntax with terms of the form $t[x\backslash u]$, which can be seen as a term t with a pending substitution. Terms with no explicit substitutions are called **pure terms**. Another way to interpret $t[x\backslash u]$ is as a **sharing device**, where u or any computation from it is shared by all free occurrences of x in t. The reduction rule $(\lambda x.t)u \mapsto_{\mathtt{dB}} t[x\backslash u]$ starts the substitution process and a **substitution calculus**, executing the pending substitution, must be defined. For instance, the λx-calculus [Lins 1986, Bloo e Rose 1996] has the following substitution calculus:

$$\begin{array}{lll}
x[x\backslash t] & \mapsto_{\mathtt{Var}} & t \\
y[x\backslash t] & \mapsto_{\mathtt{GC}} & y,\ x \neq y \\
(u\,r)[x\backslash t] & \mapsto_{\mathtt{App}} & u[x\backslash t]\,r[x\backslash t] \\
(\lambda y.u)[x\backslash t] & \mapsto_{\mathtt{Abs}} & \lambda y.u[x\backslash t]
\end{array}$$

where the Barendregt's Variable Convention (BVC) [Barendregt 1985], when bound and free variables have different names, is assumed in rule $\mapsto_{\mathtt{Abs}}$. Note that substitutions are propagated through the structure of a term until a variable is reached, where it is either replaced by the substitution term or is garbage collected. There are different approaches in defining substitution calculi, which by the Curry-Howard isomorphism [Sørensen e Urzyczyn 2006] correspond to different proof normalisation procedures in different logical systems. Explicit substitutions are interpreted as (explicit) **cuts** when considering such a relation and the substitution process as a **cut elimination** procedure. For instance, a typing rule for terms with explicit substitutions can be given by

$$\frac{\Gamma \vdash u : \tau \qquad \Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash t[x\backslash u] : \sigma}$$

in which erasing term annotations from the typing derivation, where variable assignments such as $x : \tau$ are considered as labelled assumptions, holds

$$\frac{\Gamma \vdash \tau \qquad \Gamma, x : \tau \vdash \sigma}{\Gamma \vdash \sigma}$$

which resembles a cut rule in sequent calculi [Troelstra e Schwichtenberg 2000]. **Full substitution** calculi, where all free occurrences of a variable are replaced, as the $\mathtt{x}$-calculus defined above, relates with proof normalisation in natural deduction [Troelstra e Schwichtenberg 2000] while a **linear substitution calculus**, where each free occurrence of a variable is individually replace at once in a substitution step, is related with cut elimination in Proof-Nets [Accattoli 2018]. For instance, $(x\,x)[x \backslash t]$ reduces to either $(t\,x)[x \backslash t]$ or $(x\,t)[x \backslash t]$ in one reduction step in the linear substitution calculus.

CAPÍTULO 3

The Node Replication Calculus

For the sake of a self-contained presentation, we present in this chapter the main concepts from [Kesner, Peyrot e Ventura 2024] linked with the current work.

Definition 3.1 (General Syntax) *Given a countably infinite set of variables $x, y, z, ...$, we consider the following grammars.*

(Terms)	t, u, r, s	$::=$	$x \mid \lambda x.t \mid tu \mid t[x\backslash u] \mid t[x\backslash\backslash\lambda y.u]$
(Pure Terms)	p, q	$::=$	$x \mid \lambda x.p \mid pq$
(Term Contexts)	$\mathtt{C}$	$::=$	$\Diamond \mid \lambda x.\mathtt{C} \mid \mathtt{C}t \mid t\mathtt{C} \mid \mathtt{C}[x\backslash t] \mid \mathtt{C}[x\backslash\backslash\lambda y.u] \mid t[x\backslash\mathtt{C}] \mid t[x\backslash\backslash\lambda y.\mathtt{C}]$
(List Contexts)	$\mathtt{L}$	$::=$	$\Diamond \mid \mathtt{L}[x\backslash u] \mid \mathtt{L}[x\backslash\backslash\lambda y.u]$

In addition to explicit substitutions, the syntax of terms in the node replication calculus is extended by **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. As discussed in Chap. 1, full laziness is achieved by skeleton extraction. We use an example to illustrate the issue.

Example 3.2 *Let $t = (\lambda x.x\,x)(\lambda z.z(\mathtt{I}\,\mathtt{I})) \mapsto_{\mathtt{dB}} (x\,x)[x\backslash(\lambda z.z(\mathtt{I}\,\mathtt{I}))]$, where $\mathtt{I} = \lambda x.x$. The term $u = (\lambda z.z(\mathtt{I}\,\mathtt{I}))$ 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. **Relation** $\Downarrow^{\theta}$, a big-step semantics for splitting θ-skeleton and MFEs with θ a set of variables, is presented in Fig. 3.1.

Correctness of $\Downarrow^{\theta}$ is proved in [Kesner, Peyrot e Ventura 2024] for pure terms. Therefore, $\lambda y.t \Downarrow^{\emptyset} \mathtt{L}\langle\lambda y.t'\rangle$, or equivalently $t \Downarrow^{\{y\}} \mathtt{L}\langle t'\rangle$, corresponds to the splitting of $\lambda y.t$ in its skeleton $\lambda y.t'$ and a list context $\mathtt{L}$ with its MFE's.

The lazy strategy considers restricted terms defined by the grammars below.

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

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

Figura 3.1: Relation $\Downarrow^{\theta}$

Definition 3.3 (Restricted Terms Syntax) *Let $|p|_x$ be the number o free occurrences of x in p, where $|\mathtt{LL}|_x$ is the straightforward extension to (commutative) list contexts:*

$$\begin{array}{llll}
\textbf{\textit{(Linear Cut Values)}} & T & ::= & \lambda x.\mathtt{LL}\langle p \rangle \textit{ where } y \in \mathsf{dom}(\mathtt{LL}) \implies |p|_y = 1 \\
\textbf{\textit{(Commutative Lists)}} & \mathtt{LL} & ::= & \Diamond \mid \mathtt{LL}[x\backslash p] \mid \mathtt{LL}[x\backslash\backslash T] \textit{ where } |\mathtt{LL}|_x = 0 \\
\textbf{\textit{(Values)}} & v & ::= & \lambda x.p \\
\textbf{\textit{(Restricted Terms)}} & U & ::= & x \mid v \mid UU \mid U[x\backslash U] \mid U[x\backslash\backslash T]
\end{array}$$

The **call-by-need strategy** $\to_{\mathtt{flneed}}$ is then defined by the closure of rules in Fig.3.2 under the **need contexts**, on the set of terms $\mathtt{U}$, given by the following grammar:

$$\textbf{(Need Contexts)} \quad \mathtt{N} ::= \Diamond \mid \mathtt{N}t \mid \mathtt{N}[x \lhd t] \mid \mathtt{N}\langle\langle x \rangle\rangle[x\backslash \mathtt{N}]$$

where $\mathtt{N}\langle\langle \_ \rangle\rangle$ denotes capture-free application of contexts. $\to_{\mathtt{flneed}}$ is a weak strategy, since no reduction steps are considered under abstractions.

$$\begin{array}{lll}
\mathtt{L}\langle \lambda x.p \rangle u & \mapsto_{\mathtt{dB}} & \mathtt{L}\langle p[x\backslash u] \rangle \\
\mathtt{N}\langle\langle x \rangle\rangle[x\backslash \mathtt{L}\langle \lambda y.p \rangle] & \mapsto_{\mathtt{spl}} & \mathtt{L}\langle \mathtt{L}'\langle \mathtt{N}\langle\langle x \rangle\rangle[x\backslash\backslash \lambda y.p'] \rangle \rangle \quad \text{if } p \Downarrow^{\{y\}} \mathtt{L}'\langle p' \rangle \\
\mathtt{N}\langle\langle x \rangle\rangle[x\backslash\backslash v] & \mapsto_{\mathtt{sub}} & \mathtt{N}\langle\langle v \rangle\rangle[x\backslash\backslash v]
\end{array}$$

Figura 3.2: Call-by-Need Strategy

Strategy $\to_{\mathtt{flneed}}$ is proved to be **deterministic** [Kesner, Peyrot e Ventura 2024], where a characterisation of $\mathtt{flneed}$-nfs is also presented. Some of these properties depend on defining a small-step semantics for skeleton extraction, presented in Fig. 3.3.

