BRICS RS-08-3 J. Munk: A Study of Syntactic and Semantic Artifacts and its Application to Lambda Definability, Strong Normalization,

BRICS

Basic Research in Computer Science

A Study of Syntactic and Semantic Artifacts and its Application to Lambda Definability, Strong Normalization, and Weak Normalization in the Presence of State

Johan Munk

BRICS Report Series **RS-08-3**

ISSN 0909-0878 **April 2008**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark

Telephone: +45 8942 9300
Telefax: +45 8942 5601
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/08/3/`

A Study of Syntactic and Semantic Artifacts and its Application to Lambda Definability, Strong Normalization, and Weak Normalization in the Presence of State

Johan Munk[1]
(Advisor: Olivier Danvy)

May 4, 2007. Revised August 22, 2007

[1]IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: <jmunk@daimi.au.dk>
Student id: 20000345

Abstract

Church's lambda-calculus underlies the syntax (i.e., the form) and the semantics (i.e., the meaning) of functional programs. This thesis is dedicated to studying man-made constructs (i.e., artifacts) in the lambda calculus. For example, one puts the expressive power of the lambda calculus to the test in the area of lambda definability. In this area, we present a course-of-value representation bridging Church numerals and Scott numerals. We then turn to weak and strong normalization using Danvy et al.'s syntactic and functional correspondences. We give a new account of Felleisen and Hieb's syntactic theory of state, and of abstract machines for strong normalization due to Curien, Crégut, Lescanne, and Kluge.

Contents

Introduction

Part I Eric Raymond once wrote that learning Lisp makes one a better programmer, and we believe that the same holds for functional programming. In fact, it even seems to us that studying Church's λ-calculus, which is standard material for functional programmers, makes one a better computer scientist altogether.

We thus begin this thesis with this standard material and we formally introduce Church's original version of the λ-calculus and the most standard properties of the λ-calculus (Chapter 1).

Understanding the λ-calculus, however, does not come with understanding its basic elements in separation. We thus turn to lambda definability in Chapter 2, where we show how to syntactically represent various functions in the λ-calculus, i.e., we study *syntactic artifacts* (i.e., man-made constructs) in the λ-calculus. Going beyond the usual basic lambda representations, we identify a course-of-value representation underlying both the Church numerals and the Scott numerals. This representation enables us to improve on known lambda representations, and is a joint work with Olivier Danvy.

Various extensions and modifications of the λ-calculus exist, with Plotkin's λ_v-calculus as one of the most prominent examples. We give a brief review in Chapter 3.

In Chapter 4, we introduce three kinds of *semantic artifacts*, to give a formal 'meaning' to programs in a language: (1) denotational semantics, in which definitional interpreters have their roots, (2) reduction semantics, which Felleisen credits Plotkin's work as the main inspiration for, and (3) abstract machines, with Landin's SECD machine as the first example, historically.

So, Landin's pioneering work on connecting the λ-calculus to functional programming languages, Plotkin's formal proofs of the correspondences between λ-calculi and evaluation mechanisms, and Reynolds's work on definitional interpreters have been an inspiration to many computer scientists. In Chapter 5, we review Plotkin's results, which bridge the gap between calculi and standard programming languages. We then introduce the 'syntactic correspondence', which connects reduction semantics and abstract machines and has been developed by Biernacka, Danvy, and Nielsen, and we introduce the 'functional correspondence', which connects interpreters and abstract machines and has been developed by Ager, Biernacki, Danvy, and Midtgaard. Danvy et al. credit Reynolds for pioneering the functional correspondence by CPS-transforming and defunctionalizing an interpreter to make it first-order. The syntactic correspondence, however, is original to Danvy and Nielsen.

With the syntactic correspondence as our main tool, we investigate the relation between reduction semantics and abstract machines in the presence of state variables and assignment constructs in the language (Section 6). We do so by choosing an existing calculus including state variables — Felleisen and Hieb's λ_v-$S(\mathbf{t})$-calculus. We show that the syntactic correspondence, perhaps unexpectedly, also applies in such an impure setting.

Part II The second part of this thesis addresses *strong normalization*, i.e., normalization of terms to full normal forms, and our domain is terms in Church's λ-calculus. We do not consider strong normalization in, e.g., Plotkin's λ_v-calculus.

We take the usefulness of strong normalization for given, and thus consider the problem of implementing the partial function mapping terms to normal forms. Typically, one implements a normalizer in either of the following three ways:

(i) through normalization by evaluation: one defines an evaluation function from terms to values, mapping syntactically equivalent terms to the same semantic value, and a left-inverse reification function from values to terms. The normalization function is defined as the composition of these two functions. Such normalization functions are usually higher-order compositional functions specified in direct style.

(ii) with an abstract machine: one specifies a state-transition function, the iteration of which yields a normal form, given a term.

(iii) within the calculus: given a one-step reduction relation and a normalization strategy, one iterates the one-step reduction relation according to the normalization strategy. In this thesis we only consider deterministic strategies.

The approaches (i) and (ii) are 'reduction-free' whereas approach (iii) is 'reduction-based' according to Danvy et al.'s use of the words.

So far, the functional correspondence and the syntactic correspondence have only been investigated in a weak setting, i.e., when considering evaluation mechanisms found in programming languages like Scheme or Haskell. Considering strong normalization, a new area of applications appears, and it is the point of Part II to illustrate this new area.

All the reduction semantics we know of define a weak reduction scheme. In Chapter 7 we show how to define a reduction semantics for strong normalization with *actual substitution*, i.e., with use of a meta-substitution construct as usually employed in the definition of reductions in the λ-calculus.

In Chapter 8 we migrate to the world of *explicit substitutions*, i.e., where substitutions are represented explicitly in the terms of the calculi. We present a new calculus — the $\lambda\hat{s}$-calculus — which is a direct offspring of the standard implementation of actual substitution and therefore inherits the standard properties of the λ-calculus. We show how the syntactic correspondence applies to a strategy for strong normalization in this calculus (represented as a reduction semantics): we mechanically derive an abstract machine for strong normalization.

In Chapter 9 we investigate a normalizer defined by Lescanne. We show that this normalizer is a cousin of an abstract machine derived from a strategy for strong normalization in the $\lambda\hat{\hat{s}}$-calculus, which is a variant of the $\lambda\hat{s}$-calculus. We also review how Lescanne's normalizer can be improved in practice.

Curien has also defined a normalizer for strong normalization. In Chapter 10 we present this normalizer and we show that the syntactic correspondence and the functional correspondence mechanically link reduction semantics to abstract machines and abstract machines to normalization functions for Curien's normalizer. This chapter is joint work with Olivier Danvy and Kevin Millikin.

As yet another example we apply the functional correspondence and the syntactic correspondence to Crégut's `KN`-machine for strong normalization and we present the corresponding normalization function and reduction semantics (Chapter 11).

In his recent textbook, Kluge presents an abstract machine for strong normalization — the `HOR`-machine. This machine is introduced as the implementation of a head-order reduction strategy. In Chapter 12, we review the underlying normalization mechanism and we show that Crégut's `KN`-machine and the `HOR`-machine in essence are the same machine.

Technical details This thesis is not associated to any implementation, though every abstract machine, reduction semantics, interpreter, and normalization function as well as all derivations have been implemented in Standard ML. The author used a Scheme-to-SML parser he implemented during his undergraduate compiler-construction course. This parser was instrumental to define useful test examples for the various implementations. Finally, every representation of functions developed in Chapter 2 has been implemented in Scheme.

Part I

λ-calculi and programming languages

Chapter 1

The λ-calculus

In the 1930's Church invented the *λ-calculus* [11]. Originally the λ-calculus was part of a bigger work on finding a foundation for logic [44, page 53]. In an attempt to remove ambiguities in mathematical statements about functions, Church suggested the *lambda notation* to represent functions and function applications. The λ-calculus is a formal system and gives precise definitions of *meaningful formulas* and all possible ways to manipulate such meaningful or *well-formed* formulas or just λ-*terms*. The definitions formally specify how λ-terms relate to each other in the λ-calculus.

Roadmap In this chapter we introduce the λ-calculus. We define the λ-terms in Section 1.1 including standard conventions on the λ-terms and properties on variables. We define a fundamental relation between λ-terms (Section 1.2) used to define equality in the calculus (Section 1.3). Equipped with a notion of λ-terms in normal form, we review standard properties about equivalence of terms to normal forms (Section 1.4) and algorithms to find a normal form equivalent to a given term, when one exists (Section 1.5). We touch on other representations of the λ-calculus in Section 1.6 and Section 1.7.

1.1 λ-terms

In the original presentation [11, page 8] Church defines the set of well-formed formulas inductively via symbols like 'λ', '(', and ')', such that terms are finite sequences of symbols. In other words, terms have a linear structure where only a subset of all formulas are well-formed. Church hence uses parentheses for grouping of sub-formulas to avoid ambiguities and he proves various properties on well-formed formulas like, e.g., matching parentheses. We abandon this linear notion of formulas and directly define the set of λ-terms via an abstract grammar:

$$
\begin{array}{llll}
\mathit{Var} & x & & (\mathit{unspecified}) \\
\mathit{Term} & t & ::= & x \mid \lambda x.t \mid t\ t
\end{array}
$$

All terms are built inductively from an unspecified (but enumerable) set of *variables*, from λ-*abstractions*, and from terms composed of two terms in juxtaposition: *applications*.[1]

[1] Applications was originally called *combinations*. In 1975 Plotkin still used that term [59], and so did Abelson and Sussman (with Sussman) in 1985 [1].

The grammar is abstract in the sense that it does not define a language of character strings but a language of trees where internal nodes correspond to the use of either the abstraction construct or the application construct, and with all leaves being variables.

The major part of the properties proved by Church and Kleene about well-formed formulas (i.e., λ-terms) are not relevant in an abstract setting. In the rest of this text all grammars define abstract syntax trees.

Pure and applied λ-calculi Church's original calculus is sometimes called the *pure λ-calculus* to emphasize that no basic constants or functional constants are included in the term language. Regularly since Landin's work *The mechanical evaluation of expressions* [50], such constants are included in λ-terms [33, 40, 59]. Such an 'impure' λ-calculus is usually called an 'applied' λ-calculus [45, Section 4.7].

Untyped and typed calculi In the grammar of terms, no notion of type is included. The λ-calculus is sometimes also referred to as the *untyped λ-calculus* to emphasize the difference from various kinds of typed λ-calculi, which have subsequently been developed. In this text we only consider untyped λ-calculi.

1.1.1 Conventions

Concrete conventions We have defined terms with abstract-syntax constructions, which eliminate the need for parentheses in grouping of sub-terms. Unfortunately, abstract-syntax trees are not convenient when presenting sample terms in a text. Hence in the following we use a linear abbreviation for the syntax trees when giving examples.

We will use parentheses for grouping when they are needed. We follow two conventions:

(i) Application is left associative: $t_1\, t_2\, t_3$ means $(t_1\, t_2)\, t_3$.

(ii) Application has higher precedence than abstraction: $\lambda x.t_1\, t_2$ means $\lambda x.(t_1\, t_2)$

These conventions keep the number of parentheses to a minimum and are standard.

Abstract conventions In theorems, propositions and various kinds of defining equations we use variables like x and t. These variables (possibly with a subscript or a superscript) are *metavariables* ranging over a set of objects normally defined by a grammar. For example, t and t' range over the set *Term* as defined above.

When we define equations by cases, the use of metavariables lets us exploit an advanced but straightforward kind of pattern matching. Examples clarify the idea:

$$\begin{array}{lrcl} (a) & \mathit{foo}\; t & = & \ldots \\ (b) & \mathit{bar}\; x & = & \ldots \\ (c) & \mathit{baz}\; ((\lambda x.t)\; t') & = & \ldots \end{array}$$

Here the pattern in equation (a) matches every term, the pattern in (b) matches only the subset of terms that are also variables and the pattern in (c) matches all applications where the left-hand sub-term is an abstraction. A pattern thus implicitly corresponds to a universal quantifier with a condition. In the right-hand side of the equations, the metavariables are bound and can be used as part of the expressions.

In logical expressions the use of metavariables also implicitly implies an universal quantifier. In each case the quantifier will be clear from context.

1.1.2 Free variables and bound variables

Free variables The notion of *free variables* of a term $\mathtt{t}$, $\mathtt{FV}\,\mathtt{t}$, is inductively defined as follows:

$$\begin{array}{rcl} \mathtt{FV}\,x & = & \{x\} \\ \mathtt{FV}\,(\lambda x.t) & = & (\mathtt{FV}\,t)\backslash\{x\} \\ \mathtt{FV}\,(t\,t') & = & (\mathtt{FV}\,t) \cup (\mathtt{FV}\,t') \end{array}$$

If $\mathtt{FV}\,t = \{\}$, t is said to be *closed*.

Bound variables Likewise, a corresponding notion of *bound variables* is inductively defined as follows:

$$\begin{array}{rcl} \mathtt{BV}\,x & = & \{\} \\ \mathtt{BV}\,(\lambda x.t) & = & (\mathtt{BV}\,t) \cup \{x\} \\ \mathtt{BV}\,(t\,t') & = & (\mathtt{BV}\,t) \cup (\mathtt{BV}\,t') \end{array}$$

Variable occurrences In general the set of bound variables and the set of free variables of a term are not disjoint, i.e., a variable can be bound and free in the same term. For example, $\mathtt{FV}\ ((\lambda x.x)\ x) = \mathtt{BV}\ ((\lambda x.x)\ x) = \{x\}$. For that reason the different *occurrences* of a variable in a term are distinguished. An occurrence of a variable x is bound in t if and only if that occurrence is a sub-term of the abstraction $\lambda x.t'$ for some term t' which itself is a sub-term of t. In other words, an abstraction $\lambda x.t'$ binds all occurrences of the variable x in the *body* t' of the abstraction. By this definition an occurrence of a variable is either bound or free, and this property of an occurrence of a variable is hence well-defined. In the above sample term, $(\lambda x.x)\ x$, only the right-most occurrence of x is free.

The λK-calculus or the λI-calculus Church distinguished between two formal systems called the λ**I**-*calculus* and the λ**K**-*calculus*. Terms in the λ**I**-calculus have a condition on abstractions: For $\lambda x.t$ to be well-formed, x must have a free occurrence in t, i.e., the condition is $x \in \mathtt{FV}\,t$. In the λ**K**-calculus this condition is not present. In other words, what we call the λ-calculus is what Church originally called the λ**K**-calculus.

1.1.3 λ-terms modulo bound variable names

In the following, terms only differing in choice of bound variable names are not distinguished. This choice is a standard convention also used by Barendregt [7, Convention 2.1.12]. For example, the terms $\lambda x.xc$ and $\lambda y.yc$ are identified, but $\lambda x.xc$ and $\lambda x.xd$ are not. To formalize this notion of equality an equivalence relation $=_\alpha$ is inductively defined:

$$\frac{}{x =_\alpha x} \qquad \frac{t_1 =_\alpha t_1',\ t_2 =_\alpha t_2'}{t_1\ t_2 =_\alpha t_1'\ t_2'} \qquad \frac{t_1\{y/x\} =_\alpha t_2,\ x = y \text{ or } (y \notin \mathtt{FV}\,t_1 \text{ and } y \notin \mathtt{BV}\,t_1)}{\lambda x.t_1 =_\alpha \lambda y.t_2}$$

In the third rule the first condition $t_1\{y/x\}$ denotes the term t_1 with all free occurrences of x replaced by y. In the second condition $x = y$ is included such that $=_\alpha$ is reflexive. We restrict y via the condition $(y \notin \mathtt{FV}\,t_1$ and $y \notin \mathtt{BV}\,t_1)$ such that y is 'fresh'. In the following a term is a representative of an equivalence class generated by $=_\alpha$.

1.2 Reductions and normal forms

Notion of reduction The fundamental relation in the λ-calculus between λ-terms is called β and is the *notion of reduction* on λ-terms:

$$((\lambda x.t)\, t', t\{t'/x\}) \in \beta$$

where $t\{t'/x\}$ again is a meta-notation denoting the result of substituting t' for free occurrences of x in t:

$$\begin{array}{rcl} x\{t/x\} & = & t \\ y\{t/x\} & = & y, \text{ if } x \neq y \\ (t_1\, t_2)\{t/x\} & = & (t_1\{t/x\})\,(t_2\{t/x\}) \\ (\lambda x.t)\{t'/x\} & = & \lambda x.t \\ (\lambda y.t)\{t'/x\} & = & \lambda y.t\{t'/x\}, \text{ if } x \neq y \text{ and } y \notin \mathtt{FV}\, t' \end{array}$$

The substitution is left undefined for terms where a renaming of bound variables is needed: Because of $=_\alpha$ introduced in Section 1.1.3 it is — given $\lambda y.t$ — always possible to choose $\lambda y'.t'$ such that $\lambda y'.t' =_\alpha \lambda y.t$ and $(\lambda y'.t')\{t''/x\}$ is defined for all t'' and x.

Traditionally β is stated as a *contraction rule*:

$$\beta : \qquad (\lambda x.t)\, t' \to t\{t'/x\}$$

If $(t, t') \in \beta$ the first component t is called a β-*redex*, and the second component t' is called the corresponding *contractum*. It follows immediately that if a given term t is a variable, an abstraction, or an application where the left sub-term is not an abstraction β does not relate t to any term: t does not take the form of a β-redex, and no β-*reduction* is possible.[2]

β-normal forms A term that does not contain *any* β-redex as a sub-term, is said to be in β-*normal form*. A grammar generating terms in β-normal form reads:

$$\begin{array}{rrcl} \textit{ANForm} & a & ::= & x \mid a\, n \\ \textit{NForm} & n & ::= & a \mid \lambda x.n \end{array}$$

No applications in the terms have an abstraction as left sub-term.

1.3 One-step reduction and equality

Compatibility As stated in Section 1.2, the notion of reduction only relates a term t to a term t' if t takes the form of a β-redex and t' is the corresponding contractum. To be able to relate t to t' even when t is not overall a β-redex three compatibility rules are formed:

$$\frac{t_1 \to t_1'}{t_1\, t_2 \to t_1'\, t_2} \qquad \frac{t_2 \to t_2'}{t_1\, t_2 \to t_1\, t_2'} \qquad \frac{t \to t'}{\lambda x.t \to \lambda x.t'}$$

From these rules it follows that any sub-term that is also a β-redex can be the subject of a reduction.

[2] An alternative way to use α-equivalence on terms, is to treat changes of variable names explicitly by defining a notion of reduction α, such that an α-reduction constitute the change of a bound variable and the corresponding bound occurrences. Church originally defined such an α-rule.

Alternative specification of compatibility The above rules specify in what syntactic *contexts* β-reductions can be performed:

$$\textit{Context} \qquad C \ ::= \ [\,] \mid C\ t \mid t\ C \mid \lambda x.C$$

A context is either just a hole or a term with a hole instead of one sub-term. C[t] denotes the term obtained from textually replacing the hole in C by the term t. With $t = (\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))$ the following listing illustrates the principle:

$$\begin{array}{ll}
t = C_1[t_1], & C_1 = (\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ [\,])\ (\lambda x.w\ (x\ x))) \text{ and } t_1 = x\ x \\
t = C_2[t_2], & C_2 = (\lambda x.[\,])\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))) \text{ and } t_2 = (\lambda y.y)\ z \\
t = C_3[t_3], & C_3 = (\lambda x.(\lambda y.y)\ z)\ [\,] \text{ and } t_3 = (\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)) \\
t = C_4[t_4], & C_4 = (\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ [\,]))) \text{ and } t_4 = x \\
t = C_5[t_5], & C_5 = [\,] \text{ and } t_5 = (\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))) \\
\vdots &
\end{array}$$

Of all the possible contexts only three result in the corresponding t_i being a β-redex. The redexes are t_2, t_3, and t_5.

The compatibility rules are equivalently specified with a grammar of contexts: Two terms t and t' are related if a context C and a β-redex r exist, such that $t = C[r]$ and t' is t with r replaced by the contractum of r. Formally:

$$\rightarrow_\beta: \qquad t \rightarrow_\beta t' \quad \text{iff} \quad t = C[r], t' = C[r'], (r, r') \in \beta$$

It follows that $\rightarrow_\beta$ relates two terms t and t' if t' is the result of choosing *any* sub-term of t that is as β-redex and replace that redex with the corresponding contractum. $\rightarrow_\beta$ is called *one-step* β*-reduction* and is said to be defined as the *compatible closure* of the notion of reduction relative to the specified contexts.

For the above example term, $\rightarrow_\beta$ relates t to exactly the three terms obtained by performing the contraction of one of the β-redexes t_2, t_3, and t_5:

$$\begin{array}{lll}
(C_2[t_2],\ (\lambda x.z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))) & \in & \rightarrow_\beta \\
(C_3[t_3],\ (\lambda x.(\lambda y.y)\ z)\ (w\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))))) & \in & \rightarrow_\beta \\
(C_5[t_5],\ (\lambda y.y)\ z) & \in & \rightarrow_\beta
\end{array}$$

β-reduction The reflexive transitive closure of one-step β-reduction defines β*-reduction*, $\rightarrow_\beta^*$, which relates two terms if the second term can be derived in zero or more steps from the first term by a series of β-reductions. Simple examples of terms related by a series of β-reductions are (starting with the sample term introduced above):

$$\begin{array}{lll}
(\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))) & \rightarrow_\beta & (\lambda y.y)\ z \\
& \rightarrow_\beta & z \\
\\
(\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))) & \rightarrow_\beta & (\lambda x.z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))) \\
& \rightarrow_\beta & z \\
\\
(\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))) & \rightarrow_\beta & (\lambda x.(\lambda y.y)\ z)\ (w\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))) \\
& \rightarrow_\beta & (\lambda x.(\lambda y.y)\ z)\ (w\ (w\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))))) \\
& \rightarrow_\beta^* & (\lambda x.(\lambda y.y)\ z)\ (w\ (w\ \ldots((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))\ldots))
\end{array}$$

In the first case a normal form is obtained via a series of reductions consisting of two β-reductions. The same normal form is obtained by another series of reductions in the second case. In the third case an infinite series starting with the same term is outlined. Reductions are performed but no normal form is obtained.

1.3.1 β-equivalence and convertibility

β-equivalence or *convertibility* of terms [11, page 13] is defined as the smallest equivalence relation $=_\beta$ over $\rightarrow_\beta$ (i.e., the symmetric, reflexive, and transitive closure of $\rightarrow_\beta$). For example, the λ-terms above are all *equivalent* in the λ-calculus, and they are therefore also said to be *convertible*. We will use the term 'equivalent' in this text.[3]

Because of the equivalence relation $=_\alpha$, defined in Section 1.1.3, the equations in the system on the one hand are independent of the actual names of the *bound* variables. On the other hand the equations are dependent on the names of *free* variables.

1.4 Uniqueness of normal forms

Divergence As illustrated by the last example series in Section 1.3, in general not all terms are equivalent to a normal form. A counter example is the right sub-term of the sample term from Section 1.3:

$$\begin{array}{lll} (\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)) & =_\beta & w\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))) \\ & =_\beta & w\ (w\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))) \\ & =_\beta & w\ (w\ \ldots((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))\ldots) \end{array}$$

The initial term is on the one hand not a normal form, because it is an application that constitutes a β-redex. On the other hand, it only contains this single β-redex, and it is not possible by a series of reductions to remove the initial β-redex. Finally, the term can not be obtained by some reduction without the existence of a β-redex. That is, at least one β-redex will always be present in terms equivalent to the terms in the example, and by definition none of the terms are equivalent to a normal form. Terms not equivalent to a normal form are said to *diverge*.

Unique normal forms As mentioned in Section 1.3 it follows from the compatibility rules that the next redex to contract in a term with more than one redex can be arbitrarily chosen. Nevertheless it can be shown that no term is equivalent to more than one normal form. This property of the λ-calculus follows from the fact that the notion of reduction is *Church-Rosser*, which means the relation $\rightarrow_\beta^*$ satisfies a diamond property:

Theorem 1 (Church-Rosser)
Let $t \rightarrow_\beta^* t_1$*, and* $t \rightarrow_\beta^* t_2$*.*
Then there exists t_3 *such that* $t_1 \rightarrow_\beta^* t_3$*, and* $t_2 \rightarrow_\beta^* t_3$*.*

A proof can be found in Barendregt's textbook [7, page 62]. Assuming t_1 and t_2 are different normal forms immediately gives a contradiction. It follows that if t_1 and t_2 both are normal

[3] Instead of defining convertibility as the smallest equivalence relation over $\rightarrow_\beta$, Church defined α-conversions, β-conversions, and *inverse* β-conversions (called an *expansion*). In that case, convertibility is just defined as the compatible closure of these three conversion rules.

forms, they have to be equal. If a term t is equivalent to a normal form n, it is therefore well-defined to say *the normal form* of t and t *has normal form* n.

The Church-Rosser property also implies that if a term has a normal form, this normal form can always be obtained by zero or more reductions. Furthermore, it is not possible to perform a series of reductions on the term without the possibility to extend that series and obtain the normal form. For example, the last reduction sequence in Section 1.3 can at any point be made finite by reducing the out-most redex and in one more reduction obtain the normal form.

Consistency Uniqueness of normal forms also imply that the λ-calculus is *consistent*: for two different normal forms n, n' it holds that $n \neq_\beta n'$ which means that $=_\beta$ defines more than one equivalence class.

1.5 Reduction strategies

As stated in Section 1.4 the normal form of a term, if it exists, can always be found by a series of reductions. Though in general the redexes for contraction can not be chosen arbitrarily. A counter example is the sample term from Section 1.3:

$$(\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))$$

In that section three different reduction sequences are presented: The first sequence obtains the normal form z of the term by twice choosing the whole term as the redex for contraction. The second sequence obtains z by choosing the inner leftmost redex first and secondly the whole term as the redex for contraction. However, the third sequence can be extended forever without obtaining the normal form, by choosing the rightmost redex repeatedly.

A *reduction strategy* constitutes a way to deterministically select a β-redex and perform the contraction given a term not in normal form. Because β is a function on redexes, a reduction strategy turns out to be a partial function on terms, such that t is mapped to t', where t' is t with one redex replaced by the corresponding contractum.

1.5.1 Normal-order reduction

Normal-order reduction $\mapsto_{\bar{n}}$ constitutes a reduction strategy for the λ-calculus and is defined as always choosing the *leftmost* of the *outermost* redexes, where an outermost redex means a redex that is not a sub-term of another redex itself.[4] Formally, the relation is defined as a restricted compatible closure of β to obtain a function which is defined on all terms not in normal form:

$$\frac{(t, t') \in \beta}{t \mapsto_{\bar{n}} t'} \qquad \frac{\forall t : (t_1\ t_2, t) \notin \beta,\ t_1 \mapsto_{\bar{n}} t_1'}{t_1\ t_2 \mapsto_{\bar{n}} t_1'\ t_2}$$

$$\frac{t \mapsto_{\bar{n}} t'}{\lambda x.t \mapsto_{\bar{n}} \lambda x.t'} \qquad \frac{t_2 \mapsto_{\bar{n}} t_2'}{a\ t_2 \mapsto_{\bar{n}} a\ t_2'}$$

The metavariable convention from Section 1.1.1 lets us state the fourth rule very concisely. In the definition of normal forms on page 8, a is defined to range over normal forms that

[4] Normal-order reduction is also known as *Standard reduction*.

are not abstractions. The fourth rule hence says: When the left sub-term is a normal form that is not an abstraction, the overall application cannot be β-reduced. Normalization must continue in the right sub-term and $(a\ t_2, t) \notin \beta$ is implied.

Normal-order reduction is well-defined since no rules overlap and β is a partial function. Because $\rightarrow_\beta$ of Section 1.3 is a proper relation, it is called a one-step reduction *relation*. Because $\mapsto_{\bar{n}}$ is a function on terms, it is called a one-step reduction *function*. Also, $\mapsto_{\bar{n}}$ is partial, because it is not defined on β-normal forms.

Equality in the λ-calculus has not changed by the above definition of the restricted compatibility closure. Taking $\mapsto^*_{\bar{n}}$ as the reflexive transitive closure of $\mapsto_{\bar{n}}$, $=_{\bar{n}}$ is defined as the smallest equivalence over $\mapsto^*_{\bar{n}}$. The term $(\lambda x.w\,(x\,x))\,(\lambda x.w\,(x\,x))\,((\lambda x.y)\,(x\,x))$ illustrates that $=_{\bar{n}}$ defines a proper subset of $=_\beta$: On one hand an equation in the λ-calculus reads

$$(\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))\ ((\lambda x.y)\ (x\ x)) \quad =_\beta \quad (\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))\ y$$

which is established by a contraction of the redex $(\lambda x.y)\ (x\ x)$. On the other hand $=_{\bar{n}}$ clearly does not contain that equation. Seen together $=_{\bar{n}} \subset =_\beta$ holds.

1.5.2 Standardization and normalization

The crucial property of normal-order reduction is: If any reduction sequence yields a normal form, normal-order reduction also will. Normal-order reduction thus defines a standard reduction strategy:

Theorem 2 (Standardization)
$t \rightarrow^*_\beta n$ *if and only if* $t \mapsto^*_{\bar{n}} n$

In Section 1.4 it was explained that if t is equivalent to the normal form n in the λ-calculus then n can always be obtained by a series of β-reductions. By the standardization theorem there is a deterministic way to select the β-reductions and obtain n. Performing a series of reductions on the sample term $(\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))$ as in Section 1.3 following the normal-order reduction strategy reads

$$\begin{aligned}(\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))) &\mapsto_{\bar{n}} (\lambda y.y)\ z \\ &\mapsto_{\bar{n}} z\end{aligned}$$

That is, the first series of reductions in Section 1.3 follows the normal-order reduction strategy.

Normalization Reducing a term to its normal form (if it exists) is called *normalization*. By Theorem 2 normalization in the λ-calculus can be defined:

$$\textit{normalize}_{\bar{n}}\ t = n \quad \text{iff} \quad t \mapsto^*_{\bar{n}} n$$

This function is well-defined since $\mapsto_{\bar{n}}$ is not defined on normal forms, and it is partial because it is not defined on terms with no normal form.

It follows from the rules of the normal-order reduction strategy in Section 1.5.1 that contractions are allowed also inside the body of abstractions and in the right sub-term of an application when the left sub-term is a normal form that is not an abstraction. When such contractions are allowed normalization yields β-normal forms and is also called *strong normalization*.

From the standardization theorem it also follows that equations involving terms with normal forms are all contained in both $=_\beta$ and $=_{\tilde{n}}$. In other words, the equations contained in $=_\beta$ but not in $=_{\tilde{n}}$ all involve terms with no normal form, which is also the crucial property of the sample term in Section 1.5.1. By construction the following proposition is immediate:

Proposition 1
$\textit{normalize}_{\tilde{n}}\ t = n \iff t =_\beta n$

1.5.3 Reducing to weak head normal forms

In Section 1.5.1 the normal-order reduction strategy for the λ-calculus was presented and in Section 1.5.2 strong normalization following this strategy was shown to yield β-normal forms. An alternative to reductions to normal forms is reductions to *weak head normal forms*, which is a relaxed notion of normal forms:

Weak head normal forms The following grammar defines the weak head normal forms in the λ-calculus: [5]

$$\begin{array}{lrcl} & a & ::= & x \mid a\ t \\ \textit{WHNForm} & w & ::= & a \mid \lambda x.t \end{array}$$

The normal forms are contained in the weak head normal forms. A comparison of the definition with that of normal forms in Section 1.2 pinpoints the differences: (1) All abstractions are weak head normal forms and not only if the body is in normal form. (2) An application is a weak head normal form if the left sub-term is either a variable or an application, in which the leftmost application has a variable as left sub-term — regardless of the right sub-terms on the path from the root to that application.

Weak head normalization Following this explanation and the very similar structure of the grammars of normal forms and weak head normal forms a reduction strategy to weak head normal forms is defined like the normal-order reduction strategy. The two differences from normal-order reduction mentioned above are ensured by not having two of the rules found in Section 1.5.1: The rule for reducing the body of abstractions and the rule for reducing the right-hand sub-term of applications. The definition of the obtained strategy $\mapsto_n$ reads as follows.

$$\frac{(t, t') \in \beta}{t \mapsto_n t'} \qquad \frac{t_1 \mapsto_n t_1'}{t_1\ t_2 \mapsto_n t_1'\ t_2}$$

By not having the rule for reducing abstractions, $(t_1\ t_2, t) \notin \beta$ is implied by $t_1 \mapsto_n t_1'$ and is therefore removed as a condition in the second rule. The implication together with β being a function ensures that $\mapsto_n$ is well-defined. Normalization to weak head normal form is accordingly defined as follows:

$$\textit{normalize}_n\ t = w \quad \text{iff} \quad t \mapsto_n^* w$$

This function is well-defined since $\mapsto_n$ is not defined on weak head normal forms, and it is partial, because it is not defined on terms with no weak head normal form.