Correctness of $\to_{\mathtt{st}}$ for pure terms is proved in [Kesner, Peyrot e Ventura 2024], where $\lambda y.z[z\backslash p] \Downarrow_{\mathtt{st}} \lambda y.\mathtt{LL}\langle p' \rangle$, denoting the $\mathtt{st}$-normal form obtained from $\lambda y.z[z\backslash p]$, corresponds to the splitting of $\lambda y.p$ in skeleton $\lambda y.p'$ and the commutative list $\mathtt{LL}$ with its MFE's. Rule $\mapsto_{\mathtt{spl}}$ in Fig. 3.2 is then replace by

$$\mathtt{N}\langle\langle x \rangle\rangle[x\backslash \mathtt{L}\langle \lambda y.p \rangle] \mapsto_{\mathtt{spl}} \mathtt{L}\langle \mathtt{LL}\langle \mathtt{N}\langle\langle x \rangle\rangle[x\backslash\backslash \lambda y.p'] \rangle \rangle \text{ if } \lambda y.z[z\backslash p] \Downarrow_{\mathtt{st}} \lambda y.\mathtt{LL}\langle p' \rangle$$

$$\frac{}{t[x\backslash y] \mapsto^{y}_{\mathtt{var}} t\{x\backslash y\}} \qquad \frac{y \in \mathsf{fv}(p_1 p_2)}{t[x\backslash p_1 p_2] \mapsto^{y}_{\mathtt{app}} t\{x\backslash x_1 x_2\}[x_1\backslash p_1][x_2\backslash p_2]}$$

$$\frac{y \in \mathsf{fv}(\lambda z.p)}{t[x\backslash\lambda z.p] \mapsto^{y}_{\mathtt{dist}} t[x\backslash\backslash\lambda z.w[w\backslash p]]} \qquad \frac{y \in \mathsf{fv}(\lambda z.\mathrm{LL}\langle p\rangle) \qquad z \notin \mathsf{fv}(\mathrm{LL})}{t[x\backslash\backslash\lambda z.\mathrm{LL}\langle p\rangle] \mapsto^{y}_{\mathtt{abs}} \mathrm{LL}\langle t\{x\backslash\lambda z.p\}\rangle}$$

$$\frac{t \mapsto^{y} t' \qquad y \in \mathsf{fv}(t) \qquad y \notin \mathsf{fv}(\mathrm{LL})}{\lambda y.\mathrm{LL}\langle t\rangle \to_{\mathtt{st}} \lambda y.\mathrm{LL}\langle t'\rangle}\ (\mathrm{CTX1})$$

$$\frac{t \to_{\mathtt{st}} t' \qquad y \in \mathsf{fv}(t) \qquad y \notin \mathsf{fv}(\mathrm{LL})}{\lambda y.\mathrm{LL}\langle u[x\backslash\backslash t]\rangle \to_{\mathtt{st}} \lambda y.\mathrm{LL}\langle u[x\backslash\backslash t']\rangle}\ (\mathrm{CTX2})$$

Figura 3.3: Relation $\to_{\mathtt{st}}$

Finally, we introduce some notions used in the characterisation of `flneed`-nfs. First, **needed free variables** are defined by:

$$\mathsf{ndv}(x) := \{x\} \qquad \mathsf{ndv}(t[y\backslash u]) := \begin{cases} (\mathsf{ndv}(t) \setminus \{y\}) \cup \mathsf{ndv}(u) & \text{if } y \in \mathsf{ndv}(t) \\ \mathsf{ndv}(t) & \text{if } y \notin \mathsf{ndv}(t) \end{cases}$$

$$\mathsf{ndv}(tu) := \mathsf{ndv}(t) \qquad \mathsf{ndv}(t[x\backslash\backslash u]) := \mathsf{ndv}(t)$$

$$\mathsf{ndv}(\lambda x.t) := \emptyset$$

Lemma 3.4 ([Kesner, Peyrot e Ventura 2024]) *Let $t \in \mathtt{U}$. Then $x \in \mathsf{ndv}(t)$ iff there exists a context $\mathtt{N}$ such that $t = \mathtt{N}\langle\langle x\rangle\rangle$.*

Second, terms of `U` in `flneed`-nf are proved to be be characterized by the grammar `Ne`, with $\overline{\mathtt{Ne}}$ the grammar of neutral terms, in [Kesner, Peyrot e Ventura 2024]:

Definition 3.5 (`flneed`-Normal Forms)

$$Ne ::= \mathrm{L}\langle\lambda x.t\rangle \mid \overline{Ne}$$

$$\overline{Ne}, \overline{Ne_0} ::= x \mid \overline{Ne}\,t \mid \overline{Ne}[x \triangleleft u]\ x \notin \mathsf{ndv}(\overline{Ne}) \mid \overline{Ne}[x\backslash\overline{Ne_0}]\ x \in \mathsf{ndv}(\overline{Ne})$$

CAPÍTULO 4

Abstract Machine Extraction

Evaluation functions or **evaluators** are implemented as the transitive closure of a reduction (operational) semantics and for a context-based reduction semantics each step consists in three stages: (1) decomposition; (2) contraction whenever possible; and (3) recomposition. For instance, let $t = (\lambda x.\lambda y.y\,x)u\,v\,w\,s$ and the weak call-by-name strategy with the D-contexts presented in Chap. 2. 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, and the result used in the recomposition. 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.

Refocusing was introduced in [Danvy e Nielsen 2004] as a general approach to extract abstract machines from context-based reduction semantics, where information obtained in the decomposition stage in one step is used in the following reduction step, in order to avoid rework in the decomposition/recomposition process. For instance, decomposition of t' defined above 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'.

In this way, the first two iterations while decomposing both t and t', holding the exact same contexts, are executed only once. A **refocus function** mapping a contractum and a context to the new (reduction) context and a redex, then replaces the (3)-

(1) recomposition/decomposition transition. Three syntactic properties are identified in [Danvy e Nielsen 2004] to be sufficient for deriving a refocus function from an evaluator.

The refocusing technique was formalised in the Coq Proof Assistant [Sieczkowski, Biernacka e Biernacki 2010], where an axiomatisation of a reduction semantics was proposed and proved sufficient to automate 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 the decomposition of any term t:

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 with term t 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.

Beside substitution based calculi approached in [Danvy e Nielsen 2004], the technique in [Sieczkowski, Biernacka e Biernacki 2010] is also applied to explicit substitution calculi and other extensions. However, the so-called hybrid strategies cannot be handled by the approach. For example, given $v ::= n \mid \lambda x.v$, the β-nfs, with $n ::= x \mid n\,v$, the (strong) neutral normal-forms, the normal order can be defined by the closure of $\mapsto_\beta$ by $\mathtt{E}$-contexts defined as follows: $\mathtt{E} ::= \Diamond_{\mathtt{E}} \mid \lambda x.\mathtt{E} \mid \mathtt{D}\,t \mid n\mathtt{E}$ $\mathtt{D} ::= \Diamond_{\mathtt{D}} \mid \mathtt{D}\,t \mid n\mathtt{E}$
The strategy, defined by the $\mathtt{E}$-contexts, has a substrategy, defined by the $\mathtt{D}$-contexts. More generally, hybrid strategies may have several substrategies, each one defined by a distinct context variable with a corresponding (kinded) context hole. On the other hand, uniform strategies such as the weak call-by-name uses only one context variable while defining its reduction contexts, since the same strategy is uniformly considered along

the evaluation process. A generalisation of the refocusing procedure was presented in [Biernacka, Charatonik e Zielinska 2017] to handle hybrid strategies, where the conditions on grammars for the generalised procedure are specified, called normal grammars (Definition 1), with further restrictions to guarantee uniqueness of decompositions (Definition 3). Among the adaptations to generalise the procedure are the parametrisation of contexts and values by kinds and strict orders for each kind. The decomposition and recomposition of values functions are called $\Downarrow$ and $\Uparrow$, respectively.

The generalised procedure was applied in [Biernacka e Charatonik 2019] for both weak and strong call-by-need calculi, the former a uniform strategy while the latter is hybrid, and the current development follows their specification approach.

CAPÍTULO 5

Towards a Lazy Node-Replication Machine

The weak call-by-need calculus in [Biernacka e Charatonik 2019] uses two explicit substitution constructors to differentiate when the substitution term needs to be evaluated. Such construct is called an **active** or a **strict substitution**, denoted by $t[\![x \backslash u]\!]$. The original paper on strong call-by-need [Balabonski et al. 2017] has no such a constructor, but the strong strategy defined in [Biernacka e Charatonik 2019] uses the same approach. The present specification for a node replication calculus based in the $\rightarrow_{\mathtt{flneed}}$ strategy also extends the calculus with active/strict cuts.

The **syntax of term expressions** from Def. 3.1 is extended to include **active cuts** $t[\![x \triangleleft u]\!]$:

$$
\begin{array}{llll}
\textbf{(Terms)} & t, u, r, s & ::= & x \mid \lambda x.t \mid tu \mid t[x \backslash u] \mid t[x \backslash\backslash \lambda y.u] \mid t[\![x \backslash u]\!] \mid t[\![x \backslash\backslash \lambda y.u]\!] \\
\textbf{(Term Values)} & v & ::= & \lambda x.t \\
\textbf{(List Contexts)} & \mathtt{L} & ::= & \Diamond \mid \mathtt{L}[x \backslash u] \mid \mathtt{L}[x \backslash\backslash \lambda y.u]
\end{array}
$$

with the corresponding reduction contexts:

$$
\textbf{(Needy Contexts)} \quad \mathtt{N} ::= \Diamond \mid \mathtt{N}t \mid \mathtt{N}[x \triangleleft t] \mid \mathtt{N}\langle\langle x \rangle\rangle[\![x \backslash \mathtt{N}]\!]
$$

Elementary contexts are needy contexts where $\mathtt{N} = \Diamond$. **Reduction rules** for the extended syntax is presented below, with relation $\Downarrow^{\theta}$ as defined in 3.1.

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

The $\rightarrow_{\mathtt{flneed}}$ strategy is considered on $\mathtt{U}$ terms while here the reduction strategy is considered for any term originated along a reduction starting from a pure term. Normal-

forms are expected to be one of two kinds, as the $\to_{\texttt{flneed}}$-nfs (Def. 3.5):

$$\begin{array}{llll} \textbf{(Needy Terms)} & n^x & ::= & x \mid n^x\, t \mid n^x[y \triangleleft t] \mid n^y[\![y \backslash n^x]\!] \\ \textbf{(Answers)} & a & ::= & v \mid a[x \triangleleft t] \end{array}$$

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. $a = \mathtt{L}\langle v \rangle$. When restricted to terms in $\mathtt{U}$, needy terms and answers coincides with terms in $\mathtt{Ne}$. Following the approach established in [Biernacka, Charatonik e Zielinska 2017], 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**, as in both weak and strong call-by-need strategies specified in [Biernacka e Charatonik 2019].

On the other hand, differently from [Biernacka e Charatonik 2019] where needy terms play the role of intermediate results only, the current weak strategy also considers them as final results since open terms, *i.e.* terms with free variables, are allowed.

As mentioned in Chap. 4, refocusing is based on keeping a stack of elementary contexts to avoid a rework while identifying the next redex. We now define the two functions used as inputs in the refocusing procedure.

Definition 5.1 (⇓/⇑ Functions) *Down (⇓) and Up (⇑) functions –where* $\mathtt{V}_{\_}$ *and* $\mathtt{R}$ *indicates a value and a redex, respectively– are defined by:*

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

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 achieved, the latter triggering the application of ⇑, recomposing values in order to identify the next redex. We present an example to illustrate the procedure.

Example 5.2 (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 elementary contexts.* ⇑ *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.*

CAPÍTULO 6

Coq Formal Development

The formalization for the node replication calculus presented in Chap. 5 and extraction of the abstract machine in Coq is based on the weak call-by-need example specified in [Biernacka e Charatonik 2019], which can be found at https://bitbucket.org/pl-uwr/generalized_refocusing. This original codebase provides a framework for specifying and extracting the abstract machine utilizing the generalized refocusing procedure. It was implemented using Coq version 8.11.1, which we also use in our work. Our full development can be found at https://github.com/felipeagc/generalized_refocusing/blob/master/examples/node_replication.v. Coq code with proofs omitted is included in App. A.

As stated in [Biernacka, Charatonik e Zielinska 2017], the implementation is split into two modules, one that implements the reduction semantics (satisfying the PRE_REF_SEM module signature), and one that implements the lower-level reduction strategy (satisfying the signature returned by REF_STRATEGY).

Starting with the module that defines the reduction strategy –which we call Lam_cbnd_PreRefSem–, the 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:

```
Inductive ck := N.
```

Among the adaptations necessary to implement the node replication strategy, we extended the expr definition to include explicit distributors besides explicit substitution, along with their strict variants. The same was done to the needy type definition, where a needy term n^x has type needy x.

```
Inductive id :=
  | Id : nat -> id.

Definition var := id.

Inductive expr :=
  | Var       : var -> expr
  | Lam       : var -> expr -> expr
```

```
  | App       : expr -> expr -> expr
  | ExpSubst  : expr -> var -> expr -> expr
  | ExpSubstS : forall x : var, needy x -> expr -> expr
  | ExpDist   : expr -> var -> var -> expr -> expr
  | ExpDistS  : forall x : var, needy x -> var -> expr -> expr
with
  needy : var -> Type :=
  | nVar       : forall x : var, needy x
  | nApp       : forall x : var, needy x -> expr -> needy x
  | nExpSubst  : forall x y,
    x <> y -> needy x -> expr -> needy x
  | nExpSubstS : forall x y,
    needy y -> needy x -> needy x
  | nExpDist   : forall x y z : var,
    x <> y -> needy x -> expr -> 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`[1] (dependent) type is one of the must-do definitions, which in our calculus is based on term values with type `val N`, answers as defined in Chap. 5 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.
Hint Unfold value.
```

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 (see App. A). 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

[1] `value` cannot be defined as an inductive type due to a limitations of the machine-extraction formalisation presented in [Biernacka, Charatonik e Zielinska 2017] thus the use of the `answer` inductive type.

difference is that `ansCtx` type is also parametrised by context kinds while the type `sub` of lists of cuts is not. Since this change only amounts to the removal of the context kind parameter from `ansCtx`, and the original code only had one context kind, no other major changes related to this were necessary. Redices are defined as in Chap. 5:

```
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 elementary contexts `eck` was also adapted to match the definition of elementary contexts in Chap. 5:

```
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.
```

The key departure from the approach in [Biernacka e Charatonik 2019] is the implementation of the `skl_extract` function, which encodes the skeleton extraction process defined in Fig. 3.1. To implement this step, we needed additional auxiliary functions, 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 Chap. 5, 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 the previous module –`Lam_cbnd_PreRefSem`– passed as a parameter. $\Downarrow$ and $\Uparrow$ functions are implemented as `dec_term` and `dec_context`, respectively;

```
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.