[5] Considering only closed terms, the set of weak head normal forms simplifies to the set of abstractions.

No uniqueness of weak head normal forms In Section 1.4 the uniqueness of the normal form of a term was presented. An analogous result for weak head normal forms does not hold. The term $(\lambda x.x)\ (\lambda y.(\lambda z.y)\ w)$ is a counter example:

$$\begin{array}{rcl} (\lambda x.x)\ (\lambda y.(\lambda z.y)\ w) & \to_\beta & \lambda y.(\lambda z.y)\ w \\ & \to_\beta & \lambda y.y \end{array}$$

Both $\lambda y.(\lambda z.y)\ w$ and $\lambda y.y$ are weak head normal forms and equivalent to the first term in the λ-calculus; $\mathit{normalize}_n$ chooses among the weak head normal forms equivalent to the term the first one obtained when following the normal-order reduction strategy. Therefore

$$\mathit{normalize}_n\ ((\lambda x.x)\ (\lambda y.(\lambda z.y)\ w)) = \lambda y.(\lambda z.y)\ w$$

Following this explanation a weak version of Proposition 1 holds:[6]

Proposition 2
(i) $\mathit{normalize}_n\ t = w \Longrightarrow t =_\beta w$
(ii) $t =_\beta w \Longrightarrow \mathit{normalize}_n\ t = w'$, where $w' =_\beta w$

Part (ii) says that if a term t is equal to a weak head normal form w in the λ-calculus, normalization following the weak normal-order strategy yields a weak head normal form w' that is equal to w in the λ-calculus. This part of the proposition becomes more concrete in Section 2.8 where the notion of weak head normal forms is extended.

1.6 The λ-calculus with de Bruijn indices

Instead of a named variable the lexical offset relative to the variable's occurrence in the abstract syntax tree can serve as a term placeholder. In that case, substitution does not rely on free occurrences of variables but instead on a notion of the depths of sub-terms. For example, the term $\lambda x.x\ (\lambda y.x\ y)$ can be represented by $\lambda 1\ (\lambda 2\ 1)$ with de Bruijn indices starting from 1. Lexical offsets used this way are known as *de Bruijn indices* [29].[7] Abstractions do not bind variables explicitly, when using de Bruijn indices:

$$\begin{array}{lrcl} \mathit{Index} & i & ::= & \{1, 2, 3, \ldots\} \\ \mathit{Term}_{deB} & t & ::= & i \mid \lambda t \mid t\ t \end{array}$$

All *closed* terms in the same equivalence class of $=_\alpha$ (defined in Section 1.1.3) are represented by the same term using de Bruijn indices. For example, the terms $\lambda x.x\ (\lambda y.x\ y)$ and $\lambda y.y\ (\lambda x.y\ x)$ are both represented by $\lambda 1\ (\lambda 2\ 1)$ with de Bruijn indices starting from 1. Via a mapping ψ from named variables to the natural numbers this property holds for terms in general.

[6] Part (ii) of this proposition hinges on a lemma also used in the proof for the standardization theorem involving *standard reduction sequences*. We leave out the details.

[7] Another way to represent variables is *de Bruijn levels*. Instead of denoting an offset relative to the place of occurrence, a de Bruijn level is an offset relative to the root of the term on the path from the root to the occurrence. We do not treat de Bruijn levels in this text.

1.6.1 Correspondence with terms using named variables

The following translations between terms with named variables and terms with de Bruijn indices both use an auxiliary structure of named variables:

$$\textit{Bindings} \qquad b \quad ::= \quad \bullet \mid x \cdot b$$

In the rest of this text a list of objects is represented this way, and throughout $x_1 \cdot x_2 \cdots x_k$ abbreviates $x_1 \cdot (x_2 \cdot \ldots (x_k \cdot \bullet) \ldots)$.

From named variables to de Bruijn indices Translating a term with named variables to the corresponding term with de Bruijn indices assumes a function ψ from named variables to natural numbers. ψ can be defined via, e.g., a preorder traversal of the term.[8] ψ is assumed to be injective:

$$\begin{array}{rcll}
\textit{Varmap} \quad \psi & : & \textit{Var} \to \textit{Index} & \\
& & & \\
\texttt{toindices} & : & \textit{Term} \to \textit{Term}_{deB} & \\
\texttt{toindices}\, t & = & \texttt{aux}\,(t,\ \bullet) & \\
& & & \\
\texttt{aux} & : & \textit{Term} \times \textit{Bindings} \to \textit{Term}_{deB} & \\
\texttt{aux}\,(x,\ x_1 \cdots x_k) & = & i, & \text{if } \forall j < i : x_j \neq x \text{ and } x_i = x \\
\texttt{aux}\,(x,\ x_1 \cdots x_k) & = & k + \psi(x), & \text{if } \forall j \leq k : x_j \neq x \\
\texttt{aux}\,(\lambda x.t,\ b) & = & \lambda(\texttt{aux}\,(t,\ x \cdot b)) & \\
\texttt{aux}\,(t\,t',\ b) & = & (\texttt{aux}\,(t,\ b))\,(\texttt{aux}\,(t',\ b)) &
\end{array}$$

From de Bruijn indices to named variables Translating from de Bruijn-indexed λ-terms to terms with named variables assumes a mapping τ from indices to named variables. Like ψ, τ is assumed to be injective:

$$\begin{array}{rcll}
\textit{Indexmap} \quad \tau & : & \textit{Index} \to \textit{Var} & \\
& & & \\
\texttt{fromindices} & : & \textit{Term}_{deB} \to \textit{Term} & \\
\texttt{fromindices}\, t & = & \texttt{aux}\,(t,\ \bullet) & \\
& & & \\
\texttt{aux} & : & \textit{Term}_{deB} \times \textit{Bindings} \to \textit{Term} & \\
\texttt{aux}\,(i,\ x_1 \cdots x_k) & = & x_i, & \text{if } i \leq k \\
\texttt{aux}\,(i,\ x_1 \cdots x_k) & = & \tau(i - k), & \text{if } i > k \\
\texttt{aux}\,(\lambda t,\ b) & = & \lambda(\texttt{aux}\,(t,\ x \cdot b)), & x \text{ is fresh} \\
\texttt{aux}\,(t\,t',\ b) & = & (\texttt{aux}\,(t,\ b))\,(\texttt{aux}\,(t',\ b)) &
\end{array}$$

Here the image of *Index* under τ must be disjoint with the set of named variables introduced in the translation of abstractions. In other words, $\tau(i)$ is the i-th element (according to some ordering) in a set of variable names disjoint with the bound variables of the resulting term.

[8] When variables are strings over the letters $\{a, b, \ldots, z\}$, ψ can be defined via the standard lexicographic ordering on strings starting with $\psi\ (a) = 1$.

Mapping closed terms Mapping a *closed* term $\mathtt{t}$ with named variables to a term with de Bruijn indices and mapping back again yields a term with named variables that is equal to $\mathtt{t}$ in $=_\alpha$:

$$\mathtt{t} =_\alpha \mathtt{fromindices}\,(\mathtt{toindices}\,\mathtt{t})$$

These translations emphasize the independence of actual bound variables.

Mapping open terms The information about variable names is discarded under the translation to de Bruijn-indexed λ-terms — including the names of free variables. Therefore the above property does not hold for open terms.

Because of how `toindices` and `fromindices` uses ψ and τ another property of terms with free variables is established. When a variable occurs free in different places in a term, `toindices` maps the free occurrences to the same index and adjusts according to the *depth of occurrence*. `fromindices` maps frees de Bruijn indices relative to their *depth of occurrence*.

An example is illustrative. With ψ and τ defined via preorder traversals, the following holds.

$$\begin{aligned}
\mathtt{toindices}\,(\lambda y.(\lambda x.z\,x)\,w\,(\lambda y.(\lambda x.z\,x)\,w\,v)) &= (\lambda(\lambda 3\,1)\,3\,(\lambda(\lambda 4\,1)\,4\,5)) \\
\mathtt{toindices}\,(\lambda y.(\lambda x.w\,x)\,z\,(\lambda y.(\lambda x.w\,x)\,z\,v)) &= (\lambda(\lambda 3\,1)\,3\,(\lambda(\lambda 4\,1)\,4\,5)) \\
\mathtt{fromindices}\,(\lambda(\lambda 3\,1)\,3\,(\lambda(\lambda 4\,1)\,4\,5)) &= \lambda b_1.(\lambda b_2.f_1\,b_2)\,f_2\,(\lambda b_3.(\lambda b_4.f_1\,b_4)\,f_2\,f_3)
\end{aligned}$$

The initial terms and the term after the translation to indices and back again are not α-equivalent. But because the translation of free variables and indices are consistent, the α-equivalence could be extended (call it $=_{\hat{\alpha}}$) to cope with free variables.

A more feasible solution is to restrict τ further. Because ψ is injective it has a left inverse ψ^{-1}. Demanding τ to be this left inverse makes the property that holds for closed terms above hold for terms in general. Mapping an arbitrary term $\mathtt{t}$ to a representation with de Bruijn indices and back again yields a term α-equivalent to $\mathtt{t}$.

1.6.2 β-contraction on de Bruijn-indexed λ-terms

The notion of reduction β was in Section 1.2 introduced informally via a meta construction on terms with named variables. When using de Bruijn indices instead of named variables, this meta construction is not applicable as it relies on the possibility to use fresh variables. Instead, we explicitly define the meta construction (`substitute`) as a function on terms and use an auxiliary function (`reindex`):

$$\beta_{deB}: \quad (\lambda t)\,t' \;\rightarrow\; \mathtt{substitute}\,(t,\ (1, t'))$$

$$\begin{aligned}
\mathtt{substitute} &: \mathit{Term}_{deB} \times (\mathit{Index} \times \mathit{Term}_{deB}) \rightarrow \mathit{Term}_{deB} \\
\mathtt{substitute}\,(i,\ (j, t)) &= \begin{cases} i, & \text{if } i < j \\ \mathtt{reindex}\,(t,\ (1,\ i)) & \text{if } i = j \\ i - 1, & \text{if } i > j \end{cases} \\
\mathtt{substitute}\,(\lambda t,\ (j, t')) &= \lambda(\mathtt{substitute}\,(t,\ (j+1, t'))) \\
\mathtt{substitute}\,(t\,t',\ (j, t'')) &= (\mathtt{substitute}\,(t,\ (j, t'')))\,(\mathtt{substitute}\,(t',\ (j, t'')))
\end{aligned}$$

$$\begin{aligned}
\mathtt{reindex} &: \mathit{Term}_{deB} \times (\mathit{Index} \times \mathit{Index}) \rightarrow \mathit{Term}_{deB} \\
\mathtt{reindex}\,(i,\ (j, g)) &= \begin{cases} i, & \text{if } i < j \\ i + g - 1, & \text{if } i \geq j \end{cases} \\
\mathtt{reindex}\,(\lambda t,\ (j, g)) &= \lambda(\mathtt{reindex}\,(t,\ (j+1, g))) \\
\mathtt{reindex}\,(t\,t',\ (j, g)) &= (\mathtt{reindex}\,(t,\ (j, g)))\,(\mathtt{reindex}\,(t',\ (j, g)))
\end{aligned}$$

The substitution function differs only notationally from definitions found in standard texts (e.g., Hankin's textbook [38]).

The contraction rule β_{deB} states that the contractum of an application $(\lambda t)\ t'$ is t with de Bruijn index 1 mapping to t'. To realize the substitution, this mapping is distributed to the indices via the last two clauses of `substitute`. When distributing into the body of abstractions the single key (in the domain) of the mapping must be 'lifted'. After the first 'lift', 2 is mapped to t'.

Distributed to an index i, the mapping has in general been lifted a number of times, say (j, t'), where $j \geq 1$. If $i < j$, the mapping has been distributed into the abstraction that i refers to, and i is left untouched. If $i > j$, i is not bound in λt but because the number of abstractions on the path from i to the root of t is one less than to the root of λt, the index must be decremented yielding $i - 1$. If $i = j$ the mapping relates i to t' and an actual substitution is performed introducing t' instead of i. What is left to do is an adjustment of the free indices of t'. This reindexing is handled by `reindex` $(t', (1, i))$. The last two rules of `reindex` distribute to the indices of t'. If an index i' is reached by the mapping (j, i) two cases is needed. If $i' < j$, i' is bound in t' and is therefore left untouched. If $i' \geq j$, i' is not bound in t' and must be lifted such that it is not captured by an abstraction in t. That is, when substituting t' for index i in t, $i - 1$ abstractions have been entered and the free index i' must be lifted to $i' + i - 1$.

In the following we leave out the subscript and use β for β_{deB}, when it is clear from context that terms use de Bruijn indices.

Normal forms The normal forms is as defined in Section 1.2. The only difference is the use of de Bruijn indices instead of named variables. No β_{deB}-redexes are left in normal forms:

$$\begin{array}{rrcl} \mathit{ANForm}_{deB} & a & ::= & i \mid a\, n \\ \mathit{NForm}_{deB} & n & ::= & a \mid \lambda n \end{array}$$

1.6.3 Defining the λ-calculus for de Bruijn-indexed λ-terms

With one-step β_{deB}-reduction $\rightarrow_{\beta_{deB}}$, β_{deB}-reduction $\rightarrow^*_{\beta_{deB}}$, and β_{deB}-equality $=_{\beta_{deB}}$, for de Bruijn-indexed λ-terms defined exactly as for the λ-calculus when terms use named variables, the two specifications denote the same calculus:

$$t =_\beta t' \iff \mathtt{toindices}\, t =_{\beta_{deB}} \mathtt{toindices}\, t'$$

1.7 The λ-calculus defined as a proof system

In this chapter, the λ-calculus has been defined in terms of relations on λ-terms by the notion of reduction β, a compatibility closure $\rightarrow_\beta$, and a reflexive transitive symmetric closure $=_\beta$.

Equivalently, the λ-calculus can be defined as a proof system with axioms and inference rules. Axioms defines a basic set of facts, and inference rules are used to combine facts, inductively. Axioms constitute the notion of reduction, and inference rules constitute the compatibility rules and the reflexive transitive symmetric closure. Barendregt introduces the λ-calculus this way [7]. Equations in the λ-calculus are then usually written $\lambda \vdash t = t'$.

The two different definitions of the λ-calculus are equivalent in the sense that $t =_\beta t' \iff \lambda \vdash t = t'$ [7, Proposition 3.2.1].

In this chapter the λ-calculus has not been defined as a proof system, because the reduction-based version directly relies on term transformations via term reductions. The use of the relations $\rightarrow^*_\beta$ and $\mapsto^*_n$ emphasizes a direction on the transformations: The most substantial parts of this text concentrate on the process of obtaining possible normal forms of terms and not on proving equivalences between terms in general.

1.8 Summary

In this chapter, we introduced the λ-calculus: an abstract term language together with relations on terms. The basic relation on terms is the notion of reduction β. Allowing β-reductions in any subpart of terms defines how to change one term to an equivalent term in the calculus. We explained that any term is equal to at most one term in which there is no possible ways to use β, i.e., a β-normal form.

Normal-order reduction was introduced as a strategy on how to perform the β-reductions. This strategy yields a normal form of a term if and only if the term and the normal form are equal in the λ-calculus. We introduced a relaxed notion of normal forms — weak head normal forms — and a strategy to obtain relaxed normal forms.

We introduced λ-terms using de Bruijn indices instead of named variables and we defined consistent mappings (also on open terms via ψ and ψ^{-1}) between the two sets of terms. We explicitly defined β_{deB}, the version of β operating on de Bruijn-indexed λ-terms and justified that this set of terms and the notion of reduction β_{deB} also defines the λ-calculus.

We finally sketched how the calculus can be defined as a proof system with β denoting the basic set of facts.

Chapter 2

Definability in the λ-calculus

The terms in the λ-calculus as defined in Chapter 1 indeed only contain variables, abstractions and applications. On the one hand, *primitive values* like integers and booleans and functions on such primitives are not natively supported. Data structures like tuples and lists are also not directly supported. On the other hand, the λ-calculus is Turing-complete, i.e., the λ-calculus is as expressive as other sound formalisms like the Turing-machine [46]: It is somehow possible to represent by terms in the calculus, the above mentioned mathematical elements like integers and lists and represent functions on such elements.

Lambda definability concerns representations of elements and functions on elements in the λ-calculus. Since Church introduced the λ-calculus, various representations of such elements and functions has been proposed. Church himself suggested, e.g., representations of the natural numbers, called *numerals*, and functions operating on these *Church numerals*, together with data structures like pairs and lists. Also Rosser and Kleene suggested various functions on Church numerals.

Roadmap Lambda representations are presented in various textbooks and papers. Often these representations are specified with little motivation, if any. In this chapter, we sketch how representations can be built up incrementally. To this end, we present selected representations starting as traditional with Church numerals and some simple functions on them exploiting the computational power built into Church numerals (Section 2.1). We introduce simple structured elements (Section 2.2.1) and present Kleene's predecessor function (Section 2.2.2) and the iterative factorial function presented by Goldberg and Reynolds (Section 2.2.4).

It is our observation that the pattern of Kleene's predecessor function and the iterative factorial function is that of dynamic programming [37, Section 12.3]. In Section 2.3, we exemplify this observation and we λ-define the function finding the length of a longest common subsequence of two sequences of numerals.

In Section 2.4, we introduce *Scott numerals*, which do not have built-in computational power. A simple observation lets us introduce *streams*, which enable us to represent functions on Scott numerals in a similar way as for the Church numerals, and we present a general relationship between Church numerals and Scott numerals (Section 2.5) and a course-of-value representation [70, Chapter 3] underlying both numeral systems. This representation lets us generalize Kleene's predecessor function (Section 2.6) and we present a new version of the subtraction function *monus* on Church numerals, which is linear whereas the usual one iterating the predecessor function is quadratic.

In Section 2.7, we review functions relying on *monus* and we present a new efficient version of the quotient and remainder functions exploiting the course-of-value representation. Finally in Section 2.8, we extend the λ-calculus with basic constants, which will become useful in later chapters.

2.1 Church numerals

As described above, in particular no primitive zero value or primitive successor function is directly supported by built-in term constructors. Church numerals are β-normal forms abstracting over a successor function and a zero value and represent numbers in the style of Peano [11]:[1]

$$\begin{array}{rcll}
\ulcorner 0 \urcorner_c & := & \lambda s.\lambda z.z & \\
\ulcorner 1 \urcorner_c & := & \lambda s.\lambda z.s\ z & =_\beta \quad \lambda s.\lambda z.s\ (\ulcorner 0 \urcorner_c\ s\ z) \\
\ulcorner 2 \urcorner_c & := & \lambda s.\lambda z.s\ (s\ z) & =_\beta \quad \lambda s.\lambda z.s\ (\ulcorner 1 \urcorner_c\ s\ z) \\
& \vdots & & \\
\ulcorner n+1 \urcorner_c & := & \lambda s.\lambda z.s\ (\overbrace{s\ (\ldots (s}^{n\ \text{times}}\ z) \ldots)) & =_\beta \quad \lambda s.\lambda z.s\ (\ulcorner n \urcorner_c\ s\ z)
\end{array}$$

When applied to a successor function and a zero value these arguments are used to generate the represented natural number. But a Church numeral can of course be used differently: $\ulcorner n \urcorner_c\ f$ is an abstraction that, when applied to an argument x, applies f iteratively n times starting from x, i.e., a bounded iteration. Church numerals thus have a built-in computational power.

The successor function A term to represent the successor function on Church numerals $\ulcorner add_1 \urcorner_c$ must, when applied to the representation of n, be equivalent to the representation of $n+1$:

$$\begin{array}{rcl}
\ulcorner add_1 \urcorner_c\ \ulcorner n \urcorner_c & \overset{?}{=}_\beta & \ulcorner n+1 \urcorner_c \\
& =_\beta & \lambda s.\lambda z.s\ (\ulcorner n \urcorner_c\ s\ z)
\end{array}$$

Hence $\ulcorner add_1 \urcorner_c$ just abstracts over the argument numeral:

$$\ulcorner add_1 \urcorner_c \quad := \quad \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$$

Intuitively, $\ulcorner add_1 \urcorner_c$ constructs a numeral that, when supplied with a successor function and a zero value, generates the number represented by n and takes the successor of that number.[2] The successor function on natural numbers is said to be λ-*definable* (on Church numerals).

The addition function We define a term representing the addition function on Church numerals that exploits the built-in iterative power of the Church numerals. Especially,

$$\begin{array}{rcl}
\ulcorner n \urcorner_c\ \ulcorner add_1 \urcorner_c\ \ulcorner n' \urcorner_c & =_\beta & (\lambda n'.\overbrace{\ulcorner add_1 \urcorner_c\ (\ldots (\ulcorner add_1 \urcorner_c}^{n\ \text{times}}\ n') \ldots))\ \ulcorner n' \urcorner \\
& =_\beta & \ulcorner n + n' \urcorner_c
\end{array}$$

[1] We use $\ulcorner \cdot \urcorner$ throughout to denote *representation*. We use $\ulcorner \cdot \urcorner_c$ to denote Church numerals.

[2] An alternative definition is $\ulcorner add_1 \urcorner_c := \lambda n.\lambda s.\lambda z.n\ s\ (s\ z)$. The intuition is here that the number 1 is constructed and used as the zero of the resulting numeral.

That is, the addition function on Church numerals abstracts over the two numerals and is a bounded iteration of the successor function:

$$\ulcorner add \urcorner_c \quad := \quad \lambda n.\lambda n'.n \ulcorner add_1 \urcorner_c\, n'$$

The multiplication function Specializing addition to one argument numeral $\ulcorner n \urcorner_c$ yields an abstraction that adds n to its argument numeral: $\ulcorner add_n \urcorner_c := \lambda n'.n \ulcorner add_1 \urcorner_c\, n'$. Iteratively applying that abstraction n' times yields an abstraction that adds $n \times n'$ to its argument numeral:

$$\begin{array}{rcl} \ulcorner n' \urcorner_c\, (\ulcorner add \urcorner_c \ulcorner n \urcorner_c) & =_\beta & \lambda z.\overbrace{\ulcorner add_n \urcorner_c\, (\ldots (\ulcorner add_n \urcorner_c}^{n' \text{ times}}\, z) \ldots) \\ & =_\beta & \ulcorner add \urcorner_c \ulcorner n \times n' \urcorner_c \end{array}$$

Because $\ulcorner add \urcorner_c \ulcorner n \times n' \urcorner_c \ulcorner 0 \urcorner_c =_\beta \ulcorner n \times n' \urcorner_c$ a definition of the multiplication function on Church numerals finally just abstracts over the two numerals:

$$\ulcorner mult \urcorner_c \quad := \quad \lambda n.\lambda n'.n'\, (\ulcorner add \urcorner_c\, n)\, \ulcorner 0 \urcorner_c$$

The exponentiation function Addition was defined as bounded iteration of the simpler successor function. Likewise multiplication was defined as bounded iteration of the simpler addition function. This line of thought lets us define the exponentiation function on Church numerals as well as bounded iteration of the simpler multiplication function (We leave out the details):

$$\ulcorner exp \urcorner_c \quad := \quad \lambda n.\lambda n'.n'\, (\ulcorner mult \urcorner_c\, n)\, \ulcorner 1 \urcorner_c$$

2.2 Representing structured elements

As explained in the previous section a Church numeral represents a bounded iteration. The iteration implicitly generates a series of 'intermediate results': For $\ulcorner n \urcorner_c\, f\, a_0$ the series $a_0, \ldots a_n$ is generated with $a_{i+1} = f\, a_i$. The result is a_n. All definitions in the previous section only iterate with a_i being Church numerals, i.e., the series of intermediate results consists of *unstructured* elements. Having *structured* elements in the series gives more possible use.

2.2.1 Ordered pairs

The most simple structured element is the 2-tuple data structure: an ordered pair of elements. The operations for this 'domain' consist of one injection and two projections. With mutually consistent definitions of these three functions any representation will do: The definitions of the injection $\ulcorner pair \urcorner$ and the two projections for extracting the first element $\ulcorner \pi_1 \urcorner$ and the second element $\ulcorner \pi_2 \urcorner$ respectively, must ensure:

$$\begin{array}{rcl} \ulcorner \pi_1 \urcorner\, (\ulcorner pair \urcorner\, t_1\, t_2) & =_\beta & t_1 \\ \ulcorner \pi_2 \urcorner\, (\ulcorner pair \urcorner\, t_1\, t_2) & =_\beta & t_2 \end{array}$$

The injection function takes two elements (one at a time) and gives an abstraction that applies the argument to the two elements:

$$\ulcorner pair \urcorner \quad := \quad \lambda a.\lambda d.\lambda s.s\ a\ d$$

The idea is to let the argument of the pair-abstraction be a selector: $\lambda a.\lambda d.a$ or $\lambda a.\lambda d.d$. Each of the two projections takes a pair and chooses the right selector to which the pair is applied. According to the definition of $\ulcorner pair \urcorner$ the selector is then applied to the two elements:

$$\begin{array}{rcl} \ulcorner \pi_1 \urcorner & := & \lambda p.p\ (\lambda a.\lambda d.a) \\ \ulcorner \pi_2 \urcorner & := & \lambda p.p\ (\lambda a.\lambda d.d) \end{array}$$

In the following $\langle a, d \rangle$ denotes an ordered pair with elements a and d: Taking the first projection and taking the second projection on this pair yields a and d, respectively. In other words, $\langle a, d \rangle := \ulcorner pair \urcorner\ a\ d$.

2.2.2 Kleene's predecessor function and the corresponding subtraction function

A representation of the predecessor function on Church numerals was not defined in Section 2.1. Via the successor function a series of elements where the n-th element is $\ulcorner n-1 \urcorner_c$ given $\ulcorner -1 \urcorner_c$ can easily be defined. However, $\ulcorner -1 \urcorner_c$ is not defined.

A solution is achieved by using structured elements within the bounded iteration. Using pairs it is possible to 'piggy-bag' one extra numeral along with the 'current' numeral in each element in the generated series:

$$\begin{array}{c} \langle \ulcorner 0 \urcorner_c, \ulcorner 0 \urcorner_c \rangle \\ \langle \ulcorner 1 \urcorner_c, \ulcorner 0 \urcorner_c \rangle \\ \langle \ulcorner 2 \urcorner_c, \ulcorner 1 \urcorner_c \rangle \\ \langle \ulcorner 3 \urcorner_c, \ulcorner 2 \urcorner_c \rangle \\ \vdots \\ \langle \ulcorner n \urcorner_c, \ulcorner n-1 \urcorner_c \rangle \end{array}$$

Each pair p_i can be constructed from the previous pair p_{i-1} by taking as first component $\ulcorner add_1 \urcorner_c\ (\ulcorner \pi_1 \urcorner\ p_{i-1})$ and as second component just $\ulcorner \pi_1 \urcorner\ p_{i-1}$. An abstraction f achieving $f\,p_{i-1} =_\beta p_i$ is then $\lambda p.\langle \ulcorner add_1 \urcorner_c\ (\ulcorner \pi_1 \urcorner\ p), \ulcorner \pi_1 \urcorner\ p \rangle$. The second projection of the pair obtained as a bounded iteration of f via $\ulcorner n \urcorner_c$ starting from $\langle \ulcorner 0 \urcorner_c, \ulcorner 0 \urcorner_c \rangle$ is the predecessor of $\ulcorner n \urcorner_c$:

$$\ulcorner sub_1 \urcorner_c \quad := \quad \lambda n.\ulcorner \pi_2 \urcorner\ (n\ (\lambda p.\langle \ulcorner add_1 \urcorner_c\ (\ulcorner \pi_1 \urcorner\ p), \ulcorner \pi_1 \urcorner\ p \rangle)\ \langle \ulcorner 0 \urcorner_c, \ulcorner 0 \urcorner_c \rangle)$$

This definition of the predecessor for Church numerals is due to Kleene [44].[3] By analogy with addition, subtraction is defined by iterating the simpler predecessor function:

$$\ulcorner sub \urcorner_c \quad := \quad \lambda n.\lambda n'.n'\ \ulcorner sub_1 \urcorner_c\ n$$

Because $\ulcorner sub_1 \urcorner_c\ \ulcorner 0 \urcorner_c =_\beta \ulcorner 0 \urcorner_c$,[4]

$$\ulcorner sub \urcorner_c\ \ulcorner n \urcorner_c\ \ulcorner n' \urcorner_c \quad =_\beta \quad \begin{cases} \ulcorner n - n' \urcorner_c & \text{if } n \geq n' \\ \ulcorner 0 \urcorner_c & \text{if } n < n' \end{cases}$$

The subtraction function with this property is also know as *monus*.

[3] Kleene's original version is defined in the $\lambda\mathbf{I}$-calculus, which was briefly introduced in Section 1.1.2. In that calculus $\ulcorner 0 \urcorner_c$ is not present so the predecessor is defined such that $\ulcorner sub_1 \urcorner_c\ \ulcorner 1 \urcorner_c =_\beta \ulcorner 1 \urcorner_c$. Following the same line of thought as in our definition of the predecessor it is seen that a triple must be used instead of a pair.

[4] Indeed, applying $\ulcorner sub_1 \urcorner_c\ n'$ times to $\ulcorner n \urcorner_c$ subtracts $\ulcorner 1 \urcorner_c$ each time if $n > n'$. If $n \leq n'$, $\ulcorner 1 \urcorner_c$ is subtracted until $\ulcorner 0 \urcorner_c$ is reached. At that point it is fixed at $\ulcorner 0 \urcorner_c$ for the rest of the iterations.

2.2.3 Boolean values and functions on booleans

Representing boolean values and negation is simple:

$$
\begin{array}{rcl}
\ulcorner \mathit{true} \urcorner & := & \lambda x.\lambda y.x \\
\ulcorner \mathit{false} \urcorner & := & \lambda x.\lambda y.y \\
\ulcorner \neg \urcorner & := & \lambda b.b\, \ulcorner \mathit{false} \urcorner\, \ulcorner \mathit{true} \urcorner
\end{array}
$$

Here $\ulcorner \mathit{true} \urcorner$ and $\ulcorner \mathit{false} \urcorner$ are normal forms and

$$
\begin{array}{rcl}
\ulcorner \neg \urcorner\, \ulcorner \mathit{true} \urcorner & =_\beta & \ulcorner \mathit{false} \urcorner \\
\ulcorner \neg \urcorner\, \ulcorner \mathit{false} \urcorner & =_\beta & \ulcorner \mathit{true} \urcorner
\end{array}
$$

Many functions on numerals yielding boolean values are directly definable via the monus function:

$$
\begin{array}{rcl}
\ulcorner \mathit{zero?} \urcorner_c & := & \lambda n.n\, (\lambda x. \ulcorner \mathit{false} \urcorner)\, \ulcorner \mathit{true} \urcorner \\
\ulcorner \leqslant \urcorner_c & := & \lambda n.\lambda n'. \ulcorner \mathit{zero?} \urcorner_c\, (\ulcorner \mathit{sub} \urcorner_c\, n\, n') \\
\ulcorner \geq \urcorner_c & := & \lambda n.\lambda n'. \ulcorner \mathit{zero?} \urcorner_c\, (\ulcorner \mathit{sub} \urcorner_c\, n'\, n) \\
\ulcorner > \urcorner_c & := & \lambda n.\lambda n'. \ulcorner \neg \urcorner\, (\ulcorner \leqslant \urcorner_c\, n\, n') \\
\ulcorner < \urcorner_c & := & \lambda n.\lambda n'. \ulcorner \neg \urcorner\, (\ulcorner \geq \urcorner_c\, n\, n') \\
\ulcorner = \urcorner_c & := & \lambda n.\lambda n'.(\ulcorner \leqslant \urcorner_c\, n\, n')\, (\ulcorner \geq \urcorner_c\, n\, n')\, \ulcorner \mathit{false} \urcorner
\end{array}
$$

These representations all inherit the ‘efficiency’ or ‘inefficiency’ of the definition of $\ulcorner \mathit{sub} \urcorner_c$.

2.2.4 The factorial function

The use of pairs as elements in the bounded iteration applies directly in representations of other functions. An iterative representation of the factorial function on Church numerals, presented by Goldberg [36] and Reynolds [64], also uses pairs in the generated series of elements:

$$
\begin{array}{c}
\langle \ulcorner 1 \urcorner_c, \ulcorner 1 \urcorner_c \rangle \\
\langle \ulcorner 2 \urcorner_c, \ulcorner 1 \urcorner_c \rangle \\
\langle \ulcorner 3 \urcorner_c, \ulcorner 2 \urcorner_c \rangle \\
\langle \ulcorner 4 \urcorner_c, \ulcorner 6 \urcorner_c \rangle \\
\langle \ulcorner 5 \urcorner_c, \ulcorner 24 \urcorner_c \rangle \\
\vdots \\
\langle \ulcorner n+1 \urcorner_c, \ulcorner n! \urcorner_c \rangle
\end{array}
$$

An abstraction $\mathit{it}_{\mathit{fac}}$ achieving $\mathit{it}_{\mathit{fac}} p_{i-1} =_\beta p_i$ is $\lambda p.\langle \ulcorner \mathit{add}_1 \urcorner_c\, (\ulcorner \pi_1 \urcorner\, p), \ulcorner \mathit{mult} \urcorner_c\, (\ulcorner \pi_1 \urcorner\, p)\, (\ulcorner \pi_2 \urcorner\, p) \rangle$ The second projection of the pair obtained by bounded iteration of $\mathit{it}_{\mathit{fac}}$ via $\ulcorner n \urcorner_c$ starting from $\langle \ulcorner 1 \urcorner_c, \ulcorner 1 \urcorner_c \rangle$ is then $\ulcorner n! \urcorner_c$ (witness the first component of the initial element is $\ulcorner 1 \urcorner_c$ and not $\ulcorner 0 \urcorner_c$):

$$
\ulcorner \mathit{fac} \urcorner_c \quad := \quad \lambda n. \ulcorner \pi_2 \urcorner\, (n\, \mathit{it}_{\mathit{fac}}\, \langle \ulcorner 1 \urcorner_c, \ulcorner 1 \urcorner_c \rangle)
$$

The factorial function is simple in nature and is easily defined in a recursive manner in a mathematical setting. The factorial function is therefore the canonical example used in presentations of *fixed-point combinators* in standard textbooks [65, page 95] [38, page 82] [47, page 89]. Fixed-point combinators are needed to represent *recursive functions* (in the sense of computability). However, the factorial function is only *primitive recursive* (in the sense of computability) as demonstrated by the iterative representation above: $\ulcorner \mathit{fac} \urcorner_c$ is ‘composed’ of other primitive recursive functions.

2.2.5 Representing lists

Church lists

Analogously with Church numerals (which abstract over a successor function and a zero value) representations of lists (i.e., finite sequences) can abstract over a prepend operation *cons*, and the empty-list, *nil*. Via such *Church lists* a list $[a_0, a_1, \ldots, a_{j-1}]$ is represented by the normal form $\lambda c.\lambda n.c\, a_0\, (c\, a_1\, (\ldots (c\, a_{j-1}\, n)\ \ldots))$. Analogously with the successor function on Church numerals *cons*'ing and element to a Church list $\ulcorner cons \urcorner_p$ is a constant operation and analogously with the predecessor function on Church numerals taking the tail of a Church list $\ulcorner tl \urcorner_c$ is a linear operation. Following the line of thought leading to Kleene's predecessor function in Section 2.2.2 lets us represent the tail function on Church lists. We note that taking the head of a Church list $\ulcorner hd \urcorner_c$ is also a linear operation. We observe that a Church list represents the right-folding over the represented list and hence has build-in computational power corresponding to the represented list. We leave this representation of lists.

Lists as nested pairs

Alternatively, lists can be represented as properly nested pairs: In that case the above list $[a_0, a_1, \ldots, a_{j-1}]$ can be represented by $\langle a_0, \langle a_1, \ldots \langle a_{j-1}, \lambda s.\ulcorner true \urcorner \rangle \ldots \rangle \rangle$. The representations of the constructor prepending an element to a list $\ulcorner cons \urcorner_p$ and the destructor operations for taking the head $\ulcorner hd \urcorner_p$ and taking the tail $\ulcorner tl \urcorner_p$ of a non-empty list are immediate:

$$\begin{array}{rcl} \ulcorner nil \urcorner_p & := & \lambda s.\ulcorner true \urcorner \\ \ulcorner cons \urcorner_p & := & \ulcorner pair \urcorner \\ \ulcorner hd \urcorner_p & := & \ulcorner \pi_1 \urcorner \\ \ulcorner tl \urcorner_p & := & \ulcorner \pi_2 \urcorner \end{array}$$

The representation of the empty list $\lambda s.\ulcorner true \urcorner$ is constructed such that it is easy to represent the predicate indicating if the list is empty: $\ulcorner nil? \urcorner_p := \lambda l.l\ (\lambda a.\lambda d.\ulcorner false \urcorner)$. Every non-empty list is a pair which expects a selector and applies this selector to its first element a and its second element d. The result is in that case $\ulcorner false \urcorner$. If the list is empty applying the list to any selector (including $\lambda a.\lambda d.\ulcorner false \urcorner$) yields $\ulcorner true \urcorner$. In the following we use subscript p, when lists are represented via pairs in this way: $[a_0, a_1, \ldots, a_{j-1}]_p$

Taking the n-th element of a list With lists as the elements in the series generated by bounded iteration some standard functions on lists are immediate. We obtain the function yielding the n-th element of a list (when it contains at least $n+1$ elements) by iterating the tail function (i.e., taking the n-th tail) and taking the head of the resulting list:

$$\ulcorner nth \urcorner_p \quad := \quad \lambda n.\lambda l.\ulcorner hd \urcorner_p\ (n\ \ulcorner tl \urcorner_p\ l)$$

Using the operations for Church lists $\ulcorner hd \urcorner_c$ and $\ulcorner tl \urcorner_c$ instead, we have a representation of taking the n-th element of a Church lists, i.e., $\ulcorner nth \urcorner_c$. Taking the n-th tail of a Church list corresponds to subtraction on Church numerals. Defining $\ulcorner nth \urcorner_c$ in this way iterating $\ulcorner tl \urcorner_c$ is hence inefficient in the same way that subtraction on Church numerals is inefficient when represented via iteration of the predecessor function.

Bounded reversal of a list The ability to take the first n elements of a list l and give a list with the elements in reversed order is a bit more complicated but follows the same line of thought: In each element an accumulator is stored together with the current tail of the list l. In each iteration the head of the tail is added to the accumulator list. With $\ulcorner nil \urcorner_p$ as the initial accumulator, the accumulator is the first n elements of the list in reversed order:

$$\begin{array}{rcl} it_{revn} & := & \lambda p.\langle \ulcorner tl \urcorner_p\ (\ulcorner \pi_1 \urcorner\ p), \ulcorner cons \urcorner_p\ (\ulcorner hd \urcorner_p\ (\ulcorner \pi_1 \urcorner\ p))\ (\ulcorner \pi_2 \urcorner\ p)\rangle \\ \ulcorner revn \urcorner_p & := & \lambda n.\lambda l.\ulcorner \pi_2 \urcorner\ (n\ it_{revn}\ \langle l, \ulcorner nil \urcorner_p \rangle) \end{array}$$

Augmenting the pair representation of lists

To reverse a complete list l the length of l must be known by the above definition. Considering Church lists the length of a list can be found by applying the list to a prepending operator representing the successor function and as nil value the zero value. (This representation is the standard exercise of finding the length of a list by folding over the list.) Considering lists represented via nested pairs, finding the length of a list can not be achieved via bounded iteration — this list representation does not have build-in computational power corresponding to the represented list.

A possibility is, to augment such a list represented by nested pairs with its length: Here $[a_0,\ a_1,\ \ldots,\ a_{j-1}]_a$ abbreviates $\langle \ulcorner j \urcorner_c, \langle a_0, \langle a_1, \ldots \langle a_{j-1}, \lambda s.\ulcorner true \urcorner \rangle \ldots \rangle\rangle\rangle$. The representations of *nil*, *cons*, etc. are easily defined using the previous representations:

$$\begin{array}{rcl} \ulcorner nil \urcorner_a & := & \langle \ulcorner 0 \urcorner_c, \ulcorner nil \urcorner_p \rangle \\ \ulcorner cons \urcorner_a & := & \lambda a.\lambda l.\langle \ulcorner add_1 \urcorner_c\ (\ulcorner \pi_1 \urcorner\ l), \ulcorner cons \urcorner_p\ a\ (\ulcorner \pi_2 \urcorner\ l)\rangle \\ \ulcorner hd \urcorner_a & := & \lambda l.\ulcorner hd \urcorner_p\ (\ulcorner \pi_2 \urcorner\ l) \\ \ulcorner tl \urcorner_a & := & \lambda l.\langle \ulcorner sub_1 \urcorner_c\ (\ulcorner \pi_1 \urcorner\ l), \ulcorner tl \urcorner_p\ (\ulcorner \pi_2 \urcorner\ l)\rangle \\ \ulcorner nil? \urcorner_a & := & \lambda l.\ulcorner nil? \urcorner_p\ (\ulcorner \pi_2 \urcorner\ l) \end{array}$$

Finding the length of a list is now trivial:

$$\ulcorner len \urcorner_a \quad := \quad \ulcorner \pi_1 \urcorner$$

Reversing an augmented pair-represented list uses $\ulcorner revn \urcorner_c$ as an auxiliary function. From the resulting pair-represented list we construct the corresponding augmented list:

$$\ulcorner rev \urcorner_a \quad := \quad \lambda l.\langle \ulcorner len \urcorner_a\ l, \ulcorner revn \urcorner_c\ (\ulcorner len \urcorner_a\ l)\ (\ulcorner \pi_2 \urcorner\ l)\rangle$$

When representations has subscript a, we assume the above revised functions are used, e.g., $\ulcorner nth \urcorner_a$.

2.3 Dynamic programming

Kleene's predecessor function and the iterative factorial function on Church numerals as presented in Section 2.2 consist of (i) *construction* of an initial pair of numerals, (ii) *bounded iteration* of an auxiliary function over the pair, and (iii) *extraction* of the final answer from the resulting pair. In a functional setting the functions operate, via bounded iteration, on a pair in a store-like manner. The iterative nature is carried out by letting the iterated function consume and produce a pair.[5] Kleene's predecessor function and the iterative factorial function hence follow a general pattern known as *dynamic programming* [37, Section 12.3]:

[5]The pair of numerals is *threaded*.

(i) Define the structure of configurations and construct an initial configuration.

(ii) Specify the next configuration as a function of the current configuration in a bounded number of iterations.

(iii) Extract the result from the final configuration.

This approach to define algorithms is usually applied in connection with optimization problems, where a naive solution is slow (relative to a solution with optimal time complexity). Usually, to be useful, dynamic programming requires that an optimal solution can be described in terms of optimal solutions for subproblems, and that the same subproblems occur more times in different parts of the solution. This description also matches the predecessor function and the factorial function except that subproblems do not occur more times in different part of the solution. The solutions to subproblems are 'optimal' per definition:

$$\begin{aligned} \mathit{sub}_1\,(n) &= \begin{cases} 0, & \text{if } n = 0, 1 \\ 1 + \mathit{sub}_1\,(n-1) & \text{if } n \geq 2 \end{cases} \\ \mathit{fac}\,(n) &= \begin{cases} 1, & \text{if } n = 0 \\ n \times \mathit{fac}\,(n-1) & \text{if } n \geq 1 \end{cases} \end{aligned}$$

In the function mapping natural numbers to the corresponding Fibonacci-number subproblems do occur more times in the solution:

$$\mathit{fib}\,(n) = \begin{cases} 1, & \text{if } n = 1, 2 \\ \mathit{fib}\,(n-1) + \mathit{fib}\,(n-2), & \text{if } n \geq 3 \end{cases}$$

In fact, we encounter an exponential blowup in the number of subproblems to solve in a naive implementation of such a specification. In the following, a more realistic problem is presented.

2.3.1 A longest common subsequence

The problem of finding a longest common subsequence (not subsegment!) *LCS* of two sequences a and b fits perfectly into the problem area of dynamic programming. *LCS* is an optimization problem where the result can be defined as a combination of optimal solutions to subproblems. Also subproblems occur in more parts of the solution. It is a standard exercise to give an efficient algorithm by use of dynamic programming:

With sequences $a = [a_1, \ldots, a_n]$ and $b = [b_1, \ldots, b_{n'}]$ the base structure of configurations is a matrix $M_{(n+1)\times(n'+1)}$. Entry $M_{i,j}$ represents the length of a longest common subsequence of $[a_1, \ldots, a_i]$ and $[b_1, \ldots, b_j]$. The length of a longest common subsequence using the empty prefix of a will always be 0. That is, all entries in the first *row* is 0. Analogously with use of the empty prefix of b all entries in the first *column* is 0.

A relation between solutions indicates that each entry can be defined in terms of optimal solutions to subproblems, and that subproblems occur more times when unfolding:

$$|\,\mathit{LCS}\,([a_1, \ldots, a_i], [b_1, \ldots, b_j])\,| = \begin{cases} 1 + |\,\mathit{LCS}\,([a_1, \ldots, a_{i-1}], [b_1, \ldots, b_{j-1}])\,| & \text{if } a_i = b_j \\ \mathit{max}\,\{|\,\mathit{LCS}\,([a_1, \ldots, a_{i-1}], [b_1, \ldots, b_j])\,|, & \\ \qquad |\,\mathit{LCS}\,([a_1, \ldots, a_i], [b_1, \ldots, b_{j-1}])\,|\} & \text{if } a_i \neq b_j \end{cases}$$

That is, each entry $M_{i,j}$ can be found directly from the entries $M_{i-1,j-1}$, $M_{i-1,j}$, and $M_{i,j-1}$. The complete definition of all entries hence reads:

$$M_{i,j} \quad = \quad \begin{cases} 0 & \text{if } i = 0 \\ 0 & \text{if } j = 0 \\ 1 + M_{i-1,j-1} & \text{if } a_i = b_j,\, i,j \geq 1 \\ \mathit{max}\,\{M_{i-1,j},\ M_{i,j-1}\} & \text{if } a_i \neq b_j,\, i,j \geq 1 \end{cases}$$

Obviously, the matrix can be filled out, e.g., row by row, starting in entry $M_{1,1}$. The length of a longest common subsequence of a and b is $M_{n,n'}$, and one actual common subsequence of this length can be found by a 'backwards traversal' in the matrix starting in entry $M_{n,n'}$.

The problem nicely fits the pattern of dynamic programming: (1) Defining the base structure matrix M and constructing the initial configuration by filling out first row and first column with 0, (2) iteratively filling out the rest of the matrix, and finally (3) constructing the result by a traversal in the completely filled out matrix together constitute a solution to the problem.

2.3.2 Length of a longest common subsequence

To simplify the definitions in this section we restrict the problem to only find the *length* of a longest common subsequence *LLCS*.

Base structure and initial configuration A simple observation from the above analysis is that filling out row i does not depend on other rows than row i and row $i-1$. Because we only consider finding the length and not an actual longest common subsequence the maintained structure is simplified such that only the *last* row and not the complete matrix is maintained when building the *current* row. In addition, (parts of) the sequences a and b are also maintained because they are consumed in each iteration. Accordingly, we represent configuration by a list of length 4:

$$[\mathit{last\ row},\ \mathit{current\ row},\ a\ \mathit{suffix},\ b\ \mathit{suffix}]_a$$

For readability we define four 'projections':

$$\begin{array}{rcl} \mathit{row}_{\mathit{last}} & := & \lambda c.\ulcorner \mathit{nth} \urcorner_a \ulcorner 0 \urcorner_c\, c \\ \mathit{row}_{\mathit{cur}} & := & \lambda c.\ulcorner \mathit{nth} \urcorner_a \ulcorner 1 \urcorner_c\, c \\ \mathit{suffix}_a & := & \lambda c.\ulcorner \mathit{nth} \urcorner_a \ulcorner 2 \urcorner_c\, c \\ \mathit{suffix}_b & := & \lambda c.\ulcorner \mathit{nth} \urcorner_a \ulcorner 3 \urcorner_c\, c \end{array}$$

The initial configuration is constructed by the following abstraction when applied to the first sequence a and the length n' of the second sequence:

$$\mathit{conf}_{\mathit{start}} \quad := \quad \lambda a.\lambda n'.[\ulcorner \mathit{nil} \urcorner_a,\ \ulcorner \mathit{add}_1 \urcorner_c\, n'\, (\ulcorner \mathit{cons} \urcorner_a \ulcorner 0 \urcorner_c)\, \ulcorner \mathit{nil} \urcorner_a,\ \ulcorner \mathit{cons} \urcorner_a\, (\lambda x.x)\, a,\ \ulcorner \mathit{nil} \urcorner_a]_a$$

Here the term for the current row represents a list of $n'+1$ zeros. Dummies are used in the other three components of the initial configuration because we perform a reconfiguration *before* each row is handled.

Reconfiguration before each new row Reconfiguration is done before calculating each row: Current row becomes last row, and we reverse that new last row to prepare it for a left to right traversal. The new current row is just the list $[\ulcorner 0 \urcorner_c]_a$ which is the base case for a row. Finally, we progress one step in the sequence a and reinstall b. We define reconfiguration as an abstraction over the configuration c, and the sequence b:

$$\textit{reconf} \quad := \quad \lambda c.\lambda b.[\ulcorner \textit{rev} \urcorner_a \, (\textit{row}_{\textit{cur}} \; c), \; [\ulcorner 0 \urcorner_c]_a, \; \ulcorner \textit{tl} \urcorner_a \, (\textit{suffix}_a \; c), \; b]_a$$

Calculating one entry According to the definition in Section 2.3.1 we need to define a function yielding the maximum of two Church numerals. We define the maximum function $\ulcorner \textit{max} \urcorner_c$ on numerals via $\ulcorner \textit{sub} \urcorner_c$:

$$\ulcorner \textit{max} \urcorner_c \quad := \quad \lambda n.\lambda n'.\ulcorner \textit{sub} \urcorner_c \; n \; n' \; (\lambda x.n) \; n'$$

If $\ulcorner \textit{sub} \urcorner_c \; n \; n' \neq_\beta \ulcorner 0 \urcorner_c$, n is the maximum of the two numerals and the result is n because $\lambda x.n$ is applied at least once. Otherwise n' is the result.

We construct the function that yields $M_{i,j}$ given the three neighbors (left, diagonal, up) and the two elements a_i and b_j:

$$\textit{calcentry} \quad := \quad \lambda l.\lambda d.\lambda u.\lambda a_i.\lambda b_j.\ulcorner = \urcorner_c \; a_i \; b_j \; (\ulcorner \textit{add}_1 \urcorner_c \; d) \; (\ulcorner \textit{max} \urcorner_c \; l \; u)$$

The inner iterated function The inner iterated function *it* maps from the current configuration to the next configuration. *it* computes the next entry via an application of *calcentry* and makes progress in the last row and in the b suffix:

$$\begin{aligned}
\textit{it} \quad := \quad \lambda c.[&\ulcorner \textit{tl} \urcorner_a \, (\textit{row}_{\textit{last}} \; c), \\
&\ulcorner \textit{cons} \urcorner_a \, (\textit{calcentry} \, (\ulcorner \textit{hd} \urcorner_a \, (\textit{row}_{\textit{cur}} \; c)) \\
&\qquad\qquad\qquad\qquad (\ulcorner \textit{hd} \urcorner_a \, (\textit{row}_{\textit{last}} \; c)) \\
&\qquad\qquad\qquad\qquad (\ulcorner \textit{hd} \urcorner_a \, (\ulcorner \textit{tl} \urcorner_a \, (\textit{row}_{\textit{last}} \; c))) \\
&\qquad\qquad\qquad\qquad (\ulcorner \textit{hd} \urcorner_a \, (\textit{suffix}_a \; c)) \\
&\qquad\qquad\qquad\qquad (\ulcorner \textit{hd} \urcorner_a \, (\textit{suffix}_b \; c))) \\
&\qquad\quad (\textit{row}_{\textit{cur}} \; c), \\
&\textit{suffix}_a \; c, \\
&\ulcorner \textit{tl} \urcorner_a \, (\textit{suffix}_b \; c)]_a
\end{aligned}$$

To fill the entries of one row *it* is iterated the length of b times.

Extracting the result The length of a longest common subsequence is the last computed entry in the current row of the final configuration:

$$\textit{extract} \quad := \quad \lambda c.\ulcorner \textit{hd} \urcorner_a \, (\textit{row}_{\textit{cur}} \; c)$$

Assembling the representation of *LLCS* With the above auxiliary definitions a function computes the length of a longest common subsequence of two sequences $a = [a_1, \ldots, a_n]_a$ and $b = [b_1, \ldots, b_{n'}]_a$ by (1) constructing the initial configuration via conf_{start}, (2) iteratively n times filling a row (via n' times applying *it*) and reconfiguring, and finally (3) extracting the result via *extract* from the final configuration:

$$\ulcorner \mathit{LLCS} \urcorner_a \quad := \quad \lambda a.\lambda b.\mathit{extract}\,(\ulcorner \mathit{len} \urcorner_a\, a\,(\lambda c.\ulcorner \mathit{len} \urcorner_a\, b\,\mathit{it}\,(\mathit{reconf}\; c\; b))\,(\mathit{conf}_{start}\; a\,(\ulcorner \mathit{len} \urcorner_a\, b)))$$

With the configuration representing the 'state' of the computation the above representation of *LLCS* realize a nested for-loop. Via bounded iteration $\ulcorner \mathit{LLCS} \urcorner_a$ works for pair-represented lists augmented with its length. Exchanging the operations to those of Church lists we immediately have $\ulcorner \mathit{LLCS} \urcorner_c$ which operate on Church lists. A definition of $\ulcorner \mathit{LLCS} \urcorner_p$ would abstract over the lengths of prefixes of a and b to consider.

2.4 Scott numerals

Scott — like Church — also proposed a representation of the natural numbers in the λ-calculus [17]:[6]

$$\begin{array}{rclcl}
\ulcorner 0 \urcorner_s & := & \lambda h.\lambda t.h & & \\
\ulcorner 1 \urcorner_s & := & \lambda h.\lambda t.t\,\ulcorner 0 \urcorner_s & = & \lambda h.\lambda t.t\,(\lambda h.\lambda t.h) \\
\ulcorner 2 \urcorner_s & := & \lambda h.\lambda t.t\,\ulcorner 1 \urcorner_s & = & \lambda h.\lambda t.t\,(\lambda h.\lambda t.t\,(\lambda h.\lambda t.h)) \\
 & \vdots & & & \\
\ulcorner n+1 \urcorner_s & := & \lambda h.\lambda t.t\,\ulcorner n \urcorner_s & = & \lambda h.\lambda t.t\,(\overbrace{\lambda h.\lambda t.t\,(\ldots(\lambda h.\lambda t.t}^{n \text{ times}}\,(\lambda h.\lambda t.h))\ldots))
\end{array}$$

By construction the successor function on Scott numerals stands out by abstracting over the numeral:

$$\ulcorner \mathit{add}_1 \urcorner_s \quad := \quad \lambda n.\lambda h.\lambda t.t\; n$$

As demonstrated in the previous section the Church numerals have built-in computational power: a numeral represents a bounded iteration. Within that system a representation of the predecessor function is non-trivial. In contrast, the predecessor on Scott numerals is immediate:

$$\ulcorner \mathit{sub}_1 \urcorner_s \quad := \quad \lambda n.n\,\ulcorner 0 \urcorner_s\,(\lambda t.t)$$

2.4.1 Scott numerals are selectors in pair-represented lists

Scott numerals are closely related to lists represented as properly nested pairs as presented in Section 2.2.5. The central observation is that the Scott numeral $\ulcorner n \urcorner_s$ is a selector of the n-th element of such lists:

$$[a_0, \ldots, a_n, \ldots, a_j]_p\,\ulcorner n \urcorner_s =_\beta a_n$$

[6]We subscribe the n-th Scott numeral $\ulcorner n \urcorner_s$ with s to distinguish from the corresponding Church numeral $\ulcorner n \urcorner_c$.

2.4.2 Lists and streams

An abstraction constructing pair-represented lists on the form:

$$[f\ (\ldots\ f\ (f\ a)\ \ldots), \ldots, f\ a, a]_p$$

given f, a, and $\ulcorner n \urcorner_c$ again relies on the bounded iteration built into the Church numerals:

$$\ulcorner \mathit{cov}_{\mathit{list}} \urcorner_p \quad := \quad \lambda f.\lambda a.\lambda n.n\ (\lambda l.\ulcorner \mathit{cons} \urcorner_p\ (f\ (\ulcorner \mathit{hd} \urcorner_p\ l))\ l)\ [a]_p$$

If $n + 1$ is the length of such a list, the first element in that list is f iterated n times starting from a.

Streams are *possibly infinite* sequences. Dual to $\ulcorner \mathit{cov}_{\mathit{list}} \urcorner_p$ we seek a stream operator $\ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p$ that, given f and an initial element a produces prefixes of the stream:

$$[a, f\ a, \ldots, f\ (\ldots\ f\ (f\ a)\ \ldots), \ldots]_p$$

The n-th element in such prefixes is f iterated n times starting from a. The constraints on the definition of $\ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p$ are *co-inductive*:

$$\begin{array}{rcl} \ulcorner \mathit{hd} \urcorner_p\ (\ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p\ f\ a) & =_\beta & a \\ \ulcorner \mathit{tl} \urcorner_p\ (\ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p\ f\ a) & =_\beta & \ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p\ f\ (f\ a) \end{array}$$

Taking the head of the stream $\ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p\ f\ a$ yields a and taking the tail of that stream yields the stream specialized to the same function but now as head element f iterated once more, i.e., $f\ a$. A definition of $\ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p$ satisfying the two constraints is realized by use of the fixed-point combinator Y, which satisfies $Y\ f =_\beta f\ (Y\ f)$:

$$\ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p \quad := \quad \lambda f.Y\ (\lambda g.\lambda a.\lambda s.s\ a\ (g\ (f\ a)))$$

2.4.3 Functions on Scott numerals

The addition function As noted $\ulcorner n \urcorner_s$ is the selector of the n-th element of a pair-represented list. That is, to define addition of two numerals $\ulcorner n \urcorner_s$ and $\ulcorner n' \urcorner_s$, it would be useful to have the list $l = [\ulcorner n \urcorner_s, \ulcorner \mathit{add}_1 \urcorner_s\ \ulcorner n \urcorner_s, \ldots, \ulcorner \mathit{add}_{n'} \urcorner_s\ \ulcorner n \urcorner_s]_p$, because $l\ \ulcorner n' \urcorner_s =_\beta \ulcorner \mathit{add}_{n'} \urcorner_s\ \ulcorner n \urcorner_s =_\beta \ulcorner n + n' \urcorner_s$. Unfortunately, the construction of l is not immediate given $\ulcorner n' \urcorner_s$ because $\ulcorner n' \urcorner_s$ has no built-in computational power corresponding to n' like the corresponding Church numeral. That is, the length of the list to build is not known. A solution is to use a stream instead of a list: Given $\ulcorner n \urcorner_s$ and $\ulcorner n' \urcorner_s$ the addition function is thus defined as a specialization of $\ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p$ to the successor function:

$$\ulcorner \mathit{add} \urcorner_s \quad := \quad \ulcorner \mathit{cov}_{\mathit{stream}} \urcorner_p\ \ulcorner \mathit{add}_1 \urcorner_s$$

Initializing the stream with $\ulcorner n \urcorner_s$ and taking the n'-th element of the resulting stream yields $\ulcorner n + n' \urcorner_s$. Addition of Scott numerals is hence defined via iteration of the simpler successor function — just like addition of Church numerals.

The multiplication function, the exponentiation function, and the subtraction function Specializing $\ulcorner cov_{stream} \urcorner_p$ to other functions, which are then iterated, yields other standard functions on numerals. Like for Church numerals the multiplication function is defined by iterating addition, the exponentiation function is defined by iterating multiplication, and the subtraction function is defined by iterating the predecessor function:

$$\begin{array}{rcl} \ulcorner mult \urcorner_s & := & \lambda n. \ulcorner cov_{stream} \urcorner_p \; (\ulcorner add \urcorner_s \; n) \; \ulcorner 0 \urcorner_s \\ \ulcorner exp \urcorner_s & := & \lambda n. \ulcorner cov_{stream} \urcorner_p \; (\ulcorner mult \urcorner_s \; n) \; \ulcorner 1 \urcorner_s \\ \ulcorner sub \urcorner_s & := & \ulcorner cov_{stream} \urcorner_p \; \ulcorner sub_1 \urcorner_s \end{array}$$

Because $\ulcorner sub_1 \urcorner_s$ maps $\ulcorner 0 \urcorner_s$ to $\ulcorner 0 \urcorner_s$ the subtraction function is *monus*.

The factorial function With the iterated function it_{fac} in the definition of $\ulcorner fac \urcorner_c$ changed to use functions on Scott numerals (call it $itfac'$) the factorial function on Scott numerals is also seen to be closely related to its counterpart for Church numerals:

$$\ulcorner fac \urcorner_s \quad := \quad \lambda n. \ulcorner \pi_2 \urcorner \; (\ulcorner cov_{stream} \urcorner_p \; itfac' \; \langle \ulcorner 1 \urcorner_s, \ulcorner 1 \urcorner_s \rangle \; n)$$

2.5 A correspondence between Church numerals and Scott numerals

Translations between the Scott numeral system and the Church numeral system follow the same line of thought as the definitions from the two previous sections:

$$\begin{array}{rcl} \ulcorner church2scott \urcorner & := & \lambda n. n \; \ulcorner add_1 \urcorner_s \; \ulcorner 0 \urcorner_s \\ \ulcorner scott2church \urcorner & := & \lambda n. \ulcorner cov_{stream} \urcorner_p \; \ulcorner add_1 \urcorner_c \; \ulcorner 0 \urcorner_c \; n \end{array}$$

The functions defined for Church numerals apply such a numeral $\ulcorner n \urcorner_c$ to an iterator function f and an element a yielding the n-th iteration of f starting from a, which is the *first element in the list of intermediate results*:

$$[f\,(\ldots\; f\,(f\,a)\ldots), \ldots, f\,a, a]$$

The functions defined for Scott numerals utilize a stream obtained by a specialization of $\ulcorner cov_{stream} \urcorner_p$ to an iterator function f and an initial element a. Applying the stream to $\ulcorner n \urcorner_s$ yields the n-th iteration of f starting from a, which is the *n-th element in the stream of intermediate results:*

$$[a, f\,a, \ldots, f\,(\ldots\; f\,(f\,a)\ldots), \ldots]_p$$

A general relationship between the two numeral systems hence stands out:

$$\ulcorner n \urcorner_c \; f \; a \quad =_\beta \quad \ulcorner cov_{stream} \urcorner_p \; f \; a \; \ulcorner n \urcorner_s$$

2.6 An alternative definition of the subtraction function for Church numerals

2.6.1 Generalizing Kleene's predecessor function

The structure of Kleene's predecessor function presented in Section 2.2.2 can easily be adapted to define functions that subtract $\ulcorner 2 \urcorner_c$, $\ulcorner 3 \urcorner_c$, etc. from a Church numeral: To subtract $\ulcorner 1 \urcorner_c$, *two* numerals are used in each element in the series of intermediate results as presented in Section 2.2.2. To subtract $\ulcorner 2 \urcorner_c$, *three* numerals are used in each element:

$$\begin{array}{c}
\langle \ulcorner 0 \urcorner_c, \langle \ulcorner 0 \urcorner_c, \ulcorner 0 \urcorner_c \rangle\rangle \\
\langle \ulcorner 1 \urcorner_c, \langle \ulcorner 0 \urcorner_c, \ulcorner 0 \urcorner_c \rangle\rangle \\
\langle \ulcorner 2 \urcorner_c, \langle \ulcorner 1 \urcorner_c, \ulcorner 0 \urcorner_c \rangle\rangle \\
\langle \ulcorner 3 \urcorner_c, \langle \ulcorner 2 \urcorner_c, \ulcorner 1 \urcorner_c \rangle\rangle \\
\langle \ulcorner 4 \urcorner_c, \langle \ulcorner 3 \urcorner_c, \ulcorner 2 \urcorner_c \rangle\rangle \\
\vdots \\
\langle \ulcorner n \urcorner_c, \langle \ulcorner n-1 \urcorner_c, \ulcorner n-2 \urcorner_c \rangle\rangle
\end{array}$$

Each element in the series can be generated from the previous element via $\ulcorner add_1 \urcorner_c$, and the definition of $\ulcorner sub_2 \urcorner_c$ satisfying $\ulcorner sub_2 \urcorner_c \ulcorner n \urcorner_c =_\beta \ulcorner n-2 \urcorner_c, n \geq 2$ follows the same line of thought as Kleene's predecessor:

$$\ulcorner sub_2 \urcorner_c \;:=\; \lambda n. \ulcorner \pi_2 \urcorner (\ulcorner \pi_2 \urcorner (n\, (\lambda p. \langle \ulcorner add_1 \urcorner_c\, (\ulcorner \pi_1 \urcorner\, p), \langle \ulcorner \pi_1 \urcorner\, p, \ulcorner \pi_1 \urcorner\, (\ulcorner \pi_2 \urcorner\, p)\rangle\rangle)\, \langle \ulcorner 0 \urcorner_c, \langle \ulcorner 0 \urcorner_c, \ulcorner 0 \urcorner_c \rangle\rangle))$$

Here $\ulcorner sub_2 \urcorner_c \ulcorner 1 \urcorner_c =_\beta \ulcorner sub_2 \urcorner_c \ulcorner 0 \urcorner_c =_\beta \ulcorner 0 \urcorner_c$.