Definition dec_context {k k': ckind} (ec: elem_context_kinded k k')
    (v: value k') : elem_dec k :=
  match k, k' with N, N =>
    match ec, v with
    | eckApp t, ansVal v' s => ed_red (rApp v' s t)
    | eckApp t, ansNd _ n => ed_val (ansNd _ (nApp _ n t))
    | eckSubst x t, ansVal v' s => ed_val (ansVal v' (subSubst s x t))
    | eckDist x y u, ansVal v' s => ed_val (ansVal v' (subDist s x y u))
    | eckSubst x t, ansNd y n =>
      (match eq_var y x with
      | left peq => ed_red (rSpl y n t) (* redex! *)
      | right pneq => ed_val (ansNd y (nExpSubst y x pneq n t))
      end)
    | eckDist x z u, ansNd y n =>
      (match eq_var y x with
      | left peq => ed_red (rLs y n (vLam z u)) (* redex! *)
      | right pneq => ed_val (ansNd y (nExpDist y x z pneq n u))
      end)
    | eckPlugSubst x n, ansVal v s =>
      ed_red (rSplS _ n v s)
    | eckPlugSubst x n, ansNd _ n' =>
      ed_val (ansNd _ (nExpSubstS _ x n n'))
    end
  end.
```

The automated abstract machine extraction is based on calculi that satisfy some properties, that need to be verified in the specification above. For instance, functions converting term values, needy terms, answers and redices to terms are necessary in this approach and they

all need to be proved injective. For instance, given `redex_to_term` converting a term of type `redex k` to a term of type `expr`:

```
Lemma redex_to_term_injective : forall {k} (r r' : redex k),
                        redex_to_term r = redex_to_term r' -> r = r'.
```

Also, 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`), all necessary to be proved in order to be able to execute the abstract machine extraction procedure.

With both modules (`Lam_cbnd_PreRefSem` and `Lam_cbn_Strategy`) 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. Then, the functor `RefEvalApplyMachine` is applied to this module, which produces the abstract machine. The result of the abstract machine extraction process can be found on App. B.

CAPÍTULO 7

Conclusion

A node-replication calculus based in [Kesner, Peyrot e Ventura 2024] was specified in the Coq proof assistant, following the refocusing procedure in [Biernacka, Charatonik e Zielinska 2017], allowing the extraction of the corresponding abstract machine. Our formalisation was implemented using Coq version 8.11.1, and it is based on the framework presented in [Biernacka e Charatonik 2019], which can be found at `https://bitbucket.org/pl-uwr/generalized_refocusing`. 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 Chapter 5.

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, Peyrot e Ventura 2024] and the one defined in Chap. 5. One property to be checked is if all terms along reductions originated from pure terms in our calculus are `U` terms as in Definition 3.3. In other words, if our calculus is closed to `U` terms, even without syntax restrictions in the reduction rules definitions. 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 as a hybrid strategy, with skeleton extraction as a substrategy. Finally, in [Biernacka, Charatonik e Drab 2022] a different approach to extract an abstract machine from a higher-order evaluator using the memothunks technique to implement laziness obtains an efficient implementation for the strong call-by-need strategy. It remains to be seen whether the technique may be applied to the node-by-node replication calculi.

Referências Bibliográficas

[Accattoli 2018]ACCATTOLI, B. Proof nets and the linear substitution calculus. In: *ICTAC*. [S.l.]: Springer, 2018. (Lecture Notes in Computer Science, v. 11187), p. 37–61.

[Ariola e Felleisen 1997]ARIOLA, Z. M.; FELLEISEN, M. The call-by-need lambda calculus. *J. Funct. Program.*, v. 7, n. 3, p. 265–301, 1997.

[Ariola et al. 1995]ARIOLA, Z. M. et al. The call-by-need lambda calculus. In: *POPL*. [S.l.]: ACM Press, 1995. p. 233–246.

[Balabonski et al. 2017]BALABONSKI, T. et al. Foundations of strong call by need. *Proc. ACM Program. Lang.*, v. 1, n. ICFP, p. 20:1–20:29, 2017.

[Barendregt 1985]BARENDREGT, H. P. *The lambda calculus - its syntax and semantics*. [S.l.]: North-Holland, 1985. (Studies in logic and the foundations of mathematics, v. 103).

[Biernacka e Charatonik 2019]BIERNACKA, M.; CHARATONIK, W. Deriving an abstract machine for strong call by need. In: *FSCD*. [S.l.]: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. (LIPIcs, v. 131), p. 8:1–8:20.

[Biernacka, Charatonik e Drab 2022]BIERNACKA, M.; CHARATONIK, W.; DRAB, T. A simple and efficient implementation of strong call by need by an abstract machine. *Proc. ACM Program. Lang.*, v. 6, n. ICFP, p. 109–136, 2022.

[Biernacka, Charatonik e Zielinska 2017]BIERNACKA, M.; CHARATONIK, W.; ZIELINSKA, K. Generalized refocusing: From hybrid strategies to abstract machines. In: *FSCD*. [S.l.]: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. (LIPIcs, v. 84), p. 10:1–10:17.

[Bloo e Rose 1996]BLOO, R.; ROSE, K. H. Combinatory reduction systems with explicit substitution that preserve strong nomalisation. In: *RTA*. [S.l.]: Springer, 1996. (Lecture Notes in Computer Science, v. 1103), p. 169–183.

[Danvy e Nielsen 2004]DANVY, O.; NIELSEN, L. R. Refocusing in reduction semantics. *BRICS Report Series*, v. 11, n. 26, Nov. 2004. Disponível em: <https://tidsskrift.dk/brics/article/view/21851>.

[Diehl, Hartel e Sestoft 2000]DIEHL, S.; HARTEL, P. H.; SESTOFT, P. Abstract machines for programming language implementation. *Future Gener. Comput. Syst.*, v. 16, n. 7, p. 739–751, 2000.

[Gundersen, Heijltjes e Parigot 2013]GUNDERSEN, T.; HEIJLTJES, W.; PARIGOT, M. Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In: *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. [S.l.]: IEEE Computer Society, 2013. (LICS '13), p. 311–320. ISBN 978-0-7695-5020-6.

[Hannan e Miller 1992]HANNAN, J.; MILLER, D. From operational semantics for abstract machines. *Math. Struct. Comput. Sci.*, v. 2, n. 4, p. 415–459, 1992.

[Jones 1992]JONES, S. L. P. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. Funct. Program.*, v. 2, n. 2, p. 127–202, 1992.

[Kesner 2009]KESNER, D. A theory of explicit substitutions with safe and full composition. *Log. Methods Comput. Sci.*, v. 5, n. 3, 2009.

[Kesner, Peyrot e Ventura 2024]KESNER, D.; PEYROT, L.; VENTURA, D. Node replication: Theory and practice. *Log. Methods Comput. Sci.*, v. 20, n. 1, 2024.

[Lamping 1990]LAMPING, J. An algorithm for optimal lambda calculus reduction. In: *POPL*. [S.l.]: ACM Press, 1990. p. 16–30.

[Lins 1986]LINS, R. D. A new formula for the execution of categorial combinators. In: *CADE*. [S.l.]: Springer, 1986. (Lecture Notes in Computer Science, v. 230), p. 89–98.

[Sieczkowski, Biernacka e Biernacki 2010]SIECZKOWSKI, F.; BIERNACKA, M.; BIERNACKI, D. Automating derivations of abstract machines from reduction semantics: - A generic formalization of refocusing in coq. In: *IFL*. [S.l.]: Springer, 2010. (Lecture Notes in Computer Science, v. 6647), p. 72–88.

[Sørensen e Urzyczyn 2006]SøRENSEN, M. H.; URZYCZYN, P. *Lectures on the Curry-Howard Isomorphism*. [S.l.]: Elsevier, 2006.

[Troelstra e Schwichtenberg 2000]TROELSTRA, A. S.; SCHWICHTENBERG, H. *Basic proof theory, Second Edition*. [S.l.]: Cambridge University Press, 2000. (Cambridge tracts in theoretical computer science, v. 43).

APÊNDICE A

Coq Code

The following is the Coq code excluding proofs.

```
Require Import Program.
Require Import Util.
Require Import refocusing_semantics.
Require Import empty_search_order.
Require Import MSets.

Require Import ListSet.
Require Import Sets.

Module Lam_cbnd_PreRefSem <: PRE_RED_SEM.
  (* We define variables as numbered identifiers. *)
  Inductive id :=
  | Id : nat -> id.

  Definition var := id.

  Theorem eq_var : forall x y : var, {x = y} + {x <> y}.

  Inductive ck := N.

  Definition ckind := ck.
  Hint Unfold  ckind.

  Inductive expr :=
  | Var       : var -> expr
  | Lam       : var -> expr -> expr
  | App       : expr -> expr -> expr
  | ExpSubst  : expr -> var -> expr -> expr
  | ExpSubstS : forall x : var, needy x -> expr -> expr
  | ExpDist   : expr -> var -> var -> expr -> expr
  | ExpDistS  : forall x : var, needy x -> var -> expr -> expr
  with
    needy : var -> Type :=
```

```
  | nVar       : forall x : var, needy x
  | nApp       : forall x : var, needy x -> expr -> needy x
  | nExpSubst  : forall x y,
    x <> y -> needy x -> expr -> needy x
  | nExpSubstS : forall x y,
    needy y -> needy x -> needy x
  | nExpDist   : forall x y z : var,
    x <> y -> needy x -> expr -> needy x.

Notation " t @ s " := (App t s) (at level 40).
Notation " # x " := (Var x) (at level 7).
Notation " t [ x \ u ] " := (ExpSubst t x u) (at level 45).
Notation " t [ x '\\' 'λ' y , u ] " := (ExpDist t x y u) (at level 46).
Notation " 'λ'  x , t " := (Lam x t) (at level 50).

Definition term := expr.
Hint Unfold term.

Fixpoint fresh_ind_n {x} (nx : needy x) : nat :=
  match nx with
  | nVar (Id x) => x
  | nApp _ nx t => max (fresh_ind_n nx) (fresh_ind t)
  | nExpSubst (Id x) (Id y) _ nx u =>
      max (max (max (fresh_ind_n nx) (fresh_ind u)) y) x
  | nExpSubstS (Id x) (Id y) nx ny =>
      max (max (max (fresh_ind_n nx) (fresh_ind_n ny)) y) x
  | nExpDist (Id x) (Id y) (Id z) _ nx u =>
      max (max (max (max (fresh_ind_n nx) (fresh_ind u)) z) y) x
  end
with fresh_ind (t : term) : nat :=
  match t with
  | Var (Id x) => x
  | Lam (Id x) p => max (fresh_ind p) x
  | App p q => max (fresh_ind p) (fresh_ind q)
  | ExpSubst p (Id x) u =>
      max (max (fresh_ind p) (fresh_ind u)) x
  | ExpSubstS (Id x) nx u =>
      max (max (fresh_ind u) (fresh_ind_n nx)) x
  | ExpDist p (Id x) (Id y) u =>
      max (max (fresh_ind p) (fresh_ind u)) (max x y)
  | ExpDistS (Id x) nx (Id y) u =>
      max (max (fresh_ind_n nx) (fresh_ind u)) (max x y)
  end.

Definition fresh_var t := Id (fresh_ind t).
```

```
Inductive sub :=
| subEmpty : sub
| subSubst : sub -> var -> term -> sub
| subDist  : sub -> var -> var -> term -> sub.

Fixpoint sub_to_term (s : sub) (t : term) :=
  match s with
  | subEmpty => t
  | subSubst s' x r => ExpSubst (sub_to_term s' t) x r
  | subDist s' x y r => ExpDist (sub_to_term s' t) x y r
  end.

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

Definition val_to_term {k} (v : val k) : term :=
  match v with
  | vLam x t => Lam x t
  end.

Coercion val_to_term : val >-> term.

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

Definition value := answer.
Hint Unfold value.

Fixpoint needy_to_term {x} (n : needy x) : term :=
  match n with
  | nVar x => Var x
  | nApp _ n t => App (needy_to_term n) t
  | nExpSubst _ y _ n e => ExpSubst (needy_to_term n) y e
  | nExpSubstS _ y n_y n_x => ExpSubstS y n_y (needy_to_term n_x)
  | nExpDist _ y z _ n_x e => ExpDist (needy_to_term n_x) y z e
  end.

Definition answer_to_term {k} (a : answer k) : term :=
  match a with
  | ansVal v s => sub_to_term s v
  | ansNd _ n   => needy_to_term n
  end.

Coercion answer_to_term : answer >-> term.
Coercion needy_to_term : needy >-> term.
```

```
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.
Hint Unfold redex.

Definition redex_to_term {k} (r : redex k) : term :=
  match r with
  | rApp v l t => App (sub_to_term l (val_to_term v)) t
  | rSplS x n v s => ExpSubstS x n (sub_to_term s (val_to_term v))
  | rSpl x n t => ExpSubst (needy_to_term n) x t
  | rLsS x n (vLam y t) => ExpDistS x n y t
  | rLs x n (vLam y t) => ExpDist (needy_to_term n) x y t
  end.

Coercion redex_to_term : redex >-> term.

Lemma val_to_term_injective :
  forall {k} (v v' : val k),
  val_to_term v = val_to_term v' -> v = v'.

Lemma sub_to_term_needy :
  forall {k} (s s' : sub) (v : val k) {x} (n : needy x),
  sub_to_term s v = sub_to_term s' n -> False.

Lemma needy_to_term_injective :
  forall {x y} (n : needy x) (n' : needy y),
  needy_to_term n = needy_to_term n' -> n ~= n' /\ x = y.

Lemma sub_to_term_val_injective :
  forall {k} (s s' : sub) (v v' : val k),
  sub_to_term s v = sub_to_term s' v' ->
    s = s' /\ v = v'.

Lemma sub_to_term_var_injective :
  forall (s s' : sub) x x',
  sub_to_term s (Var x) = sub_to_term s' (Var x') ->
    s = s' /\ x = x'.

Lemma answer_val_not_needy :
  forall {k x} (s: sub) (v : val k) (n : needy x),
```

```
  sub_to_term s v = needy_to_term n -> False.