In general $n' + 1$ numerals are used in the definition of a function $\ulcorner sub_{n'} \urcorner_c$ that satisfies $\ulcorner sub_{n'} \urcorner_c \ulcorner n \urcorner_c =_\beta \ulcorner n - n' \urcorner_c$ when $n \geq n'$ and $\ulcorner sub_{n'} \urcorner_c \ulcorner n \urcorner_c =_\beta \ulcorner 0 \urcorner_c$ when $n < n'$.

2.6.2 A course-of-value representation of Church numerals

We observe that, instead of generating the whole series of intermediate elements the last element in the series can be constructed immediately as a list by specializing $\ulcorner cov_{list} \urcorner_p$ defined in Section 2.4.2 to iterate the successor function starting from $\ulcorner 0 \urcorner_c$:

$$\begin{array}{rcl}
\ulcorner c2cov \urcorner_p & := & \ulcorner cov_{list} \urcorner_p \ulcorner add_1 \urcorner_c \ulcorner 0 \urcorner_c \\
\ulcorner c2cov \urcorner_p \ulcorner n \urcorner_c & =_\beta & [\ulcorner n \urcorner_c, \ulcorner n-1 \urcorner_c, \ldots, \ulcorner 0 \urcorner_c]_p
\end{array}$$

$\ulcorner c2cov \urcorner_p$ is thus a mapping from Church numerals to a *course-of-value* representation of Church numerals here using nested pairs [70, Chapter 3]. The n'-th prefix of this list constitute the series of intermediate results encountered when iterating $\ulcorner sub_1 \urcorner_c$ in the subtraction $\ulcorner sub \urcorner_c \ulcorner n \urcorner_c \ulcorner n' \urcorner_c$ — only in reversed order. If $n \geq n'$, by analogy with streams underlying Scott numerals, $\ulcorner n - n' \urcorner_c$ is the n'-th element in the list:

$$\ulcorner minus \urcorner_c \;:=\; \lambda n. \lambda n'. \ulcorner nth \urcorner_p\, n'\, (\ulcorner c2cov \urcorner_p\, n)$$

Here, $\ulcorner minus \urcorner_c \ulcorner n \urcorner_c \ulcorner n' \urcorner_c =_\beta \ulcorner sub \urcorner_c \ulcorner n \urcorner_c \ulcorner n' \urcorner_c$, if $n \geq n'$. Unfortunately, this equivalence does not hold in general when $n < n'$.

2.6.3 Lists as generalized pairs

The reason $\ulcorner minus \urcorner_c$ does not coincide with $\ulcorner sub \urcorner_c$ is the use of pair-represented lists for the underlying course-of-value representation. Another list representation generalizes the definition of pairs and represents lists directly:

$$\begin{array}{rcl}
\ulcorner nil \urcorner_g & := & \lambda s. \lambda n. n \\
\ulcorner cons \urcorner_g & := & \lambda h. \lambda t. \lambda s. \lambda n. s\, h\, t \\
\ulcorner hd \urcorner_g & := & \lambda l. l\, (\lambda h. \lambda t. h)\, \ulcorner 0 \urcorner_c \\
\ulcorner tl \urcorner_g & := & \lambda l. l\, (\lambda h. \lambda t. t)\, \ulcorner nil \urcorner_g \\
\ulcorner nil? \urcorner_g & := & \lambda l. l\, (\lambda h. \lambda t. \ulcorner false \urcorner)\, \ulcorner true \urcorner
\end{array}$$

Two series of equations show that this representation is consistent:

$$
\begin{array}{rcl}
\ulcorner hd \urcorner_g \, (\ulcorner cons \urcorner_g \, a \, l) & =_\beta & \ulcorner hd \urcorner_g \, (\lambda s.\lambda n.s \, a \, l) \\
& =_\beta & (\lambda s.\lambda n.s \, a \, l) \, (\lambda h.\lambda t.h) \, \ulcorner 0 \urcorner_c \\
& =_\beta & a
\end{array}
$$

$$
\begin{array}{rcl}
\ulcorner tl \urcorner_g \, (\ulcorner cons \urcorner_g \, a \, l) & =_\beta & \ulcorner tl \urcorner_g \, (\lambda s.\lambda n.s \, a \, l) \\
& =_\beta & (\lambda s.\lambda n.s \, a \, l) \, (\lambda h.\lambda t.t) \, \ulcorner nil \urcorner_g \\
& =_\beta & l
\end{array}
$$

According to the new representation a list is properly nested pairs abstracting over the result in case of taking $\ulcorner hd \urcorner_g$ or $\ulcorner tl \urcorner_g$ on the empty list. The correctness of the following two equations is hence immediate:

$$
\begin{array}{rcl}
\ulcorner tl \urcorner_g \, \ulcorner nil \urcorner_g & =_\beta & \ulcorner nil \urcorner_g \\
\ulcorner hd \urcorner_g \, \ulcorner nil \urcorner_g & =_\beta & \ulcorner 0 \urcorner_c
\end{array}
$$

These properties do not hold for pair-represented lists. For Church lists and for pair-represented lists augmented with the length the properties could be ensured.

2.6.4 The alternative definition of the subtraction function

The above two properties give that taking the tail of a list more times than the number of elements just gives the empty list, and taking the n-th element of a list with fewer elements yields $\ulcorner 0 \urcorner_c$. In other words, with this representation the tail function has the same nature on lists as the monus function of Section 2.1 has on Church numerals. With $\ulcorner c2cov \urcorner_p$ and $\ulcorner nth \urcorner_p$ adjusted to the generalized list representation ($\ulcorner c2cov \urcorner_g$ and $\ulcorner nth \urcorner_g$) the definition of $\ulcorner minus \urcorner_c$ hence coincides with the monus function on all Church numerals:

$$\ulcorner sub \urcorner_c \quad := \quad \lambda n.\lambda n'.\ulcorner nth \urcorner_g \, n' \, (\ulcorner c2cov \urcorner_g \, n)$$

This representation of *monus* is linear whereas the previous representation is quadratic.[7]

2.7 The quotient function and the remainder function

Relying on the monus function Representations of the quotient function and the remainder function can be defined by iterating the subtraction function. The function $next_{n'}$ for $n' > 0$, is actually iterated:

$$
next_{n'} \, \langle \ulcorner r \urcorner_c, \ulcorner q \urcorner_c \rangle \quad = \quad \begin{cases} \langle \ulcorner sub \urcorner_c \, \ulcorner r \urcorner_c \, \ulcorner n' \urcorner_c, \ulcorner q+1 \urcorner_c \rangle & \text{if } r \geq n' \\ \langle \ulcorner r \urcorner_c, \ulcorner q \urcorner_c \rangle & \text{if } r < n' \end{cases}
$$

That is, a pair of numerals is threaded such that the quotient can be incremented in each iteration where $r \geq n'$. Iteratively applying $next_{n'}$ to the initial pair $\langle \ulcorner n \urcorner_c, \ulcorner 0 \urcorner_c \rangle$ the pair is fixed after at most n iterations because the numeral in the first component each step — by use of $\ulcorner sub \urcorner_c$ — becomes n' smaller until it is fixed for the rest of the iterations. The remainder is the first component and the quotient n/n' is the second component in the final

[7]If the course-of-value representation used Church lists or augmented pair-represented lists the new representation of *monus* would not be linear.

pair (n iterations are used if $n' = 1$). Representations of the quotient function *quo* and the remainder function *rem* heavily relying on $\ulcorner sub \urcorner_c$ are accordingly defined (with $next_{n'}$ represented as $next\ \ulcorner n' \urcorner_c$):

$$\begin{array}{rcl} next & := & \lambda n'.\lambda p.\ulcorner < \urcorner_c\ (\ulcorner \pi_1 \urcorner\ p)\ n'\ p\ \langle \ulcorner sub \urcorner_c\ (\ulcorner \pi_1 \urcorner\ p)\ n', \ulcorner add_1 \urcorner_c\ (\ulcorner \pi_2 \urcorner\ p)\rangle \\ \ulcorner quo \urcorner_c & := & \lambda n.\lambda n'.\ulcorner \pi_2 \urcorner\ (n\ (next\ n')\ \langle n, \ulcorner 0 \urcorner_c\rangle) \\ \ulcorner rem \urcorner_c & := & \lambda n.\lambda n'.\ulcorner \pi_1 \urcorner\ (n\ (next\ n')\ \langle n, \ulcorner 0 \urcorner_c\rangle) \end{array}$$

Again, the efficiency of the representations depends on the choice of definition of $\ulcorner sub \urcorner_c$. In every iteration $\ulcorner sub \urcorner_c$ is used in the test ($\ulcorner < \urcorner_c$). Until the quotient has been found $\ulcorner sub \urcorner_c$ is also used to construct the first component in the next pair.

Relying on the monus function but with result indicator The information that the result after q iterations is obtained is thrown away by having an explicit check for $r < n'$ in each iteration. In addition to the two components of the pair a boolean ($\ulcorner true \urcorner$ or $\ulcorner false \urcorner$) can be threaded as an indicator for whether the test has to be computed or the remainder and the quotient have been found. The achievement is definitions that use $\ulcorner sub \urcorner_c$ only in the iterations until the result has been found. For the rest of the iterations, both the test and the construction are constant operations.

Exploiting the course-of-value representation In general though, $\ulcorner quo \urcorner_c\ \ulcorner n \urcorner_c\ \ulcorner n' \urcorner_c$ and $\ulcorner rem \urcorner_c\ \ulcorner n \urcorner_c\ \ulcorner n' \urcorner_c$ are not linear in n and n' since in each iteration a list proportional to n is created because of the monus function $\ulcorner sub \urcorner_c$.

An alternative is not to rely on the monus function by deforesting the intermediate lists. The iterated function $next_{n'}$, $n' > 0$ here as first component threads a list instead of a Church numeral:

$$next_{n'}\ \langle r, \ulcorner q \urcorner_c\rangle \quad = \quad \begin{cases} \langle r', \ulcorner q+1 \urcorner_c\rangle & \text{if } r'\text{, the } n'\text{-th tail of } r\text{, is not the empty list} \\ \langle r, \ulcorner q \urcorner_c\rangle & \text{if the } n'\text{-th tail of } r \text{ is the empty list} \end{cases}$$

If this function is iterated starting from the pair $\langle l, \ulcorner 0 \urcorner_c\rangle$ where l is the course-of-value representation of n using generalized lists (i.e., $l = \ulcorner c2cov \urcorner_g\ \ulcorner n \urcorner_c$) the resulting pair is fixed after at most n iterations (where the length of l each time becomes n' smaller) with the quotient n/n' as the second component (again n iterations are used if $n' = 1$). The first component is the course-of-value representation of the remainder. The corresponding Church numeral is just the head element:

$$\begin{array}{rcl} next & := & \lambda n'.\lambda p.(\lambda t.\ulcorner nil? \urcorner_g\ t\ p\ \langle t, \ulcorner add_1 \urcorner_c\ (\ulcorner \pi_2 \urcorner\ p)\rangle)\ (n'\ \ulcorner tl \urcorner_g\ (\ulcorner \pi_1 \urcorner\ p)) \\ \ulcorner quo \urcorner_c & := & \lambda n.\lambda n'.\ulcorner \pi_2 \urcorner\ (n\ (next\ n')\ \langle \ulcorner c2cov \urcorner_g\ n, \ulcorner 0 \urcorner_c\rangle) \\ \ulcorner rem \urcorner_c & := & \lambda n.\lambda n'.\ulcorner hd \urcorner_g\ (\ulcorner tl \urcorner_g\ (\ulcorner \pi_1 \urcorner\ (n\ (next\ n')\ \langle \ulcorner c2cov \urcorner_g\ n, \ulcorner 0 \urcorner_c\rangle))) \end{array}$$

Remark: It is essential that the course-of-value representation of numerals use the generalized pair representation of lists such that $\ulcorner tl \urcorner_g\ \ulcorner nil \urcorner_g = \ulcorner nil \urcorner_g$ and $\ulcorner hd \urcorner_g\ \ulcorner nil \urcorner_g = \ulcorner 0 \urcorner_c$

The predicate $\ulcorner nil? \urcorner_g$ is a constant-time operation (as opposed to $\ulcorner zero? \urcorner_c$ which is a linear operation). Therefore the achievement is representations of the quotient function and the remainder function that operate in linear time. In contrast, when building on $\ulcorner sub \urcorner_c$ the functions operate in quadratic time: A Church numeral $\ulcorner r \urcorner_c$ (proportional to n) is at each iteration mapped to the course-of-value representation via $\ulcorner c2cov \urcorner_g$ which operates in linear time. Then the n'-th tail l' is found followed by a (constant time) mapping of l' to the corresponding Church numeral. In the next iteration l' is constructed again via $\ulcorner c2cov \urcorner_g$. In the revised version above, the mapping from Church numerals to the course-of-value representation is only performed once—before the first iteration. The mapping back to Church numerals is only performed once—after the last iteration.[8]

2.8 Extending the λ-calculus with literals and a corresponding primitive successor function

As developed in this chapter Church numerals are defined as abstractions over a primitive zero value and a primitive successor function. Instead of representing numbers by λ-terms we introduce *literal* numbers including a primitive $\ulcorner 0 \urcorner$ and a corresponding primitive successor function as a *basic function* directly in the calculus. Such basic entities are collectively called *basic constants*.

The λβδ-calculus The term language of the λ-calculus is extended with literal integers and the corresponding primitive successor $\mathtt{S}$:

$$\begin{array}{lrcl} \textit{Var} & x & ::= & (\textit{unspecified}) \\ \textit{Term} & t & ::= & x \mid \lambda x.t \mid t\,t \mid b \\ \textit{Basic} & b & ::= & l \mid q \\ \textit{Lit} & l & ::= & \{\ulcorner 0 \urcorner, \ulcorner 1 \urcorner, \ulcorner 2 \urcorner, \ulcorner 3 \urcorner, \ldots\} \\ \textit{BasicF} & q & ::= & \mathtt{S} \end{array}$$

The new productions give rise to an extended notion of reduction $\beta\delta := \beta \cup \delta$, where δ utilizes a function $\mathtt{capp}$, which in our setting is very simple:[9]

$$\delta : \qquad q\,l \ \rightarrow\ \mathtt{capp}\,(q,\ l)$$

$$\begin{array}{rcl} \mathtt{capp} & : & \textit{BasicF} \times \textit{Lit} \rightarrow \textit{Basic} \\ \mathtt{capp}\,(\mathtt{S}, \ulcorner n \urcorner) & = & \ulcorner n+1 \urcorner \end{array}$$

The set of βδ-normal forms contain no β-redex and no δ-redex:

$$\begin{array}{lrcl} & a & ::= & x \mid a\,n \mid l\,n \mid q\,d \\ & d & ::= & a \mid \lambda x.n \mid q \\ \textit{NForm} & n & ::= & d \mid l \end{array}$$

Compared to the definition of normal forms in the λ-calculus on page 8 one extra nonterminal d is needed to denote a term in normal form, which is not a literal.

The translations from a Church numeral or a Scott numeral into the corresponding literal are immediate:

$$\begin{array}{rcl} \ulcorner \textit{toliteral} \urcorner_c & := & \lambda n.n\,\mathtt{S}\,\ulcorner 0 \urcorner \\ \ulcorner \textit{toliteral} \urcorner_s & := & \textit{cov}_{\textit{stream}}\,\mathtt{S}\,\ulcorner 0 \urcorner \end{array}$$

[8] By analogy with previously defined functions the various versions of the quotient function and the remainder function can directly be translated to corresponding versions operating on Scott numerals.

[9] Note that $\ulcorner n \urcorner$ no longer has a subscript because it denotes an unspecified representation of the number n.

Standard properties of the $\lambda\beta\delta$-calculus Extending the notions of reduction from β to $\beta\delta$ yields a relation which is also Church-Rosser [59]. Furthermore, the restricted compatibility closure defining normal-order reduction found in Section 1.5 can be adjusted to cope with δ, and still satisfy the standardization theorem. Adding a more general version of δ, which defines a contraction rule for the application of a range of basic functions to literals, is also possible. The Church-Rosser property would still hold and the normal-order reduction strategy again satisfies the standardization theorem. We use the term language and the δ-rule as defined above, because it is simpler but still sufficient for illustration.

Reduction to weak head normal forms The set of weak head normal forms is like in the λ-calculus generated from a grammar structurally equal to the grammar of normal forms:

$$
\begin{array}{lrcl}
 & a & ::= & x \mid a\,t \mid l\,t \mid q\,d \\
 & d & ::= & a \mid \lambda x.t \mid q \\
\textit{WHNForm} & w & ::= & d \mid l
\end{array}
$$

Normalization to weak head normal forms in the λ-calculus was defined on page 13 via a restricted compatible closure of the notion of reduction. We let weak head normalization be defined via a restricted compatible closure over $\beta\delta$ to cope with the basic constants:

$$
\frac{(t, t') \in \beta\delta}{t \mapsto_n t'} \qquad \frac{t_1 \mapsto_n t_1'}{t_1\ t_2 \mapsto_n t_1'\ t_2} \qquad \frac{t_2 \mapsto_n t_2'}{q\ t_2 \mapsto_n q\ t_2'}
$$

The adjusted definition of $\mapsto_n$ is the only change in the function for weak head normalization:

$$\textit{normalize}_n\ t = w \quad \text{iff} \quad t \mapsto_n^* w$$

Proposition 2 on page 14 also holds for the new definitions of $\textit{normalize}_n$ and weak head normal forms. Especially $\textit{normalize}_n\ t = b \iff t =_{\beta\delta} b$.

In the following we will often just say the λ-calculus when it is clear from context that it is the $\lambda\beta\delta$-calculus over the extended term language from this section.

2.9 Summary

In this chapter we defined Church numerals and we developed standard functions exploiting the bounded iteration built into such numerals by iterating simpler functions. We defined structured elements to present Kleene's definition of the predecessor function on Church numerals and thereby the corresponding definition of monus as iterating the predecessor function.

Following the line of thought in Kleene's predecessor function, we presented the iterative factorial function on Church numerals presented by Goldberg and Reynolds. Identifying the general pattern as that of dynamic programming, we illustrated the general principle by a sample problem *LLCS* and we presented a definition corresponding to an efficient algorithm.

We introduced Scott numerals which on the one hand do not have built-in computational power, but on the other hand have a simpler predecessor function. We observed that Scott numerals are selectors in pair-represented lists, and furthermore by the definition of streams the usual functions on Scott numerals are also definable via iteration of simpler functions.

The similar structure between functions for Scott numerals and the corresponding for Church numerals gave rise to a general relationship between Scott numerals and Church numerals. This general relationship emphasized a course-of-value representation underlying the Church numerals and made us identify an inefficiency in the representation of *monus* on Church numerals, when it is defined by iterating the predecessor function. We presented an efficient representation of *monus* on Church numerals.

Usually, the definitions of standard functions like the quotient function and the remainder function are defined via the monus function and thereby inherit the properties of the actual representation of *monus*. Exploiting the course-of-value representation of numerals we presented alternative representations of the quotient function and the remainder function that do not use *monus* at all. The new representations are linear whereas via *monus* the representations are at least quadratic. The results on the course-of-value representations are joint work with Olivier Danvy, but mistakes are mine.

Chapter 3

Other λ-calculi

The calculus defined in Chapter 1 and used for definability in Chapter 2 is the original λ-calculus as defined by Church [11]. Several extensions and variants have been defined afterwards. In this chapter we review some of these calculi.

Roadmap In Section 3.1 we briefly touch on a direct extension of the λ-calculus. Thereafter we introduce in Section 3.2 another view on substitution, which does not rely on a meta-construction as in the definition of β and explicit in the definition of β_{deB}. This view leads to a new class of λ-calculi with *explicit substitutions*. We continue in Section 3.3 with a standard variation of the λ-calculus: Plotkin's λ_v-calculus [59].

3.1 The $\lambda\beta\eta$-calculus

In the λ-calculus as defined in Chapter 1, β is the only contraction rule. We can directly extend the calculus by extending the notion of reduction. We add the contraction rule η:

$$\eta : \qquad (\lambda x.t\, x) \to t, \text{ if } x \notin \mathtt{FV}\, t$$

and extend the notion of reduction to $\beta\eta := \beta \cup \eta$. We define $\beta\eta$-reduction $\to_{\beta\eta}$, its reflexive transitive closure $\to^*_{\beta\eta}$, and equality $=_{\beta\eta}$ analogously with the definitions in Section 1.3 and leave out the details.

The set of terms in $\beta\eta$-normal form is by construction no longer specified by the grammar in Section 1.2. Some β-normal forms are not η-normal forms. On the other hand, the grammar of weak head normal forms from Section 1.5.3 remains unchanged.

We emphasize two properties related to normal forms and weak head normal forms: (1) $\to^*_{\beta\eta}$ satisfies that same diamond property as $\to^*_{\beta}$, i.e., $\beta\eta$ is Church-Rosser [7, Theorem 3.3.9], and (2) The set of weak head normal forms are closed under β-reductions but not under $\beta\eta$-reductions (which follows directly from the definition of η where t can be a non-value).

With the η-rule included some of the representations of functions on numerals in Chapter 2 can be simplified. For example, $\ulcorner add_n \urcorner_c$ used in connection with the multiplication function on Church numerals can be η-reduced (n and $\ulcorner add_1 \urcorner_c$ are both closed terms):

$$\begin{array}{rcl} \ulcorner add_n \urcorner_c & := & \lambda n'.n \ulcorner add_1 \urcorner_c\, n' \\ & \to_{\beta\eta} & n \ulcorner add_1 \urcorner_c \end{array}$$

3.2 Explicit substitutions

As the notion of reduction β is defined in Section 1.2, an actual substitution is performed in constructing the contractum of a redex. The substitution was denoted by a meta-construction and intuitively explained. Such a substitution can formally be defined via a recursive descent on a term as seen in Section 1.6.2, in connection with variables represented as de Bruijn indices. The point is, a substitution is altering the term by replacing free occurrences of the variable at hand by the right-hand sub-term of the application.

An alternative way to handle substitution, known as *explicit substitutions*, is to somehow remember substitutions as part of the syntactic unit by an incorporation into the term language. No actual substitution is performed when forming the contractum of a β-redex. Instead, the notion of reduction is extended with new contraction rules for (1) distribution of such unrealized substitutions and (2) eventually realizing a substitution of a term for a variable and (3) removal of unneeded substitutions.

Curien's calculus of closures $\lambda\rho$ The $\lambda\rho$-calculus [15] uses explicit substitution. A syntactic unit in the calculus essentially is the composition of a λ-term and a mapping from variables to (in general) syntactic units. Since Landin's work on evaluation of applicative expressions such a composition is known as a *closure* [50], and the $\lambda\rho$-calculus is for that reason also known as a *calculus of closure*.

Instead of named variables the terms contain de Bruijn indices as introduced in Section 1.6. That is, the substitution part of a closure is a mapping from natural numbers to closures. This mapping is represented as a list of closures.

A formal definition of Curien's $\lambda\rho$-calculus is omitted because a minimal extension of it is treated in Section 5.2.1. The extended calculus is the $\lambda\hat{\rho}$-calculus defined by Biernacka and Danvy [8,9].

3.3 The λ_v-calculus

The λ_v-calculus, which is due to Plotkin [59], was defined with a special purpose that will become clear in later chapters.

The term language Terms of the λ_v-calculus, as defined by Plotkin, include an unspecified notion of basic constants. We will be concrete and use the term language for the $\lambda\beta\delta$-calculus defined on page 35.

Notion of reduction The notions of reduction in the λ-calculus was in Chapter 1 defined as the relation β. In the $\lambda\beta\delta$-calculus of Section 2.8 the notion of reduction is $\beta\delta$: an extension of β for an extended set of λ-terms including some basic constants. In the λ_v-calculus the notion of reduction is $\beta_v \cup \delta$, where β_v is a restricted version of β.

The β_v-rule relies on a syntactic notion of *values*, which are exactly the terms not being applications:

$$\mathit{Val} \qquad v \ ::= \ x \mid b \mid \lambda x.t$$

β_v allows only contractions where the right sub-term of a β-redex is a syntactic value:

$$\beta_v : \qquad (\lambda x.t)\, v \ \rightarrow \ t\{v/x\}$$

One-step reduction and equality One-step reduction, $\rightarrow_{\beta_v\delta}$, in the λ_v-calculus is, like in the λ-calculus, defined as the compatible closure of the notion of reduction according to the same unrestricted compatibility rules or contexts found in Section 1.3.

We likewise take $\rightarrow^*_{\beta_v\delta}$ as the reflexive transitive closure of $\rightarrow_{\beta_v\delta}$. Equality $=_{\beta_v\delta}$ is defined as the smallest equivalence over $\rightarrow^*_{\beta_v\delta}$.

Properties and normal forms The resulting calculus is also Church-Rosser [59, page 135]. That is, $\rightarrow^*_{\beta_v\delta}$ satisfies the same diamond property as $\rightarrow^*_{\beta}$.

Again the corollary of uniqueness of normal forms of terms holds, where a normal form now is a term without any β_v-redexes or δ-redexes. Because every β_v-redex is also a β-redex but not the other way around (i.e., $\beta_v \subset \beta$) the definition of β_v-normal forms will be an extension of the grammar of β-normal forms from Section 1.2 employing an extra nonterminal to represent non-value normal forms h_v:

$$\begin{array}{lcl} h_v & ::= & a_v\ n_v \mid (\lambda x.n_v)\ h_v \\ a_v & ::= & x \mid h_v \\ n_v & ::= & a_v \mid \lambda x.n_v \end{array}$$

One production is added for the case where the left sub-term of an application can be reduced to an abstraction but the right sub-term cannot be reduced to a value. To cope with the basic constants we introduce yet another nonterminal, d_v, just like in the grammar of normal forms in the $\lambda\beta\delta$-calculus on page 35:

$$\begin{array}{llcl} & h_v & ::= & a_v\ n_v \mid (\lambda x.n_v)\ h_v \mid l\ n_v \mid q\ d_v \\ & a_v & ::= & x \mid h_v \\ & d_v & ::= & a_v \mid \lambda x.n_v \mid q \\ \mathit{NForm}_v & n_v & ::= & d_v \mid l \end{array}$$

h_v still denotes the non-value normal forms.

Values vs. normal forms in the β_v-rule If the definition of β_v reads $(\lambda x.t)\ n_v \rightarrow t\{n_v/x\}$, where n_v is a normal form instead of a value, the diamond property is no longer satisfied, i.e., the notion of reduction would not be Church-Rosser. A minor change to our running sample term gives an illustrative counter example:

$$\begin{array}{rcl} (\lambda x.(\lambda y.z)\ (x\ x))\ (\lambda x.w\ (x\ x)) & \rightarrow_{\beta_v\delta} & (\lambda x.z)\ (\lambda x.w\ (x\ x)) \\ & \rightarrow_{\beta_v\delta} & z \\ \\ (\lambda x.(\lambda y.z)\ (x\ x))\ (\lambda x.w\ (x\ x)) & \rightarrow_{\beta_v\delta} & \lambda y.z\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))) \\ & \rightarrow_{\beta_v\delta} & (\lambda y.z)\ (w\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))) \\ & \rightarrow^*_{\beta_v\delta} & (\lambda y.z)\ (w\ \ldots ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))\ldots) \end{array}$$

Two reductions can be used to obtain the normal form z because $x\ x$ (which is not a value) is a normal form. But a series of reductions can be performed on the same term reaching terms that can never yield z by a series of reductions. In other words, the definition of values is essential, such that only the second series of reductions is possible in the calculus. Remark in addition that the above example does not involve basic constants. That is, the use of values and not normal forms in the β_v-rule is essential also in a purified version of the λ_v-calculus with only β_v as the notion of reduction.

β **vs.** β_v On the one hand, the fact that a term has a normal form in the $\lambda\beta\delta$-calculus does not imply that it has a normal form in the λ_v-calculus. A counter example is the term $(\lambda x.(\lambda y.z)\ (x\ x))\ (\lambda x.w\ (x\ x))$ from above. In the $\lambda\beta\delta$-calculus the term has normal form z because the two reductions in the first series above are β-reductions. In the λ_v-calculus only the second series is possible and that series can not be extended to obtain a normal form.

On the other hand, even though β_v is a restriction of β, a term with a normal form in the λ_v-calculus does not necessarily have a normal form in the $\lambda\beta\delta$-calculus. For example, the term $(\lambda d.(\lambda x.w\,(x\,x)))\,(y\,y)\,(\lambda x.w\,(x\,x))$ is a normal form in the λ_v-calculus but diverge in the $\lambda\beta\delta$-calculus because $(\lambda d.(\lambda x.w\,(x\,x)))\,(y\,y)\,(\lambda x.w\,(x\,x)) =_\beta (\lambda x.w\,(x\,x))\,(\lambda x.w\,(x\,x))$, which is the diverging sample term from Section 1.4. The above property influences on definability in the λ_v-calculus.

3.3.1 Standard reduction

In Section 1.5 the one-step reduction function $\mapsto_{\bar{n}}$ defined a strategy to perform reductions on terms in the λ-calculus. Likewise, an analogous one-step reduction function $\mapsto_{\bar{v}}$ can be defined for the λ_v-calculus. The definition is again defined as a restricted compatible closure now of $\beta_v\delta$ to obtain a partial function on terms which is defined on all terms not in normal form:

$$\frac{(t,t') \in \beta_v\delta}{t \mapsto_{\bar{v}} t'} \qquad \frac{t \mapsto_{\bar{v}} t'}{\lambda x.t \mapsto_{\bar{v}} \lambda x.t'} \qquad \frac{t_1 \notin \mathit{Val},\ t_1 \mapsto_{\bar{v}} t_1'}{t_1\ t_2 \mapsto_{\bar{v}} t_1'\ t_2}$$
$$\frac{t \mapsto_{\bar{v}} t'}{a_v\ t \mapsto_{\bar{v}} a_v\ t'} \qquad \frac{t \mapsto_{\bar{v}} t'}{b\ t \mapsto_{\bar{v}} b\ t'}$$
$$\frac{t_2 \notin \mathit{Val},\ t_2 \mapsto_{\bar{v}} t_2'}{(\lambda x.t)\ t_2 \mapsto_{\bar{v}} (\lambda x.t)\ t_2'} \qquad \frac{t \mapsto_{\bar{v}} t'}{(\lambda x.t)\ h_v \mapsto_{\bar{v}} (\lambda x.t')\ h_v}$$

The first row of rules and the first rule of the second correspond to the rules for the standard reduction function for the λ-calculus. In an application, the left sub-term t_1 is reduced to either a value or a non-value normal form h_v. The second row of rules gives that in case of not obtaining an abstraction the right sub-term t_2 of the application is reduced which eventually overall yields a non-value normal form or a δ-redex. The third row gives that in the case where an abstraction is obtained, t_2 is reduced to either a value or a non-value normal form. In case a value is obtained the overall application is a β_v-redex and if a non-value normal form h_v is obtained, the body of the abstraction is reduced.