Lemma answer_to_term_injective :
  forall {k} (a a' : answer k),
  answer_to_term a = answer_to_term a' -> a = a'.

Definition value_to_term {k} (a : value k) := answer_to_term a.

Lemma value_to_term_injective : forall {k} (a a' : value k),
  value_to_term a = value_to_term a' -> a = a'.

Lemma redex_to_term_injective :
  forall {k} (r r' : redex k),
  redex_to_term r = redex_to_term r' -> r = r'.

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.

Definition elem_context_kinded := eck.
Hint Unfold elem_context_kinded.

Definition init_ckind : ckind :=  N.

Definition elem_plug {k1 k2} (t : term)
  (ec : elem_context_kinded k1 k2) : term :=
  match ec with
  | eckApp t' => App t t'
  | eckSubst x s => ExpSubst t x s
  | eckDist x y u => ExpDist t x y u
  | eckPlugSubst x n => ExpSubstS x n t
  end.

Fixpoint subst_needy (x : var) (n : needy x) (s : term) : term :=
  match n with
  | nVar x' => s
  | nApp x' n t => App (subst_needy x' n s) t
  | nExpSubst x' y _ n t => ExpSubst (subst_needy x' n s) y t
  | nExpSubstS x' y ny nx => ExpSubstS y ny (subst_needy x' nx s)
  | nExpDist x' y z _ nx t => ExpDist (subst_needy x' nx s) y z t
  end.

Module S := Make Nat_as_OT.
```

```
Fixpoint fvn {x} (t : needy x) : S.t :=
  match t with
  | nVar (Id x) => S.singleton x
  | nApp _ p q => S.union (fvn p) (fv q)
  | nExpSubst (Id x) (Id y) _ nx u =>
      S.union (S.remove y (fvn nx)) (fv u)
  | nExpSubstS (Id x) (Id y) nx ny =>
      S.union (S.remove y (fvn ny)) (fvn nx)
  | nExpDist (Id x) (Id y) (Id z) _ nx u =>
      S.union (S.remove y (fvn nx)) (S.remove z (fv u))
  end
with fv (t : term) : S.t :=
  match t with
  | Var (Id x) => S.singleton x
  | Lam (Id x) p => S.remove x (fv p)
  | App p q => S.union (fv p) (fv q)
  | ExpSubst t (Id x) u =>
      S.union (S.remove x (fv t)) (fv u)
  | ExpSubstS (Id x) nx u =>
      S.union (S.remove x (fvn nx)) (fv u)
  | ExpDist t (Id x) (Id y) u =>
      S.union (S.remove x (fv t)) (S.remove y (fv u))
  | ExpDistS (Id x) nx (Id y) u =>
      S.union (S.remove x (fvn nx)) (S.remove y (fv u))
  end.

Fixpoint concat_sub (l1 : sub) (l2 : sub) : sub :=
  match l1 with
  | subEmpty => l2
  | subSubst l1 v t => subSubst (concat_sub l1 l2) v t
  | subDist l1 x y u => subDist (concat_sub l1 l2) x y u
  end.

Fixpoint skl_extract (t : term) (theta : S.t) (n : nat) :
  sub * term * nat :=
  if S.is_empty (S.inter theta (fv t)) then
    let x := Id n in
    (subSubst subEmpty x t, Var x, n+1)
  else match t with
       | Var x =>
           (subEmpty, Var x, n)
       | Lam (Id x) t =>
           let '(l, t', v) := skl_extract t (S.add x theta) n in
           (l, Lam (Id x) t', n)
       | App p q =>
           let '(l1, p', v) := skl_extract p theta n in
```

```
            let '(l2, q', v) := skl_extract q theta n in
            (concat_sub l1 l2, App p' q', n)
        | ExpSubst p (Id x) u =>
            let '(l1, p', n') := skl_extract p (S.add x theta) n in
            let '(l2, u', m) := skl_extract u theta n' in
            (concat_sub l2 l1, ExpSubst p' (Id x) u', m)
        | ExpSubstS (Id x) nx u =>
            let '(l1, u', m) := skl_extract u theta n in
            (l1, ExpSubstS (Id x) nx u', m)
        | ExpDist p (Id x) (Id y) u =>
            let '(l1, p', n') := skl_extract p (S.add x theta) n in
            let '(l2, u', m) := skl_extract u (S.add y theta) n' in
            (concat_sub l2 l1, ExpDist p' (Id x) (Id y) u', m)
        | ExpDistS (Id x) nx (Id y) u =>
            let '(l1, u', m) := skl_extract u (S.add y theta) n in
            (l1, ExpDistS (Id x) nx (Id y) u', m)
        end.

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.

Include RED_SEM_BASE_Notions.

Lemma elem_plug_injective1 :
  forall {k1 k2} (ec : elem_context_kinded k1 k2) {t0 t1},
  ec:[t0] = ec:[t1] -> t0 = t1.

Lemma wf_immediate_subterm: well_founded immediate_subterm.

Definition wf_subterm_order : well_founded subterm_order
  := wf_clos_trans_l _ _ wf_immediate_subterm.

Lemma value_trivial1 :
  forall {k1 k2} (ec: elem_context_kinded k1 k2) t,
```

```
    forall v : value k1,
    ec:[t] = v -> exists (v' : value k2), t = v'.

  Lemma value_redex :
    forall {k} (v : value k) (r : redex k),
    value_to_term v <> redex_to_term r.

  Lemma redex_trivial1 :
    forall {k k'} (r : redex k) (ec : elem_context_kinded k k') t,
    ec:[t] = r -> exists (v : value k'),
    t = v.
End Lam_cbnd_PreRefSem.

Module Lam_cbn_Strategy <: REF_STRATEGY Lam_cbnd_PreRefSem.
  Import Lam_cbnd_PreRefSem.
  Include RED_STRATEGY_STEP_Notions Lam_cbnd_PreRefSem.

  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.

  Lemma dec_term_correct : forall t k, t = elem_rec (dec_term t k).

  Definition dec_context {k k': ckind} (ec: elem_context_kinded k k')
    (v: value k') : elem_dec k :=
  match k, k' with N, N =>
    match ec, v with
    | eckApp t, ansVal v' s => ed_red (rApp v' s t)
    | eckApp t, ansNd _ n => ed_val (ansNd _ (nApp _ n t))
    | eckSubst x t, ansVal v' s => ed_val (ansVal v' (subSubst s x t))
    | eckDist x y u, ansVal v' s => ed_val (ansVal v' (subDist s x y u))
    | eckSubst x t, ansNd y n =>
      (match eq_var y x with
      | left peq => ed_red (rSpl y n t) (* redex! *)
      | right pneq => ed_val (ansNd y (nExpSubst y x pneq n t))
      end)
    | eckDist x z u, ansNd y n =>
```

```
      (match eq_var y x with
      | left peq => ed_red (rLs y n (vLam z u)) (* redex! *)
      | right pneq => ed_val (ansNd y (nExpDist y x z pneq n u))
      end)
    | eckPlugSubst x n, ansVal v s =>
      ed_red (rSplS _ n v s)
    | eckPlugSubst x n, ansNd _ n' =>
      ed_val (ansNd _ (nExpSubstS _ x n n'))
    end
  end.

  Lemma dec_context_correct :
    forall {k k'} (ec : elem_context_kinded k k') (v : value k'),
    ec:[v] = elem_rec (dec_context ec v).

  Lemma search_order_comp_if :
    forall t k k' k'' (ec0 : elem_context_kinded k k')
      (ec1 : elem_context_kinded k k''),
    immediate_ec ec0 t -> immediate_ec ec1 t -> k,
    t |~ ec0 << ec1  \/  k,
    t |~ ec1 << ec0  \/  (k' = k'' /\ ec0 ~= ec1).