Like for the λ-calculus we notice that equality in the λ_v-calculus has not changed by the above definition of the restricted compatibility closure. Taking $\mapsto_{\bar{v}}^*$ as the reflexive transitive closure of $\mapsto_{\bar{v}}$, $=_{\bar{v}}$ denotes the equivalence over $\mapsto_{\bar{v}}^*$. Now $=_{\bar{v}}$ is a proper subset of $=_{\beta_v\delta}$. $=_{\bar{v}}$ is a subset because it is a restricted compatibility closure, and the term $(\lambda x.w(xx))(\lambda x.w(xx))((\lambda x.y)x)$ illustrates that this subset is proper: On the one hand an equation in the λ_v-calculus reads

$$(\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))\ ((\lambda x.y)\ x) \quad =_{\beta_v\delta} \quad (\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))\ y$$

which is established by a contraction of the redex $(\lambda x.y)\ x$. On the other hand $=_{\bar{v}}$ does not contain that equation because the left sub-term $(\lambda x.w\,(x\,x))\,(\lambda x.w\,(x\,x))$ must be reduced to a value or a non-value normal form before considering the right sub-term. But as seen above the left sub-term diverges. Hence $=_{\bar{v}} \subset =_{\beta_v\delta}$.

3.3.2 Normalization

A standardization theorem also holds for $\mapsto_{\bar{v}}$ with respect to the λ_v-calculus. In the previous subsection it was illustrated that the set difference $=_{\beta_v\delta} \setminus =_{\bar{v}}$ is non-empty. The theorem implies that these equations only consider terms with no normal forms (which is also the crucial property of the above sample term). A term is equal to a normal form if and only if the normal form can be obtained via a series of reductions following the standard reduction $\mapsto_{\bar{v}}$:

Theorem 3 (Standardization)
$t \rightarrow^*_{\beta_v\delta} n_v$ *if and only if* $t \mapsto^*_{\bar{v}} n_v$.

Equations involving terms with normal forms are all contained in both $=_{\beta_v\delta}$ and $=_{\bar{v}}$.

Because $\mapsto_{\bar{v}}$ is a function and the standardization theorem holds, $\mapsto_{\bar{v}}$ defines a strategy that by a series of reductions eventually yields the normal form of a term if one exists. Normalization in the λ_v-calculus can therefore be defined as follows.

$$\textit{normalize}_{\bar{v}}\ t = n_v \quad \text{iff} \quad t \mapsto^*_{\bar{v}} n_v$$

This function is well-defined and partial for the same reasons as $\textit{normalize}_{\bar{n}}$ in the λ-calculus. By construction the following proposition is immediate.

Proposition 3
$\textit{normalize}_{\bar{v}}\ t = n_v \iff t =_{\beta_v\delta} n_v$

3.3.3 Reducing to weak head normal forms

Normalization in the λ_v-calculus as defined in Section 3.3.2 following the standard reduction strategy presented in Section 3.3.1 yields $\beta_v\delta$-normal forms. We again introduce weak head normal forms as a relaxed notion of normal forms together with a corresponding reduction strategy to facilitate a weak notion of normalization:

$$\begin{array}{lrcl} & h_v & ::= & a_v\ t \mid (\lambda x.t)\ h_v \mid l\ t \mid q\ d_v \\ & a_v & ::= & x \mid h_v \\ & d_v & ::= & a_v \mid \lambda x.t \mid q \\ \textit{WHNForm}_v & w_v & ::= & d_v \mid l \end{array}$$

Remark the relationship with the grammar of normal forms on page 40: The only difference is all normal forms n_v have been relaxed to general terms t.

The strategy $\mapsto_v$ is defined for all terms not in weak head normal form. Compared to the definition of $\mapsto_{\bar{v}}$ the rules are simplified greatly when considering weak normalization:[1]

$$\frac{(t, t') \in \beta_v\delta}{t \mapsto_v t'} \qquad \frac{t_1 \mapsto_v t_1'}{t_1\ t_2 \mapsto_v t_1'\ t_2}$$

$$\frac{t_2 \mapsto_v t_2'}{(\lambda x.t)\ t_2 \mapsto_v (\lambda x.t)\ t_2'} \qquad \frac{t_2 \mapsto_v t_2'}{q\ t_2 \mapsto_v q\ t_2'}$$

[1]The two rules in the second row is sometimes replaced by the rule

$$\frac{t_2 \mapsto_v t_2'}{v\ t_2 \mapsto_v v\ t_2'}$$

For example, Plotkin's *left reduction* [59, page 136] would in our notation use that rule. Under certain conditions the difference become minor, but remark that reduction would also be defined for some weak head normal forms.

As in Section 1.5.3 the rule for reducing abstractions is removed. $t_1 \notin \mathit{Val}$ is implied by $t_1 \mapsto_v t_1'$ and is removed as a condition in the second rule. Also we only consider reducing the right sub-term of an application if the left sub-term is an abstraction or a basic function q and again, $t_2 \mapsto_v t_2'$ implies $t_2 \notin \mathit{Val}$ and the overall term $(\lambda x.t)\, t_2$ in the third rule is not a β_v-redex.

With $\beta_v\delta$ being a function it is justified that the function $\mapsto_v$ is well-defined. Normalization to weak head normal forms can hence be given in terms of the reflexive transitive closure of $\mapsto_v$, denoted $\mapsto_v^*$:

$$\mathit{normalize}_v\ t = w_v \quad \text{iff} \quad t \mapsto_v^* w_v$$

This function is well-defined and partial. Like in the λ-calculus, uniqueness of weak head normal forms of terms does not hold. The term $(\lambda x.x)\,(\lambda y.(\lambda z.y)\,w)$ is again a counter example. $\mathit{normalize}_v$ chooses among the possible weak head normal forms, the first one obtained when reduction is following the standard reduction strategy.

Analogously with weak head normalization in the λ-calculus a weak version of Proposition 3 holds:[2]

Proposition 4
(i) $\mathit{normalize}_v\ t = w_v \implies t =_{\beta_v\delta} w_v$
(ii) $t =_{\beta_v\delta} w_v \implies \mathit{normalize}_v\ t = w_v'$, where $w_v =_{\beta_v\delta} w_v'$

Especially, $\mathit{normalize}_v\ t = b \iff t =_{\beta_v\delta} b$.

3.4 Summary

In this chapter we introduced variations in the defining elements of the λ-calculus. The variations led to new notions of equality of λ-terms. The bigger part of the chapter was concerned with Plotkin's λ_v-calculus. We defined reduction strategies in the λ_v-calculus for both weak and strong normalization.

[2]The right-to-left implication of the proposition again (like for weak head normalization in the λ-calculus) hinges on a lemma involving standard reduction sequences.

Chapter 4

Programming languages

In this chapter we give an introduction to programming languages.

Roadmap We briefly introduce a distinction between high-level programming languages and low-level programming languages (Section 4.1) and various kinds of programming-language classifications (Section 4.2). With programming languages defined as syntax together with a formal 'meaning' of syntax or *semantics* (Section 4.3) we briefly touch upon syntax (Section 4.4) and continue with the main part of this chapter concerning semantics (Section 4.5). We informally introduce a tiny sample programming language **TLLF** and introduce three kinds of semantics artifacts by giving formal semantics for **TLLF**: We introduce *denotational semantics* in Section 4.5.1, *abstract machines* in Section 4.5.2, and *reduction semantics* in Section 4.5.3.

4.1 Machine-level and high-level programming languages

When a computer is used to perform some kind of calculation, the operations the machine must perform are stated via executable programs in a *machine-level programming language* that is understood by the computer. Programs in a machine level language are hardly human readable — even in symbolic form. A machine-level program includes a list of labeled instructions directly executable by the computer. When the computer executes one instruction it unambiguously know how to perform that instruction. The instructions consists of, e.g., moving data from one register to another, performing a simple arithmetic calculation or logic operation, and conditional jumping to another place in the list of instructions depending on the content of a register. Computers with such an instruction set are Turing complete, i.e., an algorithm to solve every mathematically computable problem can be expressed via the instruction set. But, because every computer only understand one such machine-level language and because such languages are hardly human readable even in moderate size, *high-level programming languages* have been invented. This makes the programming languages machine-independent and the higher abstraction level lets programs be specified in a human readable language. Here the languages are 'high-level' in the sense that the constructs of the languages specify rules for state changes that do not correspond one-one to primitive operations on the machine. Of course programs in high-level languages must be translated into the machine language before execution. For some high-level languages this translation is done as a batch job with a *compiler* before the execution. For other languages an *interpreter* in a loop translates one construct of the program, executes the corresponding low-level machine instructions and continues with the next construct. Other languages again use a hybrid, where the high-level program as a batch job is translated to a program interpreted by a *virtual machine*.

4.2 Paradigms

High-level programming languages can be grouped according to various aspects. Some languages are *general purpose* languages. Others are more *domain specific* by including specialized constructs wrt. some domain. Some languages are *object oriented*, some are not.

Another — for this context more important — way to group high-level programming languages is by the central unit of constructs. In *imperative* programming languages the central unit is the statement. Routines (or methods) are used to make compound statements. A program in such a language is intuitively a specification of an execution of a program on a machine. Computation is the sequential execution of individual statements which changes the state of the machine.

In *logic/constraint* programming languages the central unit is the relation. Relations define the constraints on variables. Computation is then the unification that overall tries to make the relations hold by assigning values to variables. Emphasis is on the constraints to fulfil and not the sequence.

In *functional* programming languages the central unit is the expression. Expressions are evaluated yielding values and functions are used to abstract evaluation. In a purely functional language (i.e., no imperative constructs are included) *referential transparency* is guaranteed. That is, any expression can be exchanged with its value without changing the meaning of the program. In other words, the order of evaluation is irrelevant (when not considering possible nontermination). Computation is the evaluation of expressions.

4.3 Defining a programming language

A programming language consists of (1) a definition of well-formed programs — the *syntax*; and, to be able to write a compiler or an interpreter for a programming language, (2) a definition of the exact meaning of the syntactic constructs — the *semantics*.

Quoting Strachey from his seminal paper *Fundamental Concepts in Programming Languages* [67, page 12]:

> In a rough and ready sort of way it seems to me fair to think of the semantics as being what we want to say and the syntax as how we have to say it.

For a machine-level language a machine with the corresponding instruction set actually is definitional for both syntax and semantics. In the following we concentrate on high-level languages and illustrate the concepts by example.

The illustration language The sample programming language is a functional language inspired by the $\lambda\beta\delta$-calculus from Section 2.8. The language includes variables, first-class expression abstractions and applications of such abstractions. Furthermore, literal integers and a primitive successor function for literals are included. Application is lazy in the sense that arguments to abstractions are not evaluated before application. The scope of a binding of a variable in the application of an abstraction is the (lexical) body of the abstraction. We will refer to this **T**iny, **L**azy, **L**exically scoped **F**unctional language by **TLLF**.

4.4 Syntax

For high-level languages the syntax of a programming language is usually specified by a grammar of constructions like the specification on page 5 of terms in the λ-calculus. Usually though, *concrete syntax* is defined which specifies actual representations of programs as strings of characters. A compiler uses a *parser* to map concrete syntax to abstract syntax. The logical information about the used constructs and grouping is preserved in the abstract syntax tree. What has been removed is the linear structure and with that the tokens and precedence rules needed to unambiguously separate the constructs. Quoting David Schmidt [65, page 8]:

> We claim that the derivation trees [abstract syntax trees] are the *real* sentences of a language, and strings of symbols are just abbreviations for the trees.

Abstract syntax of TLLF The abstract syntax of **TLLF** coincide with that of the λβδ-calculus and the λ_v-calculus except wrt. variables:

$$\begin{array}{llll} \textit{Var} & i & & \{1, 2, 3, \ldots\} \\ \textit{Exp} & e & ::= & i \mid \lambda e \mid e\,e \mid b \\ \textit{Basic} & b & ::= & l \mid q \\ \textit{Lit} & l & ::= & \{\ulcorner 0 \urcorner, \ulcorner 1 \urcorner, \ulcorner 2 \urcorner, \ulcorner 3 \urcorner, \ldots\} \\ \textit{BasicF} & q & ::= & \mathtt{S} \end{array}$$

Because the language is lexically scoped we let variable names be part of concrete syntax. Hence variables are represented by lexical offsets relative to the occurrence. A program is a closed expression, where an expression is closed if no lexical offset exceeds the number of nesting abstractions from the occurrence to the root of the expression. Formally, $0 \vdash e$ must hold where:

$$\frac{i \leq m}{m \vdash i} \qquad \frac{m+1 \vdash e}{m \vdash \lambda e} \qquad \frac{m \vdash e, \; m \vdash e'}{m \vdash e\,e'} \qquad \frac{}{m \vdash b}$$

4.5 Semantics

Specifying the semantics of a high-level programming language is more challenging and various approaches exists. Some languages have only an informal description in prose of the semantics of the syntactic language constructs. In that case the absence of (semantic) ambiguities is hard to ensure and a (black-box) compiler or (black-box) interpreter then becomes the ultimate definitional reference. To prevent ambiguities and to be able to reason about programs in a language, e.g., to prove soundness of optimizations in a compiler, a formal specification of the semantics of a language is usually presented via a *semantic artifact*.

The two main approaches result in a *denotational semantics* [64, 65] and an *operational semantics* [56, 60], respectively. In the denotational approach the semantics is specified as a function mapping programs to their meaning, which might very well be a function of concepts like an environment, a store, and a continuation. This approach is *extensional* in the sense that the semantics of a program is solely a function on some input.

The operational approach is more closely related to the intuitive notion of a program as a specification of a program execution found in imperative languages. By this approach rules are specified to transform a program to a result in a series of steps and is for that reason said to be *intentional*.

In the following, three semantic artifacts are introduced for **TLLF**. We begin with a denotational semantics, and continue with two operational semantics: an abstract machine and a reduction semantics. In each case we introduce the kind of semantic specification before giving a semantics for **TLLF**.

4.5.1 Denotational semantics

This approach relies on domain theory [66] [65, Chapter 3] [64, Chapter 2] [70, Chapter 8].

Semantic domains and semantic algebras A *semantic domain* is a set of objects D with a partial order $\sqsubseteq$, where all possible chains $d_1 \sqsubseteq d_2 \sqsubseteq \dots$, for $d_i \in R \subseteq D$ have a least upper bound $\bigsqcup R$ in D. Such a construction is also called a *cpo* (Complete Partial Order). Here we follow Schmidt and Winskel. Reynolds refers to such a construction as a *predomain*. If D also contains a least element $\bot$, 'less than' all other objects in the set ($\bot \sqsubseteq d, \forall d \in D$), D is called a *pointed cpo*. Reynolds refers to such a construction as a *domain*.[1]

A *semantic algebra* is a domain together with related operations like operations to create elements, predicates on elements, etc. The operations are the 'interface' to the elements encapsulating the 'representation' of the elements.

Compound domains and algebras are build via a set of constructions out of existing domains. The most used such constructions are the product construction $\times$, the disjoint union construction $+$ and the function space construction $\rightarrow$.[2] The theory gives that via standard partial orders the results are all domains. When using the compound constructions corresponding operations are defined implicitly as part of the construction. For example, when using the disjoint union construction $+$, injection functions into the sub-domains and predicate functions indicating which sub-domain an element belongs to, are defined as part of the construction and left implicitly when actually forming compound domains.[3]

[1]The natural numbers $\mathbb{N}$ with the usual $\leq$ ordering as the partial order is not even a predomain because there exists chains where the least upper bound is not a natural number (e.g., $1 \leq 2 \leq \dots$ with least upper bound ∞). Adding ∞ actually defines a domain with $\bot = 1$. Usually though, $\bot$ will represent nontermination and the partial order represents the degree of information. By treating $\mathbb{N}$ as a predomain with the trivial partial order $\sqsubseteq$ where $n \sqsubseteq n'$ iff $n = n'$, we can define $\mathbb{N}_\bot := \mathbb{N} \cup \{\bot\}$ and extend the partial order to include $\bot \sqsubseteq n, \forall n \in \mathbb{N}$. The result is also a domain: the *flat* domain (or *discrete* domain) denoted by $\mathbb{N}_\bot$.

[2]Starting, e.g., with the discrete domain of natural numbers $\mathbb{N}_\bot$ one can via the product construction $\times$ construct the product domain $\mathbb{N}_\bot \times \mathbb{N}_\bot$.

[3]The theory gives that these functions are all *continuous* with respect to the partial order. Continuous in this context means that the partial order is preserved under the mapping (*monotonicity*) and that for every chain the least upper bound is mapped to the least upper bound of the chain after the mapping. When using the construction to build a function space $A \rightarrow B$ the domain consists of all the continuous functions from A to B. That only the continuous functions are included makes it possible to create recursive domains without introducing a cardinality problem.

The fixed-point theorem establishes the foundation for recursive functions. Having a recursive equation of a function between domains only using continuous functions, the function it represents is the limit of approxima-

A metalanguage To denote the operations in the primitive and the compound semantic algebras a metalanguage is needed. This language is written in λ-notation and often includes a conditional construct ($\cdot \to \cdot \mid \cdot$) and a case construct (*case* $\cdot$ *of* $(\cdot \to \cdot)^*$).

The meaning of the metalanguage used is explained differently. Schmidt explicitly gives simplification properties for the individual meta-constructs. Reynolds refers informally, e.g., to the conditional construct, as a way to define the function 'by cases'. Winskel defines a conditional construct in a defined language by use of sets directly instead of a metacircular use of a conditional construct in the metalanguage. From a denotational perspective the important thing is that the metalanguage is only used to describe mathematical objects like functions — the representation of the objects is irrelevant.

A central property of a denotational semantics is hence its descriptive nature: A denotational semantics is a *specification* of the language. In that respect simplification rules for the metalanguage is misleading as the denotation is the same mathematical object after a series of simplifications.[4]

The valuation function When defining the meaning (*denotation*) of a program in a programming language via a denotational semantics, the metalanguage is used. A *valuation function* maps well-formed terms to their corresponding meaning in a semantic domain. This valuation function is normally defined via semantic equations and must be *syntax-directed* and *compositional* [70, page 60]: Syntax-directed means that there is exactly one equation for each production in the abstract-syntax grammar. Compositional means that in each of these semantic equations, the meaning of a compound construction is expressed solely as a function on the meaning of its immediate sub-constructions.

A denotational semantics maps programs that are considered equal in the defined language to the same semantic object — they have the same denotation. According to that reasoning the valuation function $E[\![\cdot]\!]_{\mathbf{TLLF}}$ (defined below) defines equivalence classes on the set of all programs:

$$p =_{\mathbf{TLLF}} p' \quad \text{iff} \quad E[\![p]\!]_{\mathbf{TLLF}} = E[\![p']\!]_{\mathbf{TLLF}}$$

Suppose a program p correctly implements an algorithm for a problem and program p' maybe implements a solution more cleverly. It would then be useful to have a tool to make the decision $E[\![p]\!]_{\mathbf{TLLF}} = E[\![p']\!]_{\mathbf{TLLF}}$. Unfortunately, we can solve the halting-problem if such a decision-method exists by letting one of the programs be a nonterminating program. It is hence (in general) undecidable do check for equivalence.

A denotational semantics for TLLF

We define the needed semantic algebras used by the valuation function and the valuation function itself. Denotations of terms rely on an algebra of environments which again relies on a notion of denotable values (*Basic* is the sum cpo of the discrete cpos of basic functions

tions of the function. This limit function is continuous. This result is used to give a compositional specification of, e.g., looping constructs in imperative languages and constructs for recursive declarations in functional languages.

[4] If one decide on a strategy to apply the simplification rules one has an *implementation* of the language. We will elaborate further on that later.

BasicF and literals *Literal*):

$$
\begin{array}{lrcl}
\text{Domains :} & \nu \in \mathit{Val} & := & \mathit{Basic} + \mathit{Fun} \\
& d \in \mathit{DenVal} & := & \mathit{Val}_\perp \\
& f \in \mathit{Fun} & := & \mathit{DenVal} \to \mathit{DenVal} \\
& \rho \in \mathit{Env} & := & \{\mathit{mt\_env}\} \cup \mathit{DenVal} \times \mathit{Env} \\
\text{Operations :} & \mathit{mt\_env} & : & \mathit{Env} \\
& \mathit{extend\_env} & : & \mathit{Env} \to \mathit{DenVal} \to \mathit{Env} \\
& \mathit{extend\_env} & = & \lambda\rho.\lambda d.(d,\ \rho) \\
& \mathit{lookup} & : & \mathit{Env} \to \mathit{Var} \to \mathit{DenVal} \\
& \mathit{lookup} & = & \lambda\rho.\lambda i.\mathit{case}\ \rho\ \mathit{of} \\
& & & \qquad\qquad \mathit{mt\_env} \to \perp \\
& & & \qquad\qquad (d,\ \rho) \to (i = 1) \to d \mid (\mathit{lookup}\ \rho\ (i-1))
\end{array}
$$

Environments are represented by nested pairs. *mt_env* is the constant representing the empty list. We leave out the injection functions and projection functions. *DenVal* is a *lifting* of the predomain of values *Val*. To model call-by-name evaluation of applications the function space *Fun* cannot be defined as $\mathit{Val} \to \mathit{DenVal}$. In that case the result is an *eager* functional language that requires arguments in applications to be proper values. The valuation function $E[\![\cdot]\!]_{\mathbf{TLLF}}$ is syntax-directed and compositional as required:

$$
\begin{array}{rcl}
E[\![\cdot]\!]_{\mathbf{TLLF}} & : & \mathit{Exp} \to \mathit{Env} \to \mathit{DenVal} \\
E[\![i]\!]_{\mathbf{TLLF}} & = & \lambda\rho.\mathit{lookup}\ \rho\ i \\
E[\![\lambda e]\!]_{\mathbf{TLLF}} & = & \lambda\rho.\mathit{inVal}\ (\mathit{inFun}\ (\lambda d.E[\![e]\!]_{\mathbf{TLLF}}\ (\mathit{extend\_env}\ \rho\ d))) \\
E[\![e\ e']\!]_{\mathbf{TLLF}} & = & \lambda\rho.\mathit{case}\ E[\![e]\!]_{\mathbf{TLLF}}\ \rho\ \mathit{of} \\
& & \qquad \mathit{isVal}\ (\mathit{isFun}\ f) \to f\ (E[\![e']\!]_{\mathbf{TLLF}}\ \rho) \\
& & \qquad \mathit{isVal}\ (\mathit{isBasic}\ (\mathit{isBasicF}\ q)) \to \mathit{case}\ E[\![e']\!]_{\mathbf{TLLF}}\ \rho\ \mathit{of} \\
& & \qquad\qquad \mathit{isVal}\ (\mathit{isBasic}\ (\mathit{isLiteral}\ l)) \\
& & \qquad\qquad\qquad \to \mathit{inVal}\ (\mathit{inBasic}\ (\mathtt{capp}\ (q,\ l))) \\
& & \qquad\qquad \mathit{else} \to \perp \\
& & \qquad \mathit{else} \to \perp \\
E[\![b]\!]_{\mathbf{TLLF}} & = & \lambda\rho.\mathit{inVal}\ (\mathit{inBasic}\ (B[\![b]\!]_{\mathbf{TLLF}})) \\
\\
B[\![l]\!]_{\mathbf{TLLF}} & = & \mathit{inLiteral}\ l \\
B[\![q]\!]_{\mathbf{TLLF}} & = & \mathit{inBasicF}\ q
\end{array}
$$

The denotation of an abstract syntax tree of the programming language is a function from environments to denotable values. The specification is total because of the *else*-cases. Both if a program contains a 'type error' (e.g., if the denotation of the left sub-term of an application yields a constant which is not a basic function) or if it diverges the denotation is mapping every environment to $\perp$. In the denotation of applications the denotation of the argument expression is applied to the argument environment. Also the body of abstractions is evaluated in the argument environment (augmented with a new binding).

4.5.2 Abstract machines

The SECD machine To define the semantics of applicative expressions (AE), Landin designed the SECD machine [50]. As Landin later points out [51, page 159] the abstract language ISWIM (If you See What I Mean) is AE with 'syntactic sugar'. That is, AE is the 'kernel' of ISWIM: Every construct of ISWIM can be expressed in AE.

At any point in time the SECD machine's state or *configuration* is a four-tuple: $(\mathbf{S},\ \mathbf{E},\ \mathbf{C},\ \mathbf{D})$. This machine performs evaluation of applicative expressions by looking at the four components and *deterministically* determining the four components of the next configuration. Iteratively performing this mapping from configuration to configuration (possibly) obtaining a configuration, which cannot be mapped to another configuration, defines evaluation of applicative expressions via the SECD machine. That is, the SECD machine defines the semantics of applicative expressions, via an abstract specification of configurations and a definition of transitions from configurations to configurations. In that respect, this approach is a *small-step* operational semantics.[5]

Variations of the SECD machine The SECD machine is mostly important because it is the first abstract machine defined as a definitional semantics for a programming language. The original specification is not *properly tail-recursive* [12] and Danvy et al. show that the dump component $\mathbf{D}$ (which is isomorphic to a stack of $(\mathbf{S},\ \mathbf{E},\ \mathbf{C})$-triples) is an unnecessary artifact [3].

Directly simplified or extended versions also exist. For example, without support for basic constants or with support for non-local control operators [3, 25]. Hannan and Miller define such a version without support for basic functions [39]. That machine differs from the original version in a more subtle way: In the evaluation of applications the operator subexpression is evaluated before the operand subexpression resulting in *left-to-right* applicative evaluation order whereas the original has *right-to-left* applicative evaluation order. The machine also operates on expressions with de Bruijn indices instead of named variables.

Landin's original version operates directly on applicative expressions. Felleisen and Flatt have a version where such expressions must be compiled into a language of instructions before running the machine [31, Section 8.1].

Plotkin allows closures involving expressions in general instead of only abstractions [59, page 120]. Plotkin has this more general notion of closures to easily define a call-by-name version of the SECD machine.

Other machines Felleisen and Friedman designed another machine called the CEK machine [32]; a simple machine inspired by the SECD machine but with roots in Reynolds's work on definitional interpreters [32, Section 2] [61]. Several variants of the CEK machine have been presented [31]. One of these variants is the CESK machine which was presented in Felleisen's PhD-dissertation as a specification of the semantics of what he calls *Idealized Scheme* [30]. The added component $\mathbf{S}$ represents a store needed to model the language: Idealized Scheme is essentially applicative expressions extended with imperative constructs for side-effecting variables and a non-local control operator.

A wide range of machines has been presented in the literature. Some other machines are Hannan and Miller's CLS machine, the VEC machine, which was presented as an implementation for a denotational semantics [65], the never published Krivine machine [13], and the CAM with origin in categorical combinators [16].

Abstract machines and virtual machines Ager et al. distinguish between *abstract* machines and *virtual* machines [2]. According to their definition a machine is a virtual machine if and

[5] Reynolds refers to semantics of this kind by *transition semantics* [64, Chapter 6].

only if it operates on compiled terms. In that respect, e.g., Felleisen and Flatt's version of the SECD machine, the VEC-machine and the CAM are virtual machines and the SECD machine and the Krivine machine are abstract machines. We follow the distinction suggested by Ager et al. and concentrate on machines operating directly on abstract syntax.

Our notion of an abstract machine An abstract machine M is a state-transition system. It has a set of states (or *configurations*) partitioned into the *terminal configurations* ($\mathit{Terminal}_M$) and the *non-terminal configurations* ($\mathit{NonTerminal}_M$). It has a *transition relation* (σ_M) that decides the next configuration for a given non-terminal configuration. The transition relation is iterated until (possibly) a terminal configuration is obtained.

Following Plotkin [60] we have restricted σ_M to be a (total) function such that the machine is deterministic. We restrict σ_M further by only allowing elementary operations. This restriction makes the evaluation steps explicit. For example, a recursive traversal on a term is then not allowed in one transition.[6]

A total function $\mathtt{load}_M$ is specified to map a term into a corresponding start configuration of M. Likewise, a partial function $\mathtt{unload}_M$ is specified to extract the result from a terminal configuration. Running an abstract machine is then defined as the composition of loading, iterating the transition function, and possibly unloading the final result:

$$
\begin{array}{rrcl}
\mathit{Terminal}_M & a & & \\
\mathit{NonTerminal}_M & n & & \\
\mathit{Configuration}_M & c & ::= & a \mid n
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{load}_M & : & \mathit{Term} \to \mathit{Configuration}_M \\
\sigma_M & : & \mathit{NonTerminal}_M \to \mathit{Configuration}_M \\
\mathtt{unload}_M & : & \mathit{Terminal}_M \to \mathit{Val}
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{iterate}_M & : & \mathit{Configuration}_M \to \mathit{Terminal}_M \\
\mathtt{iterate}_M\ a & = & a \\
\mathtt{iterate}_M\ n & = & \mathtt{iterate}_M\ (\sigma_M\ n)
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{run}_M & : & \mathit{Term} \to \mathit{Val} \\
\mathtt{run}_M & = & \mathtt{unload}_M \circ \mathtt{iterate}_M \circ \mathtt{load}_M
\end{array}
$$

Here *Val* is the set of result values. If an abstract machine M on term t never reaches a terminal configuration, the machine is said to *diverge on* t. If a terminal configuration a is reached but $\mathtt{unload}_M$ is not defined on a, M is said to be *stuck* in a.

The above specification implies that an abstract machine M is completely specified by defining the possible terminal and nonterminal configurations and the three functions: $\mathtt{load}_M$, σ_M and $\mathtt{unload}_M$.

As defined above an abstract machine may operate directly on terms by letting terms be part of each configuration. But there is no constraint implying that such an explicit use of terms is always the case. For example, by letting the load function be a compiler the result is a virtual machine. For that reason we restrict loading and unloading as well to only perform elementary operations.

[6] Felleisen and Flatt do not restrict the transition relation to be a function and they also do not restrict that function by only allowing elementary operations. For example, the CC machine performs a recursive descent on certain terms in one transition [31, Chapter 6].

An abstract machine for TLLF

A configuration of the abstract machine consists of three parts: (1) an expression from the source language, (2) an environment mapping lexical offsets to closures, and (3) a stack where each element is either a closure or a basic function:

$$
\begin{array}{llcl}
\textit{Closure} & c & ::= & (e,\ \rho) \\
\textit{Env} & \rho & ::= & \bullet \mid c \cdot \rho \\
\textit{Stack} & s & ::= & \bullet \mid c \cdot s \mid q \cdot s \\
\textit{Val} & v & ::= & b \mid (\lambda e, \rho) \\
\textit{Configuration} & & = & \textit{Exp} \times \textit{Env} \times \textit{Stack}
\end{array}
$$

Loading the machine is defined as forming the configuration of the initial expression and the empty environment and the empty stack. Unloading consists of extracting the expression part when the stack is empty. If the expression is an abstraction it is paired with the environment which binds the free indices of the body of the abstraction:

$$
\begin{array}{rcl}
\texttt{load}_{\mathbf{TLLF}} & : & \textit{Exp} \rightarrow \textit{Configuration} \\
\texttt{load}_{\mathbf{TLLF}}\, e & = & (e,\ \bullet,\ \bullet) \\
\\
\texttt{unload}_{\mathbf{TLLF}} & : & \textit{Configuration} \rightarrow \textit{Val} \\
\texttt{unload}_{\mathbf{TLLF}}\, (b,\ \rho,\ \bullet) & = & b \\
\texttt{unload}_{\mathbf{TLLF}}\, (\lambda e,\ \rho,\ \bullet) & = & (\lambda e,\ \rho)
\end{array}
$$

The transition function from (non-terminal) configurations to configurations reads:

$$
\begin{array}{rcl}
\sigma_{\mathbf{TLLF}} & : & \textit{Configuration} \rightarrow \textit{Configuration} \\
\sigma_{\mathbf{TLLF}}\, (i,\ c_1 \cdots (e_i, \rho_i) \cdots c_j,\ s) & = & (e_i,\ \rho_i,\ s) \\
\sigma_{\mathbf{TLLF}}\, (\lambda e,\ \rho,\ c \cdot s) & = & (e,\ c \cdot \rho,\ s) \\
\sigma_{\mathbf{TLLF}}\, (e\, e',\ \rho,\ s) & = & (e,\ \rho,\ (e', \rho) \cdot s) \\
\sigma_{\mathbf{TLLF}}\, (q,\ \rho,\ (e, \rho') \cdot s) & = & (e,\ \rho',\ q \cdot s) \\
\sigma_{\mathbf{TLLF}}\, (l,\ \rho,\ q \cdot s) & = & (\texttt{capp}\, (q,\ l),\ \rho,\ s)
\end{array}
$$

Because programs are closed terms a variable can always find a corresponding binding to a closure in the environment. In case of a match, the expression and the associated environment is used for evaluation. Closures are shifted from the stack to the environment when the term-part is an abstraction. The associated environment is the application-time environment and is associated an unevaluated expression. The call-by-name application mechanism and static scope of bindings is hence achieved. When the left sub-expression of an application is evaluated to a basic function, the argument expression must be evaluated to a literal.

The definition of $\sigma_{\mathbf{TLLF}}$ defines the set of terminal configurations. Because $\texttt{unload}_{\mathbf{TLLF}}$ is partially defined on this set the machine can get stuck. Stuck configurations correspond to situations where the left subexpression of an application evaluates to a literal, or a basic function is applied to a literal where `capp` is not defined. The machine can diverge because of self-application.

The above abstract machine gives the semantics for **TLLF**. The machine is a direct extension of a well-known abstract machine: the Krivine machine. The Krivine machine performs call-by-name evaluation of pure de Bruijn-indexed λ-terms. The extension thus consists in the support for basic constants: The first rules of $\texttt{unload}_{\mathbf{TLLF}}$ and the last two rules of $\sigma_{\mathbf{TLLF}}$ are not part of the Krivine machine.

Adjusted representation By (lightweight) fusing the iterating function with the transition function another useful representation of abstract machines is achieved. The fusion consists in letting the transition function call itself after each single transition. Inlining the unload function lets the machine directly unload the result from a terminal configuration. The auxiliary structures remain unchanged, and running the extended Krivine machine from above is equivalently expressed:

$$
\begin{array}{rcl}
\sigma_{\mathbf{TLLF}} & : & \mathit{Configuration} \to \mathit{Val} \\
\sigma_{\mathbf{TLLF}}\,(b,\ \rho,\ \bullet) & = & b \\
\sigma_{\mathbf{TLLF}}\,(\lambda e,\ \rho,\ \bullet) & = & (\lambda e,\ \rho) \\
\sigma_{\mathbf{TLLF}}\,(i,\ c_1 \cdots (e_i, \rho_i) \cdots c_j,\ s) & = & \sigma_{\mathbf{TLLF}}\,(e_i,\ \rho_i,\ s) \\
\sigma_{\mathbf{TLLF}}\,(\lambda e,\ \rho,\ c \cdot s) & = & \sigma_{\mathbf{TLLF}}\,(e,\ c \cdot \rho,\ s) \\
\sigma_{\mathbf{TLLF}}\,(e\,e',\ \rho,\ s) & = & \sigma_{\mathbf{TLLF}}\,(e,\ \rho,\ (e', \rho) \cdot s) \\
\sigma_{\mathbf{TLLF}}\,(q,\ \rho,\ (e, \rho') \cdot s) & = & \sigma_{\mathbf{TLLF}}\,(e,\ \rho',\ q \cdot s) \\
\sigma_{\mathbf{TLLF}}\,(l,\ \rho,\ q \cdot s) & = & \sigma_{\mathbf{TLLF}}\,(\mathtt{capp}\,(q,\ l),\ \rho,\ s) \\
\\
\mathtt{run}_{\mathbf{TLLF}} & : & \mathit{Exp} \to \mathit{Val} \\
\mathtt{run}_{\mathbf{TLLF}}\,e & = & \sigma_{\mathbf{TLLF}}\,(e,\ \bullet,\ \bullet)
\end{array}
$$

In the following we will often use this alternative representation of abstract machines. In each situation it will be a straightforward exercise to translate to the other representation. Danvy and Millikin go into details [18].

4.5.3 Reduction semantics

To specify a reduction semantics for a programming language with a given abstract syntax, consists in defining syntactic values, notions of reductions, and a reduction strategy [30,31].

Reduction Contexts The strategy is defined via a grammar of *reduction contexts*. As strategies are defined in Section 1.5, the use of a grammar for specifying the reduction contexts implies that a non-value syntactic unit can be uniquely decomposed into a reduction context and a *potential redex* [28, page 5] (which syntactically takes the form of a redex without necessarily being an actual redex), and that the notions of reduction can be specified as a function.[7] In that respect we follow Danvy and Nielsen [28].

When contexts are defined like in Section 1.3, contexts are represented 'outside-in'. As mentioned a context is a syntactic unit with one sub-unit replaced by a hole. Filling that hole consists of replacing it by a syntactic unit. This replacing is textual in the sense that no substitution of variables is used to avoid capturing of free variables of the syntactic unit. An often used alternative representation of contexts is 'inside-out', where contexts are isomorphic to syntactic units with a hole. Obtaining a syntactic unit from such a context and a syntactic unit is then not a textual replacing of the hole and an explicit definition is needed. In general we say that we *plug* a syntactic unit into a context.

[7] As explained in section 1.5 normal-order reduction cannot easily be specified as taking the compatible closure of the notion of reduction according to a grammar of contexts, because some terms not in normal form can not be uniquely decomposed into a context and a redex. In Part II we elaborate on this point.

A one-step reduction function A one-step reduction function $\mapsto$ must, if it is defined on a non-value syntactic unit u, perform three tasks:[8]

(i) *decompose* u (uniquely) into a reduction context and a (actual) redex,

(ii) *contract* the redex following the contraction rules, and

(iii) *plug* the contractum into the context, yielding another syntactic unit u'.

$\mapsto$ is partial because it is not defined on values but also if not all potential redexes are also actual redexes.

Usually the above three 'tasks' of the one-step reduction function are specified less explicitly via a set of rules with one rule for each contraction rule in the notion of reduction: $E[r] \mapsto E[r']$ denotes a rule for the *unique decomposition* of the expression $E[r]$ into the context E and the redex r, the contraction of the redex obtaining the contractum r' and the plugging of that contractum into E obtaining $E[r']$. Such a specification is well-defined because of the assumed unique-decomposition property and because the notion of reduction defines a function. The rules can be seen as exploiting an advanced kind of pattern matching. Expressions matched by the left-hand 'pattern' is the expressions that are the result of plugging the specified kind of redex into the specified kind of context. The unique-decomposition property gives that for a given expression, it can be matched in at most one way by this pattern.

Evaluation Evaluation is defined in terms of the reflexive transitive closure of the one-step reduction function $\mapsto^*$. Evaluation is defined for a syntactic unit u iff $\mapsto^*$ relates u to a value. The result is then the value. Because $\mapsto$ is a function and it is not defined on values, evaluation is well-defined. It is partial if $\mapsto$ is not defined for some non-value syntactic units or if an infinite number of one-step reductions can be performed starting from some syntactic unit.

A reduction semantics for TLLF

The syntactic units in the reduction semantics for **TLLF** are closures. A subset of the closures are syntactic values:

Substitution	s	$::=$	$\bullet \mid c \cdot s$
Closure	c	$::=$	$e[s] \mid c\ c$
Value	v	$::=$	$(\lambda e)[s] \mid b[s]$

The values are the closures with an expression-part consisting of an abstraction or a basic constant.

Reduction contexts are represented 'outside-in' (i.e., contexts are closures with a hole and plugging a closure into a reduction context is a textual replacing of the hole):

EContext	E	$::=$	$[\,] \mid E\ c \mid q[s]\ E$

The last production represents a context with an application of a basic function, where the argument expression has to be evaluated prior to the application.

[8] The one-step reduction function is not defined on non-value terms which are uniquely decomposed into a context and a potential redex that is not an *actual* redex according to the notions of reduction.

The one-step reduction function for **TLLF** is given by four rules:

$$\begin{array}{rcl} \mathsf{E}[\mathsf{i}[\mathsf{c}_1 \cdots \mathsf{c}_i \cdots \mathsf{c}_j]] & \mapsto_{\mathbf{TLLF}} & \mathsf{E}[\mathsf{c}_i] \\ \mathsf{E}[(\lambda e)[\mathsf{s}]\ \mathsf{c}] & \mapsto_{\mathbf{TLLF}} & \mathsf{E}[e[\mathsf{c} \cdot \mathsf{s}]] \\ \mathsf{E}[(e\ e')[\mathsf{s}]] & \mapsto_{\mathbf{TLLF}} & \mathsf{E}[e[\mathsf{s}]\ e'[\mathsf{s}]] \\ \mathsf{E}[\mathsf{q}[\mathsf{s}]\ \mathsf{l}[\mathsf{s}']] & \mapsto_{\mathbf{TLLF}} & \mathsf{E}[(\mathtt{capp}\ (\mathsf{q},\ \mathsf{l}))[\mathsf{s}']] \end{array}$$

This function is partial: $\mapsto_{\mathbf{TLLF}}$ is not defined on values because they cannot be decomposed into a reduction context and a redex. Also $\mapsto_{\mathbf{TLLF}}$ is not defined on non-value closures that cannot be decomposed into a reduction context and a redex. For example, $\mapsto_{\mathbf{TLLF}}$ is not defined on closures on the form $\mathsf{l}[s]\ e[s']$.

As explained above evaluation of an expression e is defined if the reflexive transitive closure of the one-step reduction function relates e to a value:

$$\mathtt{eval}_{\mathbf{TLLF}}\ e = v \quad \text{iff} \quad e[\bullet] \mapsto^{*}_{\mathbf{TLLF}} v$$

Evaluation is well-defined because $\mapsto_{\mathbf{TLLF}}$ is not defined on values and evaluation is partial: There exist non-value expressions for which $\mapsto_{\mathbf{TLLF}}$ is not defined as exemplified above.

Adjusted representation The evaluation of an expression e consists in iteratively applying the one-step reduction function. Either $\mapsto_{\mathbf{TLLF}}$ can always be applied (e *diverges*), or an expression e' that is not mapped by $\mapsto_{\mathbf{TLLF}}$ is obtained. If e' is a value this value is the result. If e' is not a value e' is called a *stuck* expression. Following this explanation we equivalently write:

$$\begin{array}{rcl} \mathtt{eval}_{\mathbf{TLLF}} & : & \mathit{Exp} \to \mathit{Value} \\ \mathtt{eval}_{\mathbf{TLLF}}\ e & = & \mathtt{iterate}\ (e[\bullet]) \\ \\ \mathtt{iterate} & : & \mathit{Closure} \to \mathit{Value} \\ \mathtt{iterate}\, v & = & v \\ \mathtt{iterate}\ \mathsf{E}[\mathsf{i}[\mathsf{c}_1 \cdots \mathsf{c}_i \cdots \mathsf{c}_j]] & = & \mathtt{iterate}\ \mathsf{E}[\mathsf{c}_i] \\ \mathtt{iterate}\ \mathsf{E}[(\lambda e)[\mathsf{s}]\ \mathsf{c}] & = & \mathtt{iterate}\ \mathsf{E}[e[\mathsf{c} \cdot \mathsf{s}]] \\ \mathtt{iterate}\ \mathsf{E}[(e\ e')[\mathsf{s}]] & = & \mathtt{iterate}\ \mathsf{E}[e[\mathsf{s}]\ e'[\mathsf{s}]] \\ \mathtt{iterate}\ \mathsf{E}[\mathsf{q}[\mathsf{s}]\ \mathsf{l}[\mathsf{s}']] & = & \mathtt{iterate}\ \mathsf{E}[(\mathtt{capp}\ (\mathsf{q},\ \mathsf{l}))[\mathsf{s}']] \end{array}$$

In each step one reduction is performed according to the inlined contraction rule. This function is partial because the one-step reduction function is partial and also because of possible divergence.

4.6 Summary

In this chapter we introduced programming languages and we explained the need for *high-level* programming languages. Programming languages are formally defined by a grammar of abstract syntax and a semantic specification over the syntax. Three different kinds of semantics were introduced and were used to give the semantics for the sample programming language **TLLF**: A denotational description defining the meaning of programs *extensionally*, and two small-step operational descriptions defining the meaning of programs *intensionally*. In the following chapter we show that indeed the three semantics are equivalent.

We did not make a distinction between *static semantics* and *dynamic semantics*. Via the introduction of a type system the set of well-formed programs can be defined to be the set of closed expressions that can be type-checked. The introduction of such a static semantics would rule out expressions that by repeated application of the one-step reduction function would eventually reach a stuck term. Furthermore, we silently ignored axiomatic frameworks, because they will not be used in the following chapters.

Chapter 5

λ-calculi, programming languages, and semantic artifacts

In the previous chapter we loosely argued that the three different semantics for the tiny programming language **TLLF** define the *same* semantics. No proposition like

$$\mathsf{E}[\![\mathsf{e}]\!]_{\mathbf{TLLF}}\ \mathit{mt\_env} = \mathit{inVal}\ (\mathit{inBasic}\ \mathsf{b}) \iff \mathtt{run}_{\mathbf{TLLF}}\ \mathsf{e} = \mathsf{b} \iff \mathtt{eval}_{\mathbf{TLLF}}\ \mathsf{e} = \mathsf{b}[\mathsf{s}]$$

was included. In this chapter we will see how such propositions are proved by applying well-known meaning-preserving transformations to the semantic artifacts.

Roadmap We begin in Section 5.1 with Plotkin's classical work on relations between λ-calculi and programming languages [59]. We then introduce the 'syntactic' correspondence in Section 5.2 and the 'functional' correspondence in Section 5.3 investigated by Danvy et al. [2–5, 8–10]. We introduce in Section 5.2 the $\lambda\hat{\rho}$-calculus, we illustrate the syntactic correspondence starting from the reduction semantics for **TLLF** (which is a strategy in the $\lambda\hat{\rho}$-calculus), and we mechanically derive the abstract machine for **TLLF**. We illustrate the functional correspondence in Section 5.3 by starting from a definitional interpreter for **TLLF** (which is the denotational semantics for **TLLF** with a semantics for the metalanguage) and mechanically derive the abstract machine for **TLLF**.

5.1 Call by value, call by name, and the λ-calculus

Plotkin's starting point is Landin's definitional SECD machine for the programming language ISWIM which was introduced on page 49. Plotkin aims for a calculus of terms that corresponds to the core of ISWIM, i.e., applicative expressions (AE). We will be concrete and let the abstract syntax of AE coincide with the abstract syntax of **TLLF** found on page 46, i.e., pure λ-terms with literal integers and `S` as basic constants. We let $\mathit{Eval}_{\mathrm{v}}$ denote the defining function for AE according to the SECD machine. It is a partial function mapping closed AEs to values:[1]

$$\mathit{Value} \qquad v \ ::= \ \lambda\mathsf{e} \mid \mathsf{b}$$

[1] As Plotkin defines the SECD machine it evaluates closed expressions to closures. Unloading the result off the terminal state consists in mapping a closure to the expression it represents via a function *Real*.

5.1.1 Call by value

When the SECD machine encounters an application $e\,e'$, it (1) evaluates the operand expression e', (2) evaluations the operator expression e, and if that last expression yields a closure $\langle \lambda x.e'', \rho \rangle$ it (3) evaluates e'' in the environment ρ extended with a binding of x to the value of e'. Such a calling mechanism is said to be (*right-to-left*) *call-by-value*.

Correspondence with the λ_v-calculus In the following theorem, due to Plotkin, $\mathit{normalize}_v$ denotes the normalization function from Section 3.3.3; e denotes a closed AE which is also a term in the λ_v-calculus:

Theorem 4 (Plotkin I)
$\mathit{normalize}_v\ e = v \iff \mathit{Eval}_v\ e = v$

In words, Plotkin has shown that considering a closed expression either (1) evaluation via the SECD machine and weak head normalization in the λ_v-calculus (following the standard reduction strategy $\mapsto_v$) both do not yield a value or (2) they both yield the same value. From Proposition 4 on page 43 it especially follows that $e =_{\beta_v\delta} b \iff \mathit{Eval}_v\ e = b$. The SECD machine evaluates a program to a basic constant if and only if the program can be proved equal to that basic constant in the λ_v-calculus. Abstractions will be equal in $=_{\beta_v\delta}$. In general the abstractions are different because in the λ_v-calculus, reductions can be performed in the body of abstractions.

Soundness and Incompleteness of the λ_v-calculus Plotkin also introduces the notion of *operational equivalence* between expressions, which gives a more general notion of correctness of the λ_v-calculus with respect to the programming language represented by Eval_v (i.e., the SECD machine). The above property only consider closed expressions, i.e., whole programs. Operational equivalence ($\approx$) is defined via the notion of contexts for λ-expressions as defined on page 9:

$$e \approx e' \quad \text{iff} \quad \begin{cases} \mathit{Eval}_v\ C[e], \mathit{Eval}_v\ C[e'] \text{ are both undefined, or} \\ \mathit{Eval}_v\ C[e] = b \iff \mathit{Eval}_v\ C[e'] = b \\ \text{for all closed } C[e], C[e'] \end{cases}$$

In the definition e and e' are not restricted to closed λ-expressions. If one of the evaluations yields a basic constant b the other evaluation also yields b. If one of the evaluation yields an abstraction the other evaluation also yields an abstraction. Operational equivalence does not demand a relation between these two abstractions.[2]

The correctness of the λ_v-calculus wrt. evaluation via the SECD machine (i.e., Eval_v) is summarized in the following theorem.

Theorem 5 (Plotkin II)
$e =_{\beta_v\delta} e' \Longrightarrow e \approx e'$

In other words, it is only possible to prove two terms equal in the λ_v-calculus if they are operationally equivalent in the SECD machine. The λ_v-calculus is said to be *sound* wrt. the programming language defined by the SECD machine.

[2]If evaluation was defined to give, e.g., a symbol '`abstraction`' when evaluation yields an abstraction the definition of operational equivalence simplifies to $\mathit{Eval}_v\ C[e] = \mathit{Eval}_v\ C[e']$ for all closed $C[e], C[e']$. Felleisen and Flatt [31] follow that approach. They do not consider evaluation via the SECD machine.

The right-to-left direction does not hold: A counter example is two closed expressions that are not equal in the λ_v-calculus and that both make the SECD machine diverge. In any context with no free variables by construction evaluation yields the same value on the two resulting programs or evaluation is undefined for both programs. The two terms are hence operationally equivalent but by assumption they are not equal in the λ_v-calculus. The λ_v-calculus is said to be *incomplete* wrt. the programming language.

5.1.2 Call by name

Analogously with $Eval_v$ in the call-by-value case Plotkin defines $Eval_n$ to represent evaluation of expressions following *call-by-name* evaluation order. $Eval_n$ then represents a call-by-name version of the SECD machine. Plotkin shows similar theorems that directly connect the call-by-name evaluation mechanism with weak head normalization in the $\lambda\beta\delta$-calculus — the λ-calculus extended with the δ-rule to cope with the introduction of basic constants introduced in Section 2.8. Plotkin likewise proves that the $\lambda\beta\delta$-calculus is sound and incomplete wrt. the programming language defined by the call-by-name version of the SECD machine.

5.2 A syntactic correspondence

In the previous section we stated Plotkin's results from 1975, which once and for all settled the relationship between the evaluation mechanisms in functional languages and reductions in calculi: Call by name corresponds to weak head normalization in the $\lambda\beta\delta$-calculus and call by value corresponds to weak head normalization in the λ_v-calculus.[3]

Reduction-based evaluation In a reduction semantics, as defined in Section 4.5.3, evaluation is defined as iteratively applying the one-step reduction function until a value is obtained. That is, such an evaluation is *reduction-based*, which means a series of intermediate expressions are constructed, because the one-step reduction function is the composition of decomposition, contraction and plugging as described in that section. A reduction semantics is hence the definition of a calculus and a strategy to perform the reductions. In that respect, Plotkin shows that a reduction semantics for normalization to weak head normal form in the λ_v-calculus directly corresponds to evaluation via the SECD machine — the very first abstract machine. Analogously, a reduction semantics for normalization to weak head normal form in the $\lambda\beta\delta$-calculus corresponds to call-by-name evaluation via the (now modified) SECD machine.

A derivational approach In this section we illustrate a *derivational* approach to find correspondences between reduction semantics and abstract machines. Danvy and Nielsen paved the way in their work on removing the overhead in the naive implementation of a reduction semantics as iteratively applying the one-step reduction function defined for the semantics [28]. They introduce a *refocus* function. When a reduction semantics is deterministic, a

[3]The correspondences does not hold for weak head normal forms which are not values (with our definition of weak head normal forms). The SECD machine and Plotkin's left reduction evaluates right-hand sub-terms of applications even when the left-hand sub-term evaluates to a non-functional basic constant.

given non-value expression can be *uniquely* decomposed into a reduction context and a redex. That is, decomposition can be defined as a function on expressions. Because plugging is also a function it is well-defined when refocus is defined as the composition of plugging and decomposing. Evaluation can then be defined via the refocus function because each iteration ends with a plugging and the next iteration starts with a decomposition. The central observation is that as long as the refocus function is extensionally equal to the composition of plugging and decomposing, the meaning of evaluation is preserved regardless of the internal structure of the refocus function. Danvy and Nielsen introduce a mechanical way to create an efficient representation of *refocus*.

Biernacka and Danvy exploit the above result and mechanically derive abstract machines from various reduction semantics [8, 9]. In the following we illustrate this derivational approach.

5.2.1 The $\lambda\hat{\rho}$-calculus

The $\lambda\hat{\rho}$-calculus defined by Biernacka and Danvy [9] is a minimal extension of Curien's calculus of closures $\lambda\rho$ briefly introduced in Section 3.2 as a calculus with explicit substitutions. The extension is made such that it is possible to define one-step reduction in the calculus. In the following we define the $\lambda\hat{\rho}$-calculus extended to cope with basic constants. The calculus is extended the same way the λ-calculus was extended in Section 2.8.

Following Chapter 1, we specify a grammar of terms, the notion of reduction, and compatibility. One-step reduction and equality in the calculus are defined like in the λ-calculus. The difference merely consists in the $\lambda\hat{\rho}$-calculus being a calculus of *closures* instead of a calculus of *terms*.

The language The language of the $\lambda\hat{\rho}$-calculus defines a closure as either a λ-term with an associated list of closures or two closures in juxtaposition:

$$\begin{array}{lrcl} \textit{Substitution}_{\hat{\rho}} & s & ::= & \bullet \mid c \cdot s \\ \textit{Closure}_{\hat{\rho}} & c & ::= & t[s] \mid c\,c \end{array}$$

Here t ranges over de Bruijn-indexed λ-terms extended with basic constants like in Section 2.8. Closures are the syntactic units.

Notion of reduction Four contraction rules constitute the notion of reduction $\hat{\rho}$ which denote a function from closures to closures:[4]

$$\begin{array}{rrcl} \beta: & (\lambda t)[s]\,c & \to & t[c \cdot s] \\ \iota: & i[c_1 \cdots c_j] & \to & c_i\text{ , if } i \leq j \\ \pi: & (t\,t')[s] & \to & t[s]\,t'[s] \\ \delta: & q[s]\,l[s'] & \to & (\mathtt{capp}\,(q,\ l))[s'] \\ & & & \\ & \hat{\rho} & := & \beta \cup \iota \cup \pi \cup \delta \end{array}$$

The notion of reduction is context-insensitive in the sense that the contractum of a redex only depends on the subparts of the redex, and does not alter the reduction context.[5]

[4] Biernacka and Danvy do not include basic constants in the term-part of closures $t[s]$ and hence define only the first three contraction rules.

[5] Biernacka and Danvy present several context-sensitive calculi and reduction semantics in such calculi [9].

One-step reduction and equality The one-step reduction relation $\to_{\hat\rho}$, called $\hat\rho$-reduction, is defined as the compatible closure of the notion of reduction relative to contexts:

$$\textit{Context}_{\hat\rho} \qquad \mathsf{C} \quad ::= \quad [\,] \mid \mathsf{C}\,\mathsf{c} \mid \mathsf{c}\,\mathsf{C} \mid \mathsf{t}[\mathsf{c}_1 \cdots \mathsf{C} \cdot \mathsf{c}_{i+1} \cdots \mathsf{c}_j],\ 1 \le i \le j$$

Equality in the calculus $=_{\hat\rho}$ is defined as the smallest equivalence relation of the reflexive transitive closure of $\hat\rho$-reduction (denoted by $\to^*_{\hat\rho}$). As can be seen from the grammar of contexts, this calculus is *weak* in the sense that reductions cannot be performed in the body of abstractions.

Normal forms A grammar of closures generating the $\hat\rho$-normal forms is structurally very similar to normal forms in the $\lambda\beta\delta$-calculus from page 35 (with $i > j$):

$$\begin{array}{llcl}
 & s_n & ::= & \bullet \mid n \cdot s_n \\
 & a & ::= & i[n_1 \cdots n_j] \mid a\,n \mid l[s_n]\,n \mid q[s_n]\,d \\
 & d & ::= & a \mid (\lambda t)[s_n] \mid q[s_n] \\
\textit{NForm}_{\hat\rho} & n & ::= & d \mid l[s_n]
\end{array}$$

Because reductions are never performed inside the body of abstractions, these sub-terms can be arbitrary λ-terms.

Church-Rosser of $\hat\rho$ Just like $\to^*_{\beta}$, $\to^*_{\beta\eta}$, and $\to^*_{\beta_v\delta}$ all satisfy the diamond property (the notion of reduction is said to be Church-Rosser), $\to^*_{\hat\rho}$ also satisfies that property. A proof is possible via a correspondence with the λ-calculus, which is the topic of Section 5.2.2.

In the first group of calculi the syntactic units are λ-terms, which means that the contraction rules consume and produce λ-terms. Because one-step reductions in those cases are standard compatible closures of the notion of reduction, redexes can occur in all parts of the term and they can be reduced in any order and a corresponding normal form (if it exists) is unique.

In the $\lambda\hat\rho$-calculus the syntactic units are closures. The contraction rules consume closures and produce closures. That is, no reductions are possible in the term-parts of closures. In that respect the compatibility is restricted, but the uniqueness property of normal forms also holds for the $\lambda\hat\rho$-calculus.

5.2.2 Correspondence with the λ-calculus

From λ-calculus terms to $\lambda\hat\rho$-calculus closures A translation of terms from the extended λ-calculus to closures in the $\lambda\hat\rho$-calculus consists in introducing an empty substitution, and translating variables into de Bruijn indices. Translating to use de Bruijn indices is `toindices` as defined in Section 1.6 (with the trivial extension for allowing primitive constants in the terms).[6] As seen in Section 1.6 the λ-calculus can equivalently be defined with de Bruijn-indexed λ-terms. To simplify presentation in this section, we map from de Bruijn-indexed λ-terms:

$$\begin{array}{rcl}
\mathtt{toclosure}_{\hat\rho} & : & \textit{Term}_{deB} \to \textit{Closure}_{\hat\rho} \\
\mathtt{toclosure}_{\hat\rho}\ \mathsf{t} & = & \mathsf{t}[\bullet]
\end{array}$$

[6] To emphasize the connection between variable names and de Bruijn indices the notation for substitutions was also used for lists of variables in the definition of `toindices`.

From $\lambda\widehat{\rho}$-calculus closures to λ-calculus terms Translating from closures to terms is a bit more involved. All the delayed substitutions contained in the substitution part of a closure have to be forced into actual substitutions in the term:

$$
\begin{array}{rcll}
\sigma_{\widehat{\rho}} & : & \mathit{Closure}_{\widehat{\rho}} \times \mathit{Int} \times \mathit{Int} \to \mathit{Term}_{deB} & \\
\sigma_{\widehat{\rho}}\ (b[s],\ k,\ g) & = & b & \\
\sigma_{\widehat{\rho}}\ (i[s],\ k,\ g) & = & i, & \text{if } i \leq k \\
\sigma_{\widehat{\rho}}\ (i[c_1 \cdots c_j],\ k,\ g) & = & \sigma_{\widehat{\rho}}\ (c_{i-k},\ 0,\ g+k), & \text{if } k < i \leq k+j \\
\sigma_{\widehat{\rho}}\ (i[c_1 \cdots c_j],\ k,\ g) & = & i-j+g, & \text{if } i > k+j \\
\sigma_{\widehat{\rho}}\ ((\lambda t)[s],\ k,\ g) & = & \lambda(\sigma_{\widehat{\rho}}\ (t[s],\ k+1,\ g)) & \\
\sigma_{\widehat{\rho}}\ ((t\,t')[s],\ k,\ g) & = & (\sigma_{\widehat{\rho}}\ (t[s],\ k,\ g))\ (\sigma_{\widehat{\rho}}\ (t'[s],\ k,\ g)) & \\
\sigma_{\widehat{\rho}}\ (c\,c',\ k,\ g) & = & (\sigma_{\widehat{\rho}}\ (c,\ k,\ g))\ (\sigma_{\widehat{\rho}}\ (c',\ k,\ g)) &
\end{array}
$$

Without the trivial extension to cope with basic constants, the definition of $\sigma_{\widehat{\rho}}$ is still an extended version of σ presented by Biernacka and Danvy [8]: $\sigma_{\widehat{\rho}}$ also correctly translates *open* closures. $\sigma_{\widehat{\rho}}$ uses a local depth k and global depth g: k is the current depth local to the current substitution list and g is the current depth in the overall constructed term. In the translation of a closure with an abstraction as term-part, the local depth k of the closure is incremented. That is, when translating an index i three possibilities show up:

(1) If $i \leq k$, i is bound by an abstraction that is not reduced but translated. That is, i is the result.

(2) If $k < i \leq k+j$ where j is the length of the substitution, the index refers to a closure c that is introduced in an contracted application. Again, relative to the current depth k a translation of the $(i-k)$-th closure in the substitution is the result. This recursive translation is done with the local depth reset and the global depth raised by k.

(3) If $i > k+j$ the index does not refer to an unreduced abstraction or to a reduced abstraction and is therefore the $(i-j-k)$-th free index relative to the overall closure. The global depth of the occurrence is $k+g$, and the resulting index is therefore $i-j-k+k+g$, i.e., $i-j+g$.

The explicit substitutions are converted to actual substitutions by use of $\sigma_{\widehat{\rho}}$ with 0 as local and global depth:

$$
\begin{array}{rcl}
\texttt{fromclosure}_{\widehat{\rho}} & : & \mathit{Closure}_{\widehat{\rho}} \to \mathit{Term}_{deB} \\
\texttt{fromclosure}_{\widehat{\rho}}\ c & = & \sigma_{\widehat{\rho}}\ (c,\ 0,\ 0)
\end{array}
$$

An example with an open term A term is mapped to the $\widehat{\rho}$-calculus, two reductions performed, and the resulting closure is mapped back again to a λ-term. Alternatively one corresponding reduction is performed in the λ-calculus. The terms involved are for readability presented with named variables:

$$
\begin{array}{rcl}
\texttt{toindices}\,((\lambda x.\lambda y.x\ (\lambda y.x\ z))\ ((\lambda x.w\ x)\ v)) & = & ((\lambda\lambda 2\ (\lambda 3\ 4))\ ((\lambda 3\ 1)\ 3)) \\
 & & \\
\texttt{toclosure}_{\widehat{\rho}}\ ((\lambda\lambda 2\ (\lambda 3\ 4))\ ((\lambda 3\ 1)\ 3)) & = & ((\lambda\lambda 2\ (\lambda 3\ 4))\ ((\lambda 3\ 1)\ 3))[\bullet] \\
 & \to_{\widehat{\rho}} & ((\lambda\lambda 2\ (\lambda 3\ 4))[\bullet])\ (((\lambda 3\ 1)\ 3)[\bullet]) \\
 & \to_{\widehat{\rho}} & (\lambda 2\ (\lambda 3\ 4))[(((\lambda 3\ 1)\ 3)[\bullet]) \cdot \bullet] \\
\texttt{fromclosure}_{\widehat{\rho}}\ ((\lambda 2\ (\lambda 3\ 4))[(((\lambda 3\ 1)\ 3)[\bullet]) \cdot \bullet]) & = & \lambda(\lambda 4\ 1)\ 4\ (\lambda(\lambda 5\ 1)\ 5\ 3) \\
 & & \\
((\lambda\lambda 2\ (\lambda 3\ 4))\ ((\lambda 3\ 1)\ 3)) & \to_{\beta} & \lambda(\lambda 4\ 1)\ 4\ (\lambda(\lambda 5\ 1)\ 5\ 3) \\
 & & \\
\texttt{fromindices}\,(\lambda(\lambda 4\ 1)\ 4\ (\lambda(\lambda 5\ 1)\ 5\ 3)) & = & \lambda y_1.(\lambda x_2.f_2\ x_2)\ f_3\ (\lambda y_3.(\lambda x_4.f_2\ x_4)\ f_3\ f_1)
\end{array}
$$

(1) Mapping the term to the $\lambda\hat{\rho}$-calculus, (2) performing two reductions in the calculus, and (3) mapping the resulting closure to a term with de Bruijn indices and no explicit substitutions yields a term where indices corresponding to free variables are consistent (witness the first occurrence of 4 and the first occurrence of 5 correspond to the same free variable in the second to last equation).