  Lemma dec_context_term_next :
    forall {k0 k1 k2} (v : value k1) t
      (ec0 : elem_context_kinded k0 k1)
      (ec1 : elem_context_kinded k0 k2),
    dec_context ec0 v = ed_dec _ t ec1 -> so_predecessor ec1 ec0 ec0:[v].
End Lam_cbn_Strategy.
```

APÊNDICE B

Extracted Abstract Machine

The following is the extracted abstract machine.

```
Module
Lam_cbn_EAM
 : Sig
    Parameter value_trivial :
      forall (k : Lam_cbn_RefSem.ckind) (v : Lam_cbn_RefSem.value k)
        (k' : Lam_cbn_RefSem.ckind) (c : Lam_cbn_RefSem.context k k')
        (t : Lam_cbn_RefSem.term),
      (c) [t] = Lam_cbn_RefSem.value_to_term v ->
      exists v' : Lam_cbn_RefSem.value k',
        t = Lam_cbn_RefSem.value_to_term v'.
    Definition term : Type.
    Definition value : Type.
    Definition value_to_term : value -> term.
    Inductive conf : Type :=
        c_eval : term ->
                 forall k : Lam_cbn_RefSem.ckind,
                 Lam_cbn_RefSem.context ick k -> conf
      | c_apply : forall k : Lam_cbn_RefSem.ckind,
                  Lam_cbn_RefSem.context ick k ->
                  Lam_cbn_RefSem.value k -> conf
    Definition conf_rect :
      forall P : conf -> Type,
      (forall (t : term) (k : Lam_cbn_RefSem.ckind)
         (c : Lam_cbn_RefSem.context ick k), P (c_eval t c)) ->
      (forall (k : Lam_cbn_RefSem.ckind) (c : Lam_cbn_RefSem.context ick k)
         (v : Lam_cbn_RefSem.value k), P (c_apply c v)) ->
      forall c : conf, P c.
    Definition conf_ind :
      forall P : conf -> Prop,
      (forall (t : term) (k : Lam_cbn_RefSem.ckind)
         (c : Lam_cbn_RefSem.context ick k), P (c_eval t c)) ->
      (forall (k : Lam_cbn_RefSem.ckind) (c : Lam_cbn_RefSem.context ick k)
         (v : Lam_cbn_RefSem.value k), P (c_apply c v)) ->
```

```
  forall c : conf, P c.
Definition conf_rec :
  forall P : conf -> Set,
  (forall (t : term) (k : Lam_cbn_RefSem.ckind)
     (c : Lam_cbn_RefSem.context ick k), P (c_eval t c)) ->
  (forall (k : Lam_cbn_RefSem.ckind) (c : Lam_cbn_RefSem.context ick k)
     (v : Lam_cbn_RefSem.value k), P (c_apply c v)) ->
  forall c : conf, P c.
Definition conf_sind :
  forall P : conf -> SProp,
  (forall (t : term) (k : Lam_cbn_RefSem.ckind)
     (c : Lam_cbn_RefSem.context ick k), P (c_eval t c)) ->
  (forall (k : Lam_cbn_RefSem.ckind) (c : Lam_cbn_RefSem.context ick k)
     (v : Lam_cbn_RefSem.value k), P (c_apply c v)) ->
  forall c : conf, P c.
Definition configuration : Type.
Definition load : term -> configuration.
Definition final : configuration -> option value.
Definition decompile : configuration -> term.
Definition is_final : configuration -> Prop.
Inductive trans : configuration -> configuration -> Prop :=
    t_red : forall (t : Lam_cbn_RefSem.term)
              (k : Lam_cbn_RefSem.ckind)
              (c : Lam_cbn_RefSem.context ick k)
              (r : Lam_cbn_RefSem.redex k) (t0 : Lam_cbn_RefSem.term),
            Lam_cbn_RefSem.dec_term t k = Lam_cbn_RefSem.ed_red r ->
            Lam_cbn_RefSem.contract r = Some t0 ->
            trans (c_eval t c) (c_eval t0 c)
  | t_val : forall (t : Lam_cbn_RefSem.term)
              (k : Lam_cbn_RefSem.ckind)
              (c : Lam_cbn_RefSem.context ick k)
              (v : Lam_cbn_RefSem.value k),
            Lam_cbn_RefSem.dec_term t k = Lam_cbn_RefSem.ed_val v ->
            trans (c_eval t c) (c_apply c v)
  | t_term : forall (t : Lam_cbn_RefSem.term)
               (k : Lam_cbn_RefSem.ckind)
               (c : Lam_cbn_RefSem.context ick k)
               (t0 : Lam_cbn_RefSem.term) (k' : Lam_cbn_RefSem.ckind)
               (ec : Lam_cbn_RefSem.elem_context_kinded k k'),
             Lam_cbn_RefSem.dec_term t k =
             Lam_cbn_RefSem.ed_dec k' t0 ec ->
             trans (c_eval t c) (c_eval t0 (ec =: c))
  | t_cred : forall (k k' : Lam_cbn_RefSem.ckind)
               (ec : Lam_cbn_RefSem.elem_context_kinded k k')
               (c : Lam_cbn_RefSem.context ick k)
               (v : Lam_cbn_RefSem.value k')
```

```
                (r : Lam_cbn_RefSem.redex k) (t : Lam_cbn_RefSem.term),
              Lam_cbn_RefSem.dec_context ec v = Lam_cbn_RefSem.ed_red r ->
              Lam_cbn_RefSem.contract r = Some t ->
              trans (c_apply (ec =: c) v) (c_eval t c)
  | t_cval : forall (k k' : Lam_cbn_RefSem.ckind)
                (ec : Lam_cbn_RefSem.elem_context_kinded k k')
                (c : Lam_cbn_RefSem.context ick k)
                (v : Lam_cbn_RefSem.value k')
                (v0 : Lam_cbn_RefSem.value k),
              Lam_cbn_RefSem.dec_context ec v = Lam_cbn_RefSem.ed_val v0 ->
              trans (c_apply (ec =: c) v) (c_apply c v0)
  | t_cterm : forall (k k' k'' : Lam_cbn_RefSem.ckind)
                 (ec : Lam_cbn_RefSem.elem_context_kinded k k')
                 (ec0 : Lam_cbn_RefSem.elem_context_kinded k k'')
                 (c : Lam_cbn_RefSem.context ick k)
                 (v : Lam_cbn_RefSem.value k')
                 (t : Lam_cbn_RefSem.term),
               Lam_cbn_RefSem.dec_context ec v =
               Lam_cbn_RefSem.ed_dec k'' t ec0 ->
               trans (c_apply (ec =: c) v) (c_eval t (ec0 =: c))
Definition trans_ind :
  forall P : configuration -> configuration -> Prop,
  (forall (t : Lam_cbn_RefSem.term) (k : Lam_cbn_RefSem.ckind)
     (c : Lam_cbn_RefSem.context ick k) (r : Lam_cbn_RefSem.redex k)
     (t0 : Lam_cbn_RefSem.term),
   Lam_cbn_RefSem.dec_term t k = Lam_cbn_RefSem.ed_red r ->
   Lam_cbn_RefSem.contract r = Some t0 -> P (c_eval t c) (c_eval t0 c)) ->
  (forall (t : Lam_cbn_RefSem.term) (k : Lam_cbn_RefSem.ckind)
     (c : Lam_cbn_RefSem.context ick k) (v : Lam_cbn_RefSem.value k),
   Lam_cbn_RefSem.dec_term t k = Lam_cbn_RefSem.ed_val v ->
   P (c_eval t c) (c_apply c v)) ->
  (forall (t : Lam_cbn_RefSem.term) (k : Lam_cbn_RefSem.ckind)
     (c : Lam_cbn_RefSem.context ick k) (t0 : Lam_cbn_RefSem.term)
     (k' : Lam_cbn_RefSem.ckind)
     (ec : Lam_cbn_RefSem.elem_context_kinded k k'),
   Lam_cbn_RefSem.dec_term t k = Lam_cbn_RefSem.ed_dec k' t0 ec ->
   P (c_eval t c) (c_eval t0 (ec =: c))) ->
  (forall (k k' : Lam_cbn_RefSem.ckind)
     (ec : Lam_cbn_RefSem.elem_context_kinded k k')
     (c : Lam_cbn_RefSem.context ick k) (v : Lam_cbn_RefSem.value k')
     (r : Lam_cbn_RefSem.redex k) (t : Lam_cbn_RefSem.term),
   Lam_cbn_RefSem.dec_context ec v = Lam_cbn_RefSem.ed_red r ->
   Lam_cbn_RefSem.contract r = Some t ->
   P (c_apply (ec =: c) v) (c_eval t c)) ->
  (forall (k k' : Lam_cbn_RefSem.ckind)
     (ec : Lam_cbn_RefSem.elem_context_kinded k k')
```