It is clear from the contraction rules that if the sample term above was the body of an abstraction no reductions would be possible after mapping to a closure.

Translations interaction The above example illustrates the achievement: (1) Mapping term t to closure c' in the closure calculus, (2) performing a series of reductions starting with this closure, and (3) mapping the obtained closure c back to a term in the λ-calculus yields a term t' that can be obtained in a series of $\beta\delta$-reductions in the λ-calculus from the original term t:

$$(\texttt{toclosure}_{\hat{\rho}}\ \mathrm{t}) \rightarrow^*_{\hat{\rho}} \mathrm{c} \quad \Longrightarrow \quad \mathrm{t} \rightarrow^*_{\beta\delta} (\texttt{fromclosure}_{\hat{\rho}}\ \mathrm{c})$$

The achievement (proved by Biernacka and Danvy for pure λ-terms [8]) is expressible as a figure:

$$\begin{tikzcd}
\mathrm{t} \arrow[r, "\rightarrow^*_{\beta\delta}"] \arrow[d, "\texttt{toclosure}_{\hat{\rho}}"'] & \mathrm{t}' \\
\mathrm{c}' \arrow[r, "\rightarrow^*_{\hat{\rho}}"'] & \mathrm{c} \arrow[u, "\texttt{fromclosure}_{\hat{\rho}}"']
\end{tikzcd}$$

As mentioned above the $\lambda\hat{\rho}$-calculus is weak. This property becomes evident here: It does not hold that for all series of $\beta\delta$-reductions starting with t the corresponding reductions can be performed on the closure $\texttt{toclosure}_{\hat{\rho}}\ \mathrm{t}$ in the $\lambda\hat{\rho}$-calculus.

5.2.3 A normal-order reduction semantics for the $\lambda\hat{\rho}$-calculus

A normal-order reduction semantics for the syntax of the $\lambda\hat{\rho}$-calculus as defined in Section 5.2.1 has actually already been defined. The reduction semantics for the programming language **TLLF** is one such. The one-step reduction function $\mapsto_{\mathbf{TLLF}}$ (defined on page 55) is the compatible closure of the notion of reduction $\hat{\rho}$ wrt. the specified grammar of reduction contexts.

5.2.4 The reduction semantics with explicit decomposition

The definition of the function $\mapsto_{\mathbf{TLLF}}$ as defined on page 55 relies as mentioned implicitly on the unique decomposition. In this section this decomposition is made explicit.

Reduction contexts represented inside-out and plugging As reduction contexts are defined in the reduction semantics they are represented 'outside-in'. We equivalently represent the reduction contexts 'inside-out':

$$\mathit{EContext} \qquad \mathrm{E}[\,] \ ::= \ [\,] \mid \mathrm{E}[[\,]\ \mathrm{c}] \mid \mathrm{E}[\mathrm{q}[\mathrm{s}]\ [\,]]$$

A context represented 'inside-out' is no longer just a closure with a hole and we define plugging explicitly:

$$
\begin{array}{rcl}
\texttt{plug} & : & \mathit{EContext} \times \mathit{Closure} \rightarrow \mathit{Closure} \\
\texttt{plug}\,([\,],\ c) & = & c \\
\texttt{plug}\,(E[[\,]\,c'],\ c) & = & \texttt{plug}\,(E[\,],\ c\,c') \\
\texttt{plug}\,(E[q[s]\,[\,]],\ c) & = & \texttt{plug}\,(E[\,],\ q[s]\,c)
\end{array}
$$

`plug` is total.

Potential redexes and a stronger decomposition property The unique decomposition property only talks about closures where the one-step reduction function is defined. A stronger property relies on potential redexes. For the reduction semantics the set of closures that constitute the potential redexes is generated by a grammar:

$$
\mathit{PRedex}_{\hat{\rho}} \qquad p \ ::= \ i[s] \mid (\lambda t)[s]\,c \mid (t\,t)[s] \mid l[s]\,c \mid q[s]\,v
$$

The closures generated by this grammar are potential in the sense that some of them are not *actual redexes*: they are closures that cannot be decomposed but are still not redexes. For example, the closures $\mathtt{S}[\bullet]\ \mathtt{S}[\bullet]$ and $3[c \cdot c \cdot \bullet]$ are potential redexes but not actual redexes.

The stronger decomposition property says that *any* non-value closure c can be uniquely decomposed into a reduction context $E[\,]$ and a potential redex p such that $c = E[p]$. Even when $\mapsto_{\mathbf{TLLF}}$ is not defined on c.

Decomposition The decomposition of a closure into a reduction contexts and a potential redex is well-defined because of the stronger unique-decomposition property:

$$
\begin{array}{rcl}
\mathit{Decomp} & = & \mathit{PRedex} \times \mathit{EContext} \\
\texttt{decompose} & : & \mathit{Closure} \rightarrow \mathit{Value} + \mathit{Decomp} \\
\texttt{decompose}\,c & = & \texttt{decompose}'\,(c,\ [\,]) \\
\\
\texttt{decompose}' & : & \mathit{Closure} \times \mathit{EContext} \rightarrow \mathit{Value} + \mathit{Decomp} \\
\texttt{decompose}'\,(v,\ E[\,]) & = & \texttt{decompose}'_{\mathrm{aux}}\,(E[\,],\ v) \\
\texttt{decompose}'\,(i[s],\ E[\,]) & = & (i[s],\ E[\,]) \\
\texttt{decompose}'\,((t\,t')[s],\ E[\,]) & = & ((t\,t')[s],\ E[\,]) \\
\texttt{decompose}'\,(c\,c',\ E[\,]) & = & \texttt{decompose}'\,(c,\ E[[\,]\,c']) \\
\\
\texttt{decompose}'_{\mathrm{aux}} & : & \mathit{EContext} \times \mathit{Value} \rightarrow \mathit{Value} + \mathit{Decomp} \\
\texttt{decompose}'_{\mathrm{aux}}\,([\,],\ v) & = & v \\
\texttt{decompose}'_{\mathrm{aux}}\,(E[[\,]\,c],\ q[s]) & = & \texttt{decompose}'\,(c,\ E[q[s]\,[\,]]) \\
\texttt{decompose}'_{\mathrm{aux}}\,(E[[\,]\,c],\ v) & = & (v\,c,\ E[\,]), v \neq q[s] \\
\texttt{decompose}'_{\mathrm{aux}}\,(E[q[s]\,[\,]],\ v) & = & (q[s]\,v,\ E[\,])
\end{array}
$$

`decompose` is total (it is the identity function on values) and has been proved correct by Danvy and Nielsen [28].[7] Because reduction contexts are represented inside-out and are isomorphic with lists, `decompose` builds up a stack of elementary contexts.[8]

[7]Danvy and Nielsen do not have the support for basic constants but their proof directly extends to cope with such constants

[8]Such a stack of elementary contexts is also known as a *data-structure continuation* [68].

The evaluation function $\mathtt{eval}_{\mathbf{TLLF}}$ from page 55 is reformulated with an explicit use of `decompose` and `plug`:

$$
\begin{array}{rcl}
\mathtt{eval}_{\mathbf{TLLF}} & : & \mathit{Exp} \rightarrow \mathit{Value} \\
\mathtt{eval}_{\mathbf{TLLF}}\, e & = & \mathtt{iterate}\,(\mathtt{decompose}\,(e[\bullet])) \\
\\
\mathtt{iterate} & : & \mathit{Value} + \mathit{Decomp} \rightarrow \mathit{Value} \\
\mathtt{iterate}\, v & = & v \\
\mathtt{iterate}\,(i[c_1 \cdots c_i \cdots c_j],\ E[\,]) & = & \mathtt{iterate}\,(\mathtt{decompose}\,(\mathtt{plug}\,(E[\,],\ c_i))) \\
\mathtt{iterate}\,((\lambda e)[s]\ c,\ E[\,]) & = & \mathtt{iterate}\,(\mathtt{decompose}\,(\mathtt{plug}\,(E[\,],\ e[c \cdot s]))) \\
\mathtt{iterate}\,((e\, e')[s],\ E[\,]) & = & \mathtt{iterate}\,(\mathtt{decompose}\,(\mathtt{plug}\,(E[\,],\ e[s]\ e'[s]))) \\
\mathtt{iterate}\,(q[s]\ l[s'],\ E[\,]) & = & \mathtt{iterate}\,(\mathtt{decompose}\,(\mathtt{plug}\,(E[\,],\ (\mathtt{capp}\,(q,\ l))[s'])))
\end{array}
$$

5.2.5 Refocusing

If the initial application of `decompose` in the evaluation function above is preceded by plugging the closure $e[\bullet]$ into the empty context $[\,]$, every application of `decompose` is preceded by an application of `plug`. Evaluation consists of a series of reductions. In this series, every plugging is now immediately followed by a decomposition except for the last plugging where the resulting value is obtained. In other words, a refocusing is performed between each actual reduction.

Danvy and Nielsen have shown that an efficient representation of refocusing is already around [21, 28], where the intermediate closure, which is present in a direct composition of decomposing and plugging, can be deforested. This deforestation eliminates two operations with complexity proportional to the height of the closure:

$$
\begin{array}{rcl}
\mathtt{refocus} & := & \mathtt{decompose} \circ \mathtt{plug} \\
 & = & \mathtt{decompose}'
\end{array}
$$

5.2.6 Obtaining an abstract machine

Reduction-free evaluation Evaluation can be reformulated with `decompose′` as the refocusing function as presented in Section 5.2.5. Thereby evaluation is no longer reduction based: The decomposition is building a potential redex and a stack of elementary reduction contexts. Refocusing via `decompose′` just resumes decomposing with the contractum of the redex (if it is also an actual redex) and with the current stack of elementary reduction contexts. That is, intermediate closures are no longer built:

$$
\begin{array}{rcl}
\mathtt{eval}_{\mathbf{TLLF}} & : & \mathit{Exp} \rightarrow \mathit{Value} \\
\mathtt{eval}_{\mathbf{TLLF}}\, e & = & \mathtt{iterate}\,(\mathtt{refocus}\,(e[\bullet],\ [\,])) \\
\\
\mathtt{iterate} & : & \mathit{Value} + \mathit{Decomp} \rightarrow \mathit{Value} \\
\mathtt{iterate}\, v & = & v \\
\mathtt{iterate}\,(i[c_1 \cdots c_i \cdots c_j],\ E[\,]) & = & \mathtt{iterate}\,(\mathtt{refocus}\,(c_i,\ E[\,])) \\
\mathtt{iterate}\,((\lambda e)[s]\ c,\ E[\,]) & = & \mathtt{iterate}\,(\mathtt{refocus}\,(e[c \cdot s],\ E[\,])) \\
\mathtt{iterate}\,((e\, e')[s],\ E[\,]) & = & \mathtt{iterate}\,(\mathtt{refocus}\,(e[s]\ e'[s],\ E[\,]|)) \\
\mathtt{iterate}\,(q[s]\ l[s'],\ E[\,]) & = & \mathtt{iterate}\,(\mathtt{refocus}\,((\mathtt{capp}\,(q,\ l))[s'],\ E[\,]))
\end{array}
$$

Fusing iterate and refocus The auxiliary function `iterate` is a trampoline function [35] for `refocus` together with the contraction. Fusing `iterate` and `refocus` makes `refocus` call itself and therefore `iterate` superfluous:

$$
\begin{array}{rcl}
\texttt{eval}_{\mathbf{TLLF}} & : & \mathit{Exp} \to \mathit{Value} \\
\texttt{eval}_{\mathbf{TLLF}}\, e & = & \texttt{refocus}\,(e[\bullet],\ [\,])
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{refocus} & : & \mathit{Closure} \times \mathit{EContext} \to \mathit{Value} \\
\texttt{refocus}\,(v,\ E[\,]) & = & \texttt{refocus}_{\texttt{aux}}\,(E[\,],\ v) \\
\texttt{refocus}\,(i[c_1 \cdots c_i \cdots c_j],\ E[\,]) & = & \texttt{refocus}\,(c_i,\ E[\,]) \\
\texttt{refocus}\,((e\, e')[s],\ E[\,]) & = & \texttt{refocus}\,(e[s]\, e'[s],\ E[\,]) \\
\texttt{refocus}\,(c\, c',\ E[\,]) & = & \texttt{refocus}\,(c,\ E[[\,]\, c'])
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{refocus}_{\texttt{aux}} & : & \mathit{EContext} \times \mathit{Value} \to \mathit{Value} \\
\texttt{refocus}_{\texttt{aux}}\,([\,],\ v) & = & v \\
\texttt{refocus}_{\texttt{aux}}\,(E[[\,]\, c],\ (\lambda e)[s]) & = & \texttt{refocus}\,(e[c \cdot s],\ E[\,]) \\
\texttt{refocus}_{\texttt{aux}}\,(E[[\,]\, c],\ q[s]) & = & \texttt{refocus}\,(c,\ E[q[s]\, [\,]]) \\
\texttt{refocus}_{\texttt{aux}}\,(E[q[s]\, [\,]],\ l[s']) & = & \texttt{refocus}\,((\mathsf{capp}\,(q,\ l))[s'],\ E[\,])
\end{array}
$$

The above specification is an abstract machine: The generation of the initial configuration $(e[\bullet],\ [\,])$ constitute the loading.

Short-circuiting transitions In the third clause of `refocus`, where the closure takes the form $(e\, e')[s]$, it is observed that `refocus` is called with a configuration that in every case is matched by the fourth clause. Also the closure construction $c\, c$ is not used elsewhere. In other words, by merging the two transitions only one step is needed. The fourth clause is the only place where that contexts construction is used. Therefore the closure in the context always takes the form of a term paired with a substitution, and it is hence safe to remove the fourth clause of `refocus`:

$$
\begin{array}{rcl}
\texttt{eval}_{\mathbf{TLLF}} & : & \mathit{Exp} \to \mathit{Value} \\
\texttt{eval}_{\mathbf{TLLF}}\, e & = & \texttt{refocus}\,(e[\bullet],\ [\,])
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{refocus} & : & \mathit{Closure} \times \mathit{EContext} \to \mathit{Value} \\
\texttt{refocus}\,(v,\ E[\,]) & = & \texttt{refocus}_{\texttt{aux}}\,(E[\,],\ v) \\
\texttt{refocus}\,(i[c_1 \cdots c_i \cdots c_j],\ E[\,]) & = & \texttt{refocus}\,(c_i,\ E[\,]) \\
\texttt{refocus}\,((e\, e')[s],\ E[\,]) & = & \texttt{refocus}\,(e[s],\ E[[\,]\, e'[s]])
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{refocus}_{\texttt{aux}} & : & \mathit{EContext} \times \mathit{Value} \to \mathit{Value} \\
\texttt{refocus}_{\texttt{aux}}\,([\,],\ v) & = & v \\
\texttt{refocus}_{\texttt{aux}}\,(E[[\,]\, e'[s']],\ (\lambda e)[s]) & = & \texttt{refocus}\,(e[e'[s'] \cdot s],\ E[\,]) \\
\texttt{refocus}_{\texttt{aux}}\,(E[[\,]\, e'[s']],\ q[s]) & = & \texttt{refocus}\,(e'[s'],\ E[q[s]\, [\,]]) \\
\texttt{refocus}_{\texttt{aux}}\,(E[q[s]\, [\,]],\ l[s']) & = & \texttt{refocus}\,((\mathsf{capp}\,(q,\ l))[s'],\ E[\,])
\end{array}
$$

The Krivine machine All uses of closures are now as terms with an associated substitution. We hence flatten the configurations such that the machine operates directly on λ-terms. With an eye on the resulting abstract machine we remove the substitution-part of basic constants and inline $\texttt{refocus}_{\texttt{aux}}$:

$$
\begin{array}{rrcl}
\textit{EContext}' : & E[\,] & ::= & [\,] \mid E[[\,]\,(e, s)] \mid E[q\,[\,]] \\
\textit{Value}' : & \nu & ::= & b \mid (\lambda e, \rho)
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{eval}_{\textbf{TLLF}} & : & \textit{Exp} \to \textit{Value}' \\
\texttt{eval}_{\textbf{TLLF}} & = & \texttt{refocus}\,(t,\ \bullet,\ [\,])
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{refocus}\,(b,\ s,\ [\,]) & = & b \\
\texttt{refocus}\,(\lambda e,\ s,\ [\,]) & = & (\lambda e,\ s) \\
\texttt{refocus}\,(i,\ (e_1, s_1)\cdots(e_i, s_i)\cdots(e_j, s_j),\ E[\,]) & = & \texttt{refocus}\,(e_i,\ s_i,\ E[\,]) \\
\texttt{refocus}\,(\lambda e,\ s,\ E[[\,]\,(e', s')]) & = & \texttt{refocus}\,(e,\ (e', s')\cdot s,\ E[\,]) \\
\texttt{refocus}\,(e\,e',\ s,\ E[\,]) & = & \texttt{refocus}\,(e,\ s,\ E[[\,]\,(e', s)]) \\
\texttt{refocus}\,(q,\ s,\ E[[\,]\,(e, s')]) & = & \texttt{refocus}\,(e,\ s',\ E[q\,[\,]]) \\
\texttt{refocus}\,(l,\ s,\ E[q\,[\,]]) & = & \texttt{refocus}\,(\texttt{capp}\,(q,\ l),\ s,\ E[\,])
\end{array}
$$

Removing the first and the last two clauses of `refocus`, the abstract machine is the *Krivine machine* [13], which is the standard abstract machine that performs call-by-name evaluation of pure λ-terms. The extra three rules exclusively cope with basic constants.

The above derivation illustrates the syntactic correspondence but also demonstrates that the correspondence directly applies to applied λ-calculi. In later chapters we give more examples.

Relating the abstract machine and the reduction semantics for TLLF Starting with the reduction semantics for the programming language **TLLF** we via the syntactic correspondence derived an abstract machine. A comparison with the abstract machine specified on page 53 defining the semantics for **TLLF** shows that inlining $\texttt{refocus}_{\texttt{aux}}$ yields that machine: The only difference is the stack of the abstract machine for **TLLF**. But the stacks are isomorphic with the contexts used in the derived abstract machine and it is hence the same machine. That is, the abstract machine for **TLLF** and the reduction semantics for **TLLF** are equivalent specifications:

$$\texttt{run}_{\textbf{TLLF}}\ e = b \iff \texttt{eval}_{\textbf{TLLF}}\ e = b[s]$$

Considering for simplicity only basic constants as results: starting from a closed expression e the abstract machine and the reduction semantics for **TLLF** either both do not yield a basic constant or they both yield the same basic constant. This result is a corollary of the correctness of the applied transformations in the derivation of the abstract machine starting from the reduction semantics.

5.3 A functional correspondence

In Section 4.5.1 we introduced denotational semantics and briefly presented the underlying domain theory. We pointed out the descriptive nature of a denotational semantics: Two programs considered equal in the defined language are mapped to the same semantic object. In that respect, the metalanguage used in defining the valuation function is only used to identify the denotation (a point in a semantic domain) — it does not have a notion of 'operations' to 'perform' and hence also no 'ordering' of such operations.

From a computational perspective the constructs of the metalanguage are viewed as denoting operations. That perspective hence immediately introduces a notion of ordering of operations in the metalanguage, and the valuation function becomes an *interpreter* for the defined language depending on the semantics of the constructs of the metalanguage. The denotational specification become a *definitional interpreter*.[9] Reynolds investigated the relationship between the defined language and the metalanguage in such definitional interpreters in his seminal paper *Definitional Interpreters for Higher-Order Programming Languages* [61] and introduced relevant transformations. It is now standard to introduce various language constructs via definitional interpreters. For example, Friedman, Wand, and Haynes use Scheme [43] as metalanguage in *Essentials of Programming Languages* [34].

Danvy et al. have systematically investigated the transformations Reynolds employed. The research has among other things given rise to several papers on a derivational approach to the relationship between semantic artifacts [2–5] pioneered by Reynolds. The approach has been dubbed a *functional correspondence*.

5.3.1 A definitional interpreter

The denotational semantics for **TLLF** on page 49 becomes a definitional interpreter when deciding on a semantics for the metalanguage used. We assume a call-by-name metalanguage. In the extension of the environment we inline the construction of the pair and switch to the list notation used in this text. For closed expressions, looking up a variable $\mathtt{i}$ is identified to taking the $\mathtt{i}$-th element of a list. Evaluation starts with the empty environment $\bullet$:

$$
\begin{array}{rcl}
\mathit{ExpVal} & = & \mathit{Int} + \mathit{Prim} + (\mathit{DenVal} \to \mathit{ExpVal}) \\
\mathit{DenVal} & = & \mathit{ExpVal} \\
\\
\mathtt{eval} & : & \mathit{Exp} \times \mathit{DenVal}\ \mathit{list} \to \mathit{ExpVal} \\
\mathtt{eval}\,(\mathtt{i},\ \mathtt{d}_1 \cdots \mathtt{d}_\mathtt{i} \cdots \mathtt{d}_\mathtt{j}) & = & \mathtt{d}_\mathtt{i} \\
\mathtt{eval}\,(\lambda\mathtt{e},\ \rho) & = & \lambda\mathtt{d}.\mathtt{eval}\,(\mathtt{e},\ \mathtt{d} \cdot \rho) \\
\mathtt{eval}\,(\mathtt{e}\,\mathtt{e}',\ \rho) & = & \mathit{case}\ \mathtt{eval}\,(\mathtt{e},\ \rho)\ \mathit{of} \\
 & & \quad \mathtt{f} \to \mathtt{f}\,(\mathtt{eval}\,(\mathtt{e}',\ \rho)) \\
 & & \quad \mathtt{q} \to \mathit{case}\ \mathtt{eval}\,(\mathtt{e}',\ \rho)\ \mathit{of} \\
 & & \qquad\qquad \mathtt{l} \to \mathtt{capp}\,(\mathtt{q},\ \mathtt{l}) \\
\mathtt{eval}\,(\mathtt{b},\ \rho) & = & \mathtt{b} \\
\\
\mathtt{evaluate} & : & \mathit{Exp} \to \mathit{ExpVal} \\
\mathtt{evaluate}\,\mathtt{e} & = & \mathtt{eval}\,(\mathtt{e},\ \bullet)
\end{array}
$$

Instead of introducing some kind of error mechanism, the interpreter is left partial. Several of the constructs in the defined language are interpreted via a similar construct in the metalanguage. In such a setting the reader must be familiar with higher-order languages and have in-depth understanding of the metalanguage to understand the defined language. In the following we briefly introduce Reynolds's ideas to separate the defined language from the metalanguage.

5.3.2 Closure conversion

Closure conversion eliminates the use of the function space in the values by introducing *closures*: Evaluating an abstraction then results in a pair composed of the body of the abstrac-

[9] A definitional interpreter is an implementation of a big-step operational semantics.

tion and the current environment. This pair represents the evaluation of the body with the environment extended.[10]

5.3.3 Continuation-passing-style/direct-style transformations

Reynolds points out how the evaluation order of the defined language depends on the evaluation order of the metalanguage in a direct style interpreter.[11] To eliminate this evaluation-order dependency Reynolds applies a *continuation-passing-style* (CPS) transformation to the interpreter. In a (higher-order) program in CPS all functions take an extra argument, the *continuation*, a functional representation of the rest of the computation. Instead of returning a value directly, all functions apply the current continuation to the value. All intermediate results are named, and all calls are tail-calls. In other words, the computation has been sequentialized: in the interpreter no implicit control-flow remains, which otherwise depend on the evaluation order of the metalanguage. Plotkin has formally proved the evaluation-order independency of programs in CPS by simulations of call-by-value in a call-by-name evaluation order and visa versa [59].[12]

The interpreter for **TLLF** given above relies on a call-by-name evaluation order of the metalanguage. That is, to define **TLLF** evaluation-order independent we would apply a *call-by-name*-CPS transformation to the interpreter.

The left-inverses of CPS transformations maps programs in CPS into corresponding programs in direct style. Danvy has formulated these *direct-style* transformations [19].

5.3.4 Defunctionalization/refunctionalization

After the CPS transformation the interpreter is higher-order: continuations are functions from intermediate results to final answers. Reynolds introduced *Defunctionalization* as the mechanism to translate higher-order programs into first-order: For the function space of continuations we identify the (finite number of) inhabitants occurring in the program and create a sum space with one injection for each of these inhabitants. Each member holds the free variables of the function it represents. Continuation constructions become injections. Eliminations (applications of continuations) become calls to an *apply function* which (1) dispatches on the first-order continuation representation, and (2) performs the corresponding evaluation.

The left-inverse of defunctionalization transforms programs in defunctionalized form, i.e., in the image of defunctionalization, into corresponding higher-order programs. Danvy and Millikin have investigated this *refunctionalization* [26].

5.3.5 Relating interpreters and abstract machines

Reynolds used the transformations on a definitional interpreter to disconnect the defined language from the metalanguage. The result is a sequentialized, first-order specification.

[10]The term 'closure' in this context is due to Landin [50].

[11]Because **TLLF** is a pure functional language referential transparency holds. That is, if the metalanguage used call-by-value evaluation order the semantics of the language would only change in case of non-terminating evaluation of argument expressions in applications.

[12]It is noted that the notion of CPS and CPS-transformations does not originate in Reynolds work on definitional interpreters. Reynolds has (like Landin [52]) later published a paper on the discoveries of continuations [62].

Danvy et al. have observed that an interpreter after the transformations implements a state-transition system, i.e., an abstract machine. Danvy et al. have systematically applied the transformations to a wide range of interpreters and have showed that well-known abstract machines *functionally correspond* to well-known interpreters [3–5]. Especially they have shown that the above interpreter (without support for basic constants) closure-converted on the one hand corresponds to the Krivine machine when applying a call-by-name-CPS transformation,[13] and on the other hand corresponds to the CEK machine when applying a call-by-value-CPS transformation, when followed by defunctionalization [22]. Nothing changes with the introduction of basic constants. The interpreter for **TLLF** thus functionally corresponds to the abstract machine for **TLLF** specified on page 53.

5.4 Summary

In this chapter we presented Plotkin's famous results on the correspondence between the λ_v-calculus and the SECD machine, and the correspondence between the λ-calculus and a call-by-name version of the SECD machine.

We continued with the syntactic correspondence between reduction semantics (i.e., a calculus and an associated strategy) and abstract machines investigated by Biernacka, Danvy and Nielsen. We showed how the reduction semantics in the $\lambda\hat{\rho}$-calculus for **TLLF** syntactically corresponds to the abstract machine for **TLLF**.

We finally introduced the functional correspondence between interpreters and abstract machines investigated by Ager, Biernacki, Danvy, Midtgaard, and Millikin. We outlined how the definitional interpreter for **TLLF** functionally corresponds to the abstract machine for **TLLF**.

[13] The apply function must be inlined to match the standard definition of the Krivine machine.

Chapter 6

Including imperative constructs

In the previous chapters the languages of discourse have been purely functional, i.e., with referential transparency as a central property: the 'meaning' of an expression is determined solely by the value of the expression. In this chapter we add constructs to the expression language that allow expressions to have *side-effects* in addition to representing a value. Expressions evaluated for their side-effects correspond to statements in imperative programming languages.[1]

Roadmap Our starting point is Felleisen and Hieb's introduction of state variables in the λ_v-calculus [33]. In Section 6.1 we briefly introduce state variables and assignments and continue in Section 6.2 with a formal introduction of Felleisen and Hieb's calculus — the λ_v-$S(\mathbf{t})$-calculus.

In Section 6.3 we add closure-versions of the contraction rules involving state variables of the λ_v-$S(\mathbf{t})$-calculus to the $\lambda\hat{\rho}$-calculus (Section 5.2.1) and we define an applicative-order reduction semantics in the resulting calculus. This reduction semantics has context-sensitive clauses in the one-step reduction function. In Section 6.4 we observe that the reduction semantics syntactically corresponds to the CEK machine [32] with an extra component holding the store.

6.1 State variables and assignments

Variables in the λ-calculus are placeholders for λ-terms. In β-reductions variables are replaced by λ-terms. The various semantics for **TLLF** (Chapter 4) do not perform β-reductions explicitly. Instead an environment is used to represent *bindings* of variables to denotable values. The individual bindings never change. In that respect variables are not 'variable'.

State variables We introduce a new set of variables in the expression language: *state variables*. A state variable represents at any 'time' some value, but as 'time' change, the variable can represent other values. Evaluation of a state variable yields the 'current' value of the variable. That is, evaluation become *referentially opaque*: The notion of time introduces evaluation-order dependency also for expressions that cannot possibly diverge.[2]

[1] See Section 4.2 for an introduction to programming language paradigms.

[2] Expressions that cannot possibly diverge are also known as *strongly normalizing*.

Assignments to state variables To change the binding of a variables we introduce an *assignment* construct to the expression language. Evaluation of a assignment expression lets the variable in question be bound to a new value. The new binding lasts until possibly a new assignment for that variable is evaluated.

6.2 Felleisen and Hieb's revised calculus of state

Our starting point is the λ_v-$S(\mathbf{t})$-calculus defined by Felleisen and Hieb [33].

6.2.1 The term language

The term language is a direct extension of the term language for the λ_v-calculus from Section 3.3 (We omit the definition of basic constants ranged over by b):

$$\begin{array}{lcll} \mathit{Val} & v & ::= & b \mid x \mid \lambda x.t \mid \lambda x_\sigma.t \mid \sigma x_\sigma.t \\ \mathit{Term}_\sigma & t & ::= & v \mid t\,t \mid x_\sigma \end{array}$$

Here $x \in \mathit{Var}$ and $x_\sigma \in \mathit{Var}_\sigma$. *Var* is the set of variables of the λ-calculus, and Var_σ is the set of state variables. These two sets are disjoint but otherwise unspecified.

The term language extends the λ_v-calculus language by two constructs: (1) The new set of state variables Var_σ which are not values because they do not represent a single value, and (2) the σ-*capability* $\sigma x_\sigma.t$ which represents the capability to globally assign the state variable x_σ a value when applied, i.e. an assignment construct which itself is a value.

6.2.2 Conventions

In relation to the presentation of the λ_v-calculus Felleisen and Hieb mention two conventions:

1. Bound variables are always distinct from free variables in the various expressions of mathematical definitions and claims.
2. Abstractions that only differ by a renaming of bound variables are identified, e.g., $\lambda x.x \equiv \lambda y.y$.

These two conventions are taken verbatim from Felleisen and Hieb's text. The first convention gives that the β-rule can be used without the side condition that variable renaming might be needed because substitution must be *capture-free*. The second convention is the usual convention also used in this text making the above mentioned renaming possible: Textual terms are representatives of the real terms, which are the classes defined by $=_\alpha$.

6.2.3 The ρ-application

A ρ-application $\rho\theta.t$ is a useful abbreviation. It is composed of a term t and a finite function $\theta = \{(x_1, v_1), \ldots, (x_n, v_n)\}, x_i \in \mathit{Var}_\sigma$ representing a part of the state. The abbreviation is defined via two expansion rules:

$$\begin{array}{rcl} \rho\{\}.t & \equiv & t \\ \rho\{(x_1, v_1), \ldots, (x_j, v_j)\}.t & \equiv & (\lambda x_1.\ldots.\lambda x_j.(\sigma x_1.\ldots.\sigma x_j.t)\, v_1\, \ldots v_j)\,(\lambda x.x)\,\ldots(\lambda x.x) \end{array}$$

Because θ is a finite function the x_i are all different.

6.2.4 Notions of reduction

Reduction in the calculus is defined via seven contraction rules. Three of these rules rely on applicative-order reduction contexts over the revised term language:

$$E \quad ::= \quad [\,] \mid E\ t \mid v\ E$$

$$\begin{array}{rrcl}
\delta: & q\ l & \to & \mathtt{capp}\ (q,\ l) \\
\beta_v: & (\lambda x.t)\ v & \to & t\{v/x\} \\
\beta_\sigma: & (\lambda x_\sigma.t)\ v & \to & \rho\{(x_\sigma, v)\}.t \\
D: & \rho\theta \cup \{(x_\sigma, v)\}.E[x_\sigma] & \to & \rho\theta \cup \{(x_\sigma, v)\}.E[v] \\
\sigma: & \rho\theta \cup \{(x_\sigma, v')\}.E[(\sigma x_\sigma.t)\ v] & \to & \rho\theta \cup \{(x_\sigma, v)\}.E[t] \\
gc: & \rho\theta \cup \theta'.t & \to & \rho\theta'.t, \text{if } \theta \neq \{\} \text{ and } Dom(\theta) \cap \mathtt{FV}_\sigma(\rho\theta'.t) = \{\} \\
\rho_\cup: & \rho\theta'.E[\rho\theta.t] & \to & \rho\theta \cup \theta'.E[t], \text{if } \theta \neq \{\} \text{ and } \rho\theta'.E \neq [\,]
\end{array}$$

The notion of reduction **t** is defined as the union of the above seven contraction rules:

$$\mathbf{t} \quad := \quad \delta \cup \beta_v \cup \beta_\sigma \cup D \cup \sigma \cup gc \cup \rho_\cup$$

The rules δ and β_v are standard from the λ_v-calculus of Section 3.3 (with substitution straightforwardly extended to σ-capabilities).

A new state variable is introduced in rule β_σ by extending the overall state by a new part. According to the second expansion rule for ρ-applications the contractum of a β_σ-redex is $(\lambda x_\sigma.(\sigma x_\sigma.t)\ v)\ (\lambda x.x)$. That is, the introduction of state variables is *capture-free* according to the first convention in Section 6.2.2: $x_\sigma \notin \mathtt{FV}_\sigma(v)$, where $\mathtt{FV}_\sigma$ is assumed to be the straightforward variation of $\mathtt{FV}$, the function mapping a term to its free variables — here mapping to its free state variables. $\mathtt{FV}_\sigma$ also appears in the rule gc. Rules D and σ are the lookup and update of a state variable, respectively wrt. the inner-most part of the state.

Remark that in $\rho\theta \cup \theta'.t$, $\theta \cup \theta'$ denote a finite function. In the rules gc and $\rho_\cup$ the condition about the finite function not being the empty function ($\theta \neq \{\}$) is not essential. We assume these conditions are included to let the last four rules be *strongly normalizing*, i.e., using only these four rules normalization cannot possibly diverge. But in case that property is the reason for the conditions, one more condition is needed in the garbage-collection rule gc which removes an unneeded part of the state: $\theta \cup \theta'$ must imply a *partitioning* of the domain, i.e., $Dom(\theta) \cap Dom(\theta') = \{\}$. (This condition is not mentioned by Felleisen and Hieb.)

In rule $\rho_\cup$ the second condition relies on the expansion of the ρ-abbreviation and is equivalent to ($\theta' \neq \{\}$ or $E \neq [\,]$). Hence $\rho_\cup$ either 'lifts' a part of the state towards the root of the term (when $E \neq [\,]$) or $\rho_\cup$ merges two nontrivial parts of the state into one (when $E = [\,]$).

6.2.5 One-step reduction and equality

Contexts for the term language is mentioned to be modified appropriately given contexts for terms in the λ_v-calculus. The definition of contexts must hence have productions according to the new constructs:

$$C \quad ::= \quad [\,] \mid C\ t \mid t\ C \mid \lambda x.C \mid \lambda x_\sigma.C \mid \sigma x_\sigma.C$$

Felleisen and Hieb do not explicitly define one-step **t**-reduction $\to_{\mathbf{t}}$ or **t**-reduction $\to^*_{\mathbf{t}}$, but following the rest of the paper (and the usual approach also followed in this text) definitions are straightforward: One-step **t**-reduction is defined as the compatible closure of **t** according to the above grammar of contexts:

$$t \rightarrow_{\mathbf{t}} t' \quad \text{iff} \quad t = C[r], t' = C[r'], (r, r') \in \mathbf{t}$$

t-reduction $\rightarrow^*_{\mathbf{t}}$ is the reflexive transitive closure of $\rightarrow_{\mathbf{t}}$. Equality in the calculus $=_{\mathbf{t}}$ is the equivalence relation over **t**-reduction.

6.2.6 The Church-Rosser property

The standard property *Church-Rosser* of **t** is proved by Felleisen and Hieb. In other words, $\rightarrow^*_{\mathbf{t}}$ has the usual diamond property like $\rightarrow^*_{\beta}$ on page 10.

A condition is missing in connection with liftings of ρ-applications towards the root in the rule $\rho_\cup$. The following are two reduction sequences starting with the same term (where v and v' are different closed values in normal form).

$$\begin{array}{rcl}
((\lambda x_\sigma.\lambda y.x_\sigma)\, v)\; ((\lambda x_\sigma.\lambda y.y)\, v') & \rightarrow^2_{\mathbf{t}} & (\rho\{(x_\sigma, v)\}.\lambda y.x_\sigma)\; (\rho\{(x_\sigma, v')\}.\lambda y.y) \\
& \rightarrow_{\mathbf{t}} & (\rho\{(x_\sigma, v)\}.\lambda y.x_\sigma)\; \lambda y.y \\
& \rightarrow_{\mathbf{t}} & \rho\{(x_\sigma, v)\}.((\lambda y.x_\sigma)\; \lambda y.y) \\
& \rightarrow_{\mathbf{t}} & \rho\{(x_\sigma, v)\}.x_\sigma \\
& \rightarrow_{\mathbf{t}} & \rho\{(x_\sigma, v)\}.v \\
& \rightarrow_{\mathbf{t}} & v
\end{array}$$

$$\begin{array}{rcl}
((\lambda x_\sigma.\lambda y.x_\sigma)\, v)\; ((\lambda x_\sigma.\lambda y.y)\, v') & \rightarrow^2_{\mathbf{t}} & (\rho\{(x_\sigma, v)\}.\lambda y.x_\sigma)\; (\rho\{(x_\sigma, v')\}.\lambda y.y) \\
& \rightarrow_{\mathbf{t}} & \rho\{(x_\sigma, v)\}.((\lambda y.x_\sigma)\; (\rho\{(x_\sigma, v')\}.\lambda y.y)) \\
& \equiv & \rho\{(x_\sigma, v)\}.\rho\{\}.((\lambda y.x_\sigma)\; (\rho\{(x_\sigma, v')\}.\lambda y.y)) \\
& \rightarrow_{\mathbf{t}} & \rho\{(x_\sigma, v)\}.\rho\{(x_\sigma, v')\}.((\lambda y.x_\sigma)\; \lambda y.y) \\
& \rightarrow_{\mathbf{t}} & \rho\{(x_\sigma, v)\}.\rho\{(x_\sigma, v')\}.x_\sigma \\
& \rightarrow_{\mathbf{t}} & \rho\{(x_\sigma, v)\}.\rho\{(x_\sigma, v')\}.v' \\
& \rightarrow^2_{\mathbf{t}} & v'
\end{array}$$

The sample term has been reduced to two different normal forms. Hence they do not have a common reduct. In other words, the Church-Rosser property does not hold unless we add a condition to the rule $\rho_\cup$: $\mathtt{FV}_{\mathrm{E}}(E) \cap Dom(\theta) = \{\}$, where $\mathtt{FV}_{\mathrm{E}}$ maps evaluation contexts to its free state variables.[3] The condition ensures that liftings of ρ-applications in $\rho_\cup$ are *capture-free*. Usually programs are closed terms. When utilizing evaluation contexts in definitions of a one-step reduction function for a language such evaluation contexts hence cannot have free variables. In the contraction rules D, σ and gc evaluation contexts in contrast are used in a non-standard way: in the context of store-parts. Here it is possible for evaluation contexts to have free variables.

6.2.7 Standard reduction

Felleisen and Hieb define a standard reduction relation via **t**-standard evaluation contexts:

$$E \quad ::= \quad E \mid \rho\theta.E$$

How to understand this grammar of contexts is not clear. We assume the definition incorporates the applicative-order evaluation contexts also used in the contraction rules in

[3]The added condition is not implied by $\theta \cup \theta'$ which denotes a function: In general $\mathtt{FV}_{\mathrm{E}}(E) \not\subseteq \theta'$.

Section 6.2.4, such that the intended grammar reads as follows:

$$E_{\mathbf{t}} \quad ::= \quad [\,] \mid E_{\mathbf{t}}\ t \mid \nu\ E_{\mathbf{t}} \mid \rho\theta.E_{\mathbf{t}}$$

such that **t**-standard reduction is defined as the compatible closure of **t** according to the grammar of **t**-standard evaluation contexts:

$$E_{\mathbf{t}}[r] \mapsto_{\mathbf{t}} E_{\mathbf{t}}[r'] \quad \text{iff} \quad (r, r') \in \mathbf{t}$$

Remark: the above definition of evaluation contexts together with the notion of reduction does not imply unique decomposition of a non-value term (where $\mapsto_{\mathbf{t}}$ is defined) into a context and a **t**-redex. In other words, $\mapsto_{\mathbf{t}}$ does not define a function. According to Section 4.5.3, we only have a reduction semantics if $\mapsto_{\mathbf{t}}$ is a function. In general the rule gc can obviously be applied in various contexts, but removing gc is not enough to obtain a function.

6.3 An applicative-order reduction semantics with explicit substitution including the imperative constructs

In Section 5.2.1 we introduced the $\lambda\hat{\rho}$-calculus and in Section 5.2.3 we introduced a normal-order reduction semantics in that calculus. In this section we extend the $\lambda\hat{\rho}$-calculus with the imperative constructs from the previous section and define an applicative-order reduction semantics.

6.3.1 The term language

The terms correspond directly to the terms of the λ_v-$S(\mathbf{t})$-calculus, except that de Bruijn indices are used in the pure λ-calculus subset of terms. We add the ρ-application as a actual construct for closures. That is, ρ-applications are no longer abbreviations:

$$\begin{array}{lrcl}
\mathit{TValue}_{\hat{\rho}_v\mathbf{i}} & \nu_t & ::= & b \mid \lambda t \mid \lambda x_\sigma.t \mid \sigma x_\sigma.t \\
\mathit{Term}_{\hat{\rho}_v\mathbf{i}} & t & ::= & i \mid x_\sigma \mid \nu_t \mid t\ t \\
\mathit{Closure}_{\hat{\rho}_v\mathbf{i}} & c & ::= & t[s] \mid c\ c \mid \rho\theta.c \\
\mathit{Value}_{\hat{\rho}_v\mathbf{i}} & \nu & ::= & \nu_t[s] \\
\mathit{Substitution}_{\hat{\rho}_v\mathbf{i}} & s & ::= & \bullet \mid \nu \cdot s \\
\mathit{Store}_{\hat{\rho}_v\mathbf{i}} & \theta & ::= & \{\} \mid \theta \cup \{(x_\sigma, \nu)\}
\end{array}$$

Variables i are de Bruijn indices. State variables x_σ are left unspecified. Again $\theta \cup \{(x_\sigma, \nu)\}$ denote a finite function. We thus assume $x_\sigma \notin \mathit{Dom}(\theta)$.

6.3.2 The notion of reduction

The notion of reduction is an extension of $\hat{\rho}$ changed to β_v instead of β to follow Felleisen and Hieb:

$$\begin{array}{rrcl}
\beta_v: & (\lambda t)[s]\ \nu & \rightarrow & t[\nu \cdot s] \\
\iota: & i[\nu_1 \cdots \nu_j] & \rightarrow & \nu_i\ ,\ \text{if } i \leq j \\
\pi: & (t\ t')[s] & \rightarrow & t[s]\ t'[s] \\
\delta: & q[s]\ l[s'] & \rightarrow & (\mathtt{capp}\ (q,\ l))[s']
\end{array}$$

$$\hat{\rho}_v \quad := \quad \beta_v \cup \iota \cup \pi \cup \delta$$

The extension is closure versions of the contraction rules involving the imperative constructs in the λ_v-$S(\mathbf{t})$-calculus:

$$E \ ::= \ [\,] \mid E\,c \mid v\,E$$

$$\begin{array}{rrcl}
\beta_\sigma : & (\lambda x_\sigma.t)[s]\,v & \rightarrow & \rho\{(x_\sigma, v)\}.t[s] \\
D : & \rho\theta \cup \{(x_\sigma, v)\}.E[x_\sigma[s]] & \rightarrow & \rho\theta \cup \{(x_\sigma, v)\}.E[v] \\
\sigma : & \rho\theta \cup \{(x_\sigma, v')\}.E[(\sigma x_\sigma.t)[s]\,v] & \rightarrow & \rho\theta \cup \{(x_\sigma, v)\}.E[t[s]] \\
\rho_\cup : & \rho\theta'.E[\rho\theta.c] & \rightarrow & \rho\theta \cup \theta'.E[c],\ \text{if}\ \mathtt{FV}_E(E) \cap \mathit{Dom}(\theta) = \{\}
\end{array}$$

$$\mathbf{i} \ := \ \beta_\sigma \cup D \cup \sigma \cup \rho_\cup$$

The notion of reduction is thus defined as the union: $\hat{\rho}_v\mathbf{i} := \hat{\rho}_v \cup \mathbf{i}$. We have excluded the rule gc because it will not be used in the reduction semantics. Because ρ-applications are not abbreviations only the added condition of $\rho_\cup$ is needed.

6.3.3 A one-step reduction function

Because ρ-applications are not abbreviations we cannot use the rule $\rho_\cup$ to move a store-part to the root of the overall closure unless a ρ-application exists at the root. We hence assume an empty ρ-application at the root of every closure which can be established in the initial mapping from terms to closures:

$$\begin{array}{rcl}
\mathtt{toclosure}_{\hat{\rho}_v\mathbf{i}} & : & \mathit{Term}_{\hat{\rho}_v\mathbf{i}} \rightarrow \mathit{Closure}_{\hat{\rho}_v\mathbf{i}} \\
\mathtt{toclosure}_{\hat{\rho}_v\mathbf{i}}\ t & = & \rho\{\}.t[\bullet]
\end{array}$$

A one-step reduction function assures a reduction strategy. Without the rule gc we only need to decide when to use the rules where the redexes are ρ-applications. We decide to always use the $\rho_\cup$ rule if possible, such that the one-step reduction function is a gc-free version of Felleisen and Hieb's $\longmapsto_{t1}$ [33, Definition 4.7], which is iterated in the definition of evaluation.

Following applicative order and with a ρ-application at the root, a β_σ-contraction can always be followed by a $\rho_\cup$-contraction. We merge these two rules into one by defining its composition (remark how the side-condition is simplified):

$$\rho_\cup \circ \beta_\sigma : \quad \rho\theta.E[(\lambda x_\sigma.t)[s]\,v] \ \rightarrow \ \rho\theta \cup \{(x_\sigma, v)\}.E[t[s]],\ \text{if}\ x_\sigma \notin \mathtt{FV}_E(E)$$

We define evaluation contexts as either just the empty context or a standard applicative-order evaluation contexts with a ρ-application at the root:

$$E_{\hat{\rho}_v\mathbf{i}} \ ::= \ [\,] \ \mid \ \rho\theta.E$$

The compatible closure of the notion of reduction with β_σ and $\rho_\cup$ replaced by $\rho_\cup \circ \beta_\sigma$ (call it $\hat{\rho}_v\mathbf{i}'$) wrt. these evaluation contexts defines a one-step reduction function:

$$E_{\hat{\rho}_v\mathbf{i}}[r] \longmapsto E_{\hat{\rho}_v\mathbf{i}}[r'] \quad \text{iff} \quad (r, r') \in \hat{\rho}_v\mathbf{i}'$$

The one-step reduction function is well-defined: $\hat{\rho}_v$ is a function. If $\longmapsto$ is defined for c, c can be uniquely decomposed into an evaluation context ρθ.E and a $\hat{\rho}_v$-redex *or* (exclusively) c itself is the left-hand side of one of the added contraction rules D, σ, or $\rho_\cup \circ \beta_\sigma$. These rules match disjoint sets, and $D \cup \sigma \cup (\rho_\cup \circ \beta_\sigma)$ is hence a function. According to this explanation the one-step reduction function is restated with *context-sensitive* rules:

$$\begin{array}{rcl} \rho\theta.E[(\lambda t)[s]\, v] & \mapsto & \rho\theta.E[t[v \cdot s]] \\ \rho\theta.E[i[v_1 \cdots v_j]] & \mapsto & \rho\theta.E[v_i]\ \text{, if } i \leq j \\ \rho\theta.E[(t\, t')[s]] & \mapsto & \rho\theta.E[t[s]\, t'[s]] \\ \rho\theta.E[q[s]\, l[s']] & \mapsto & \rho\theta.E[(\mathtt{capp}\ (q,\ l))[s']] \\ \\ \rho\theta.E[(\lambda x_\sigma.t)[s]\, v] & \mapsto & \rho\theta \cup \{(x_\sigma, v)\}.E[t[s]],\ \text{if } x_\sigma \notin \mathtt{FV}_E(E) \\ \rho\theta \cup \{(x_\sigma, v)\}.E[x_\sigma[s]] & \mapsto & \rho\theta \cup \{(x_\sigma, v)\}.E[v] \\ \rho\theta \cup \{(x_\sigma, v')\}.E[(\sigma x_\sigma.t)[s]\, v] & \mapsto & \rho\theta \cup \{(x_\sigma, v)\}.E[t[s]] \end{array}$$

6.3.4 Evaluation

Evaluation of closed terms is defined via the reflexive transitive closure $\mapsto^*$ of the one-step reduction function on the initially constructed closure:

$$\mathtt{eval}\, t = (v,\ \theta) \quad \text{iff} \quad \rho\{\}.t[\bullet] \mapsto^* \rho\theta.v$$

If defined for term t evaluation yields a value and the current state.

Remark that the condition in the rule for introduction of new state variables can be removed when considering only closed terms: $x_\sigma \notin \mathtt{FV}_E(E)$ is then implied by $\theta \cup \{(x_\sigma, v)\}$ being a function, because θ is now global.

The factorial function revisited The following is an expression in an extended Scheme notation used for the implementation of the constructs added in this section. We assume to be in the context of the Church numerals $\ulcorner 0 \urcorner_c$, $\ulcorner 1 \urcorner_c$, and $\ulcorner 6 \urcorner_c$ (c0, c1, and c6), $\ulcorner \mathit{add}_1 \urcorner_c$ (csucc), $\ulcorner \mathit{mult} \urcorner_c$ (cmul) and $\ulcorner \mathit{toliteral} \urcorner_c$ (c-to-nat):

```
(let* ([cfac (lambda (n)
               (let* ([!count c0])
                 ((n (lambda (res)
                       ((sigma (!count)
                              ((cmul !count) res))
                        (csucc !count)))) c1)))])
  (c-to-nat (cfac c6)))
```

Variables starting with '!' are assumed to be state variables. σ-capabilities are realized via the new keyword sigma. In the assumed context the expression defines the factorial function on Church numerals, calculates $\ulcorner 6! \urcorner_c$, and converts to the built-in literal $\ulcorner 720 \urcorner$. The iterated function is a 'lambda-dropped' version of the iterative factorial function from Section 2.2.4: The incremented numeral (!count) is a state variable updated via assignments. The iterative version from Section 2.2.4 threads this numeral instead along with the intermediate results 0!, 1!, 2!, . . . , denoted by res.

6.4 Derivation of a corresponding abstract machine

6.4.1 Introduction of explicit decomposition and plugging

In the previous section the well-definedness of $\mapsto$ hinges on a unique-decomposition property. As developed in Section 5.2 we can explicitly define a function `decomp` which decomposes every closure not on the form $\rho\theta.v$ (denoted by $\mathit{Result}_{\hat\rho_v\mathbf{i}}$) into a context and a closure uniquely denoting a potential redex:

$$\mathit{PotRedex}_{\hat\rho_v\mathbf{i}} \qquad p \;::=\; i[s] \mid x_\sigma[s] \mid (t\,t)[s] \mid v\,v$$

We represent the evaluation contexts inside-out separated into the standard applicative contexts and the store context:

$$\begin{array}{lrcl} & E[\,] & ::= & [\,] \mid E[[\,]\,c] \mid E[v\,[\,]] \\ \mathit{EContext}_{\hat\rho_v\mathbf{i}} & E_{\hat\rho_v\mathbf{i}}[\,] & ::= & (E[\,],\ \theta) \end{array}$$

The definition of `decomp` follows the same template as the decomposition in the reduction semantics for **TLLF** as explained in Section 5.2 on the syntactic correspondence between reduction semantics and abstract machines. We omit the details. Plugging a closure into an evaluation context is defined via induction over applicative-order contexts and is denoted by `plug`. We again omit the details.

We thus have:

$$\begin{array}{rcl} \texttt{decomp} & : & \mathit{Closure}_{\hat\rho_v\mathbf{i}} \to \mathit{Result}_{\hat\rho_v\mathbf{i}} + (\mathit{PotRedex}_{\hat\rho_v\mathbf{i}} \times \mathit{EContext}_{\hat\rho_v\mathbf{i}}) \\ \texttt{plug} & : & \mathit{EContext}_{\hat\rho_v\mathbf{i}} \times \mathit{Closure}_{\hat\rho_v\mathbf{i}} \to \mathit{Closure}_{\hat\rho_v\mathbf{i}} \end{array}$$

We let closures in $\mathit{Result}_{\hat\rho_v\mathbf{i}}$ be fixed points of `decomp`. The evaluation function `eval` from Section 6.3.4 is reformulated with an explicit use of `decomp` and `plug`, exactly as developed in Section 5.2.4:

$$\begin{array}{rcl} \texttt{eval} & : & \mathit{Term}_{\hat\rho_v\mathbf{i}} \to \mathit{Value}_{\hat\rho_v\mathbf{i}} \times \mathit{Store}_{\hat\rho_v\mathbf{i}} \\ \texttt{eval}\,t & = & \texttt{it}\,(\texttt{decomp}\,(\rho\{\}.t[\bullet])) \\ \\ \texttt{it} & : & \mathit{Result}_{\hat\rho_v\mathbf{i}} + (\mathit{PotRedex}_{\hat\rho_v\mathbf{i}} \times \mathit{EContext}_{\hat\rho_v\mathbf{i}}) \to \mathit{Value}_{\hat\rho_v\mathbf{i}} \times \mathit{Store}_{\hat\rho_v\mathbf{i}} \\ \texttt{it}\,\rho\theta.v & = & (v,\ \theta) \\ \texttt{it}\,(i[v_1 \cdots v_i \cdots v_j],\ (E[\,],\ \theta)) & = & \texttt{it}\,(\texttt{decomp}\,(\texttt{plug}\,((E[\,],\ \theta),\ v_i))) \\ \texttt{it}\,((\lambda t)[s]\,v,\ (E[\,],\ \theta)) & = & \texttt{it}\,(\texttt{decomp}\,(\texttt{plug}\,((E[\,],\ \theta),\ t[v \cdot s]))) \\ \texttt{it}\,((t\,t')[s],\ (E[\,],\ \theta)) & = & \texttt{it}\,(\texttt{decomp}\,(\texttt{plug}\,((E[\,],\ \theta),\ t[s]\,t'[s]))) \\ \texttt{it}\,(q[s]\,l[s'],\ (E[\,],\ \theta)) & = & \texttt{it}\,(\texttt{decomp}\,(\texttt{plug}\,((E[\,],\ \theta),\ (\texttt{capp}\,(q,\ l))[s']))) \\ \texttt{it}\,((\lambda x_\sigma.t)[s]\,v,\ (E[\,],\ \theta)) & = & \texttt{it}\,(\texttt{decomp}\,(\texttt{plug}\,((E[\,],\ \theta \cup \{(x_\sigma, v)\}),\ t[s]))) \\ \texttt{it}\,(x_\sigma[s],\ (E[\,],\ \theta \cup \{(x_\sigma, v)\})) & = & \texttt{it}\,(\texttt{decomp}\,(\texttt{plug}\,((E[\,],\ \theta \cup \{(x_\sigma, v)\}),\ v))) \\ \texttt{it}\,((\sigma x_\sigma.t)[s]\,v,\ (E[\,],\ \theta \cup \{(x_\sigma, v')\})) & = & \texttt{it}\,(\texttt{decomp}\,(\texttt{plug}\,((E[\,],\ \theta \cup \{(x_\sigma, v)\}),\ t[s]))) \end{array}$$

6.4.2 Obtaining a syntactically corresponding abstract machine

Introducing a refocus function and perform fusion We follow the method of the syntactic correspondence (presented in Section 5.2) and introduce an efficient refocusing function `refocus` extensionally equivalent to the composition of `decomp` and `plug`. Again, `refocus` is immediate by construction of `decomp`, and we omit the details. Fusing `it` and `refocus` yields an abstract machine.

Short-circuiting transition and flattening of machine configurations Short-circuiting transitions of the obtained abstract machine let us eliminate the fourth rule of `it` which is the only place where the construction of closures of the form c c is used. We hence flatten the configurations of the abstract machine:

$$
\begin{array}{rcl}
\mathit{Configuration}_{eval} & = & \mathit{Term}_{\hat{\rho}_v\mathbf{i}} \times \mathit{Substitution}_{\hat{\rho}_v\mathbf{i}} \times \mathit{EContext}_{\hat{\rho}_v\mathbf{i}} \times \mathit{Store}_{\hat{\rho}_v\mathbf{i}} \\
\mathit{Configuration}_{apply} & = & \mathit{EContext}_{\hat{\rho}_v\mathbf{i}} \times \mathit{Store}_{\hat{\rho}_v\mathbf{i}} \times \mathit{Value}_{\hat{\rho}_v\mathbf{i}} \\
\mathit{Configuration}_{terminal} & = & \mathit{Value}_{\hat{\rho}_v\mathbf{i}} \times \mathit{Store}_{\hat{\rho}_v\mathbf{i}} \\
\mathit{Configuration} & = & \mathit{Configuration}_{eval} + \mathit{Configuration}_{apply} + \mathit{Configuration}_{terminal}
\end{array}
$$

We observe that the resulting abstract machine syntactically corresponding to the original applicative-order reduction semantics is the CEK machine extended with a store component and rules for the imperative constructs evaluating terms in $\mathit{Term}_{\hat{\rho}_v\mathbf{i}}$:

$$
\begin{array}{rcl}
\mathtt{run} & : & \mathit{Term}_{\hat{\rho}_v\mathbf{i}} \to \mathit{Configuration}_{terminal} \\
\mathtt{run}\; t & = & \mathtt{eval}\,(t,\ \bullet,\ [\,],\ \{\}) \\
\\
\mathtt{eval} & : & \mathit{Configuration}_{eval} \to \mathit{Configuration}_{terminal} \\
\mathtt{eval}\,(v_t,\ s,\ E[\,],\ \theta) & = & \mathtt{apply}\,(E[\,],\ \theta,\ v_t[s]) \\
\mathtt{eval}\,(i,\ v_1 \cdots v_i \cdots v_j,\ E[\,],\ \theta) & = & \mathtt{apply}\,(E[\,],\ \theta,\ v_i) \\
\mathtt{eval}\,(x_\sigma,\ s,\ E[\,],\ \theta \cup \{(x_\sigma, v)\}) & = & \mathtt{apply}\,(E[\,],\ \theta \cup \{(x_\sigma, v)\},\ v) \\
\mathtt{eval}\,(t\,t',\ s,\ E[\,],\ \theta) & = & \mathtt{eval}\,(t,\ s,\ E[[\,]\,t'[s]],\ \theta) \\
\\
\mathtt{apply} & : & \mathit{Configuration}_{apply} \to \mathit{Configuration}_{terminal} \\
\mathtt{apply}\,([\,],\ \theta,\ v) & = & (v,\ \theta) \\
\mathtt{apply}\,(E[[\,]\,t[s]],\ \theta,\ v) & = & \mathtt{eval}\,(t,\ s,\ E[v\,[\,]],\ \theta) \\
\mathtt{apply}\,(E[q[s]\,[\,]],\ \theta,\ l[s']) & = & \mathtt{apply}\,(E[\,],\ \theta,\ (\mathtt{capp}\,(q,\ l))[s']) \\
\mathtt{apply}\,(E[(\lambda t)[s]\,[\,]],\ \theta,\ v) & = & \mathtt{eval}\,(t,\ v \cdot s,\ E[\,],\ \theta) \\
\mathtt{apply}\,(E[(\lambda x_\sigma.t)[s]\,[\,]],\ \theta,\ v) & = & \mathtt{eval}\,(t,\ s,\ E[\,],\ \theta \cup \{(x_\sigma, v)\}) \\
\mathtt{apply}\,(E[(\sigma x_\sigma.t)[s]\,[\,]],\ \theta \cup \{(x_\sigma, v')\},\ v) & = & \mathtt{eval}\,(t,\ s,\ E[\,],\ \theta \cup \{(x_\sigma, v)\})
\end{array}
$$

Obtaining a corresponding evaluator The derived machine is in defunctionalized form. Refunctionalizing yields an evaluator in CPS threading the store component. We leave out the straightforward transformation.

6.5 Summary

We extended the terms of the λ-calculus to include imperative constructs and used an existing calculus: the λ_v-S($\mathbf{t}$)-calculus defined by Felleisen and Hieb. The notions of reduction $\mathbf{t}$ has a non-standard use of evaluation contexts in that they can occur in contexts and in general have free variables. Usually evaluation contexts are used in the definition of one-step reduction functions for closed expression and cannot have free variables. We noticed that in order to be Church-Rosser the rule $\rho_\cup$ must include a condition regarding the free variables of evaluation contexts.

We defined an applicative-order reduction semantics in the $\lambda\hat{\rho}$-calculus extended according to the imperative constructs of λ_v-S($\mathbf{t}$)-calculus. A one-step reduction function appeared to include context-sensitive rules. Via introduction of refocusing we outlined how the reduction semantics syntactically corresponds to an abstract machine in defunctionalized form. The machine is the CEK machine extended to cope with the imperative constructs in the term language of the λ_v-S($\mathbf{t}$)-calculus. The abstract machine maintains a store component.

The resulting abstract machine is in defunctionalized form: The specification is in the image of defunctionalization which makes it straightforward to refunctionalize and direct-style transform the specification and thereby obtain an evaluator threading a store component.

Part II

Strong normalization

Chapter 7

Strong normalization with *actual substitution*

Our concern for this chapter and the following chapters is strong normalization, i.e., normalization to full normal form. We consider the λ-calculus as defined in Chapter 1.

Roadmap In Section 1.5 we presented a restricted compatible closure for the λ-calculus in the definition of the one-step reduction function $\mapsto_{\bar{n}}$ defining *normal-order reduction to normal forms*. In Section 1.3 we saw how compatibility rules can be stated in terms of a grammar of contexts. In Section 4.5.3 on reduction semantics we saw how a grammar of reduction contexts is used to define a restricted compatibility identified to correspond to *normal-order reduction to weak head normal forms*. In this chapter we define $\mapsto_{\bar{n}}$ as a reduction semantics for normalization to full β-normal forms.

7.1 Obtaining a reduction semantics

First attempt A first attempt to define $\mapsto_{\bar{n}}$ via reduction contexts is to take the compatible closure according to the following naive definition of reduction contexts.

$$\textit{Context}_{\bar{n}} \qquad C_{\bar{n}} \; ::= \; [\,] \mid C_{\bar{n}}\ t \mid a\ C_{\bar{n}} \mid \lambda x.C_{\bar{n}} \qquad \text{*WRONG*}$$

where a is the nonterminal from the grammar of normal forms found in Section 1.2. Compared to the grammar of normal-order reduction to weak head normal form, two extra productions are needed: (1) right-hand sub-terms of applications must be normalized if the left-hand sub-term cannot be reduced to an abstraction (which suggests production $a\ C_{\bar{n}}$) and (2) the body of abstractions must be normalized too (which suggests production $\lambda x.C_{\bar{n}}$).

The problem is that the above grammar of reduction contexts together with the notion of reduction β cannot in general determine a unique decomposition of a term t not in normal form into a reduction context $C_{\bar{n}}$ and a β-redex r, such that $t = C_{\bar{n}}[r]$. In other words, the one-step reduction function would not be well-defined when using the above grammar