```
    (c : Lam_cbn_RefSem.context ick k) (v : Lam_cbn_RefSem.value k')
    (v0 : Lam_cbn_RefSem.value k),
   Lam_cbn_RefSem.dec_context ec v = Lam_cbn_RefSem.ed_val v0 ->
   P (c_apply (ec =: c) v) (c_apply c v0)) ->
  (forall (k k' k'' : Lam_cbn_RefSem.ckind)
    (ec : Lam_cbn_RefSem.elem_context_kinded k k')
    (ec0 : Lam_cbn_RefSem.elem_context_kinded k k'')
    (c : Lam_cbn_RefSem.context ick k) (v : Lam_cbn_RefSem.value k')
    (t : Lam_cbn_RefSem.term),
   Lam_cbn_RefSem.dec_context ec v = Lam_cbn_RefSem.ed_dec k'' t ec0 ->
   P (c_apply (ec =: c) v) (c_eval t (ec0 =: c))) ->
  forall c c0 : configuration, trans c c0 -> P c c0.
Definition trans_sind :
  forall P : configuration -> configuration -> SProp,
  (forall (t : Lam_cbn_RefSem.term) (k : Lam_cbn_RefSem.ckind)
    (c : Lam_cbn_RefSem.context ick k) (r : Lam_cbn_RefSem.redex k)
    (t0 : Lam_cbn_RefSem.term),
   Lam_cbn_RefSem.dec_term t k = Lam_cbn_RefSem.ed_red r ->
   Lam_cbn_RefSem.contract r = Some t0 -> P (c_eval t c) (c_eval t0 c)) ->
  (forall (t : Lam_cbn_RefSem.term) (k : Lam_cbn_RefSem.ckind)
    (c : Lam_cbn_RefSem.context ick k) (v : Lam_cbn_RefSem.value k),
   Lam_cbn_RefSem.dec_term t k = Lam_cbn_RefSem.ed_val v ->
   P (c_eval t c) (c_apply c v)) ->
  (forall (t : Lam_cbn_RefSem.term) (k : Lam_cbn_RefSem.ckind)
    (c : Lam_cbn_RefSem.context ick k) (t0 : Lam_cbn_RefSem.term)
    (k' : Lam_cbn_RefSem.ckind)
    (ec : Lam_cbn_RefSem.elem_context_kinded k k'),
   Lam_cbn_RefSem.dec_term t k = Lam_cbn_RefSem.ed_dec k' t0 ec ->
   P (c_eval t c) (c_eval t0 (ec =: c))) ->
  (forall (k k' : Lam_cbn_RefSem.ckind)
    (ec : Lam_cbn_RefSem.elem_context_kinded k k')
    (c : Lam_cbn_RefSem.context ick k) (v : Lam_cbn_RefSem.value k')
    (r : Lam_cbn_RefSem.redex k) (t : Lam_cbn_RefSem.term),
   Lam_cbn_RefSem.dec_context ec v = Lam_cbn_RefSem.ed_red r ->
   Lam_cbn_RefSem.contract r = Some t ->
   P (c_apply (ec =: c) v) (c_eval t c)) ->
  (forall (k k' : Lam_cbn_RefSem.ckind)
    (ec : Lam_cbn_RefSem.elem_context_kinded k k')
    (c : Lam_cbn_RefSem.context ick k) (v : Lam_cbn_RefSem.value k')
    (v0 : Lam_cbn_RefSem.value k),
   Lam_cbn_RefSem.dec_context ec v = Lam_cbn_RefSem.ed_val v0 ->
   P (c_apply (ec =: c) v) (c_apply c v0)) ->
  (forall (k k' k'' : Lam_cbn_RefSem.ckind)
    (ec : Lam_cbn_RefSem.elem_context_kinded k k')
    (ec0 : Lam_cbn_RefSem.elem_context_kinded k k'')
    (c : Lam_cbn_RefSem.context ick k) (v : Lam_cbn_RefSem.value k')
```

```
      (t : Lam_cbn_RefSem.term),
    Lam_cbn_RefSem.dec_context ec v = Lam_cbn_RefSem.ed_dec k'' t ec0 ->
    P (c_apply (ec =: c) v) (c_eval t (ec0 =: c))) ->
  forall c c0 : configuration, trans c c0 -> P c c0.
Definition transition : configuration -> configuration -> Prop.
Definition dnext_conf : configuration -> option configuration.
Definition next_conf :
  Entropy.entropy -> configuration -> option configuration.
Definition rws : rewriting_system.REWRITING_SYSTEM configuration.
Record SafeRegion (P : configuration -> Prop) : Prop := Build_SafeRegion
  { preservation : forall st1 st2 : configuration,
                   P st1 -> rewriting_system.transition st1 st2 -> P st2;
    progress : forall st1 : configuration,
               P st1 ->
               is_final st1 \/
               (exists st2 : configuration,
                  rewriting_system.transition st1 st2) }
Definition preservation :
  forall P : configuration -> Prop,
  SafeRegion P ->
  forall st1 st2 : configuration,
  P st1 -> rewriting_system.transition st1 st2 -> P st2.
Definition progress :
  forall P : configuration -> Prop,
  SafeRegion P ->
  forall st1 : configuration,
  P st1 ->
  is_final st1 \/
  (exists st2 : configuration, rewriting_system.transition st1 st2).
Definition value_to_conf : value -> configuration.
Parameter final_correct :
  forall c : configuration,
  final c <> None ->
  ~ (exists c' : configuration, rewriting_system.transition c c').
Parameter trans_computable0 :
  forall c1 c2 : configuration,
  rewriting_system.transition c1 c2 <-> dnext_conf c1 = Some c2.
Parameter trans_computable :
  forall c1 c2 : configuration,
  rewriting_system.transition c1 c2 <->
  (exists e : Entropy.entropy, next_conf e c1 = Some c2).
Parameter finals_are_vals :
  forall (st : configuration) (v : value), final st = Some v <-> st = v.
Parameter dnext_is_next :
  forall (e : Entropy.entropy) (c : configuration),
  next_conf e c = dnext_conf c.
```

```
  End
:= (RefEvalApplyMachine Lam_cbn_RefSem)
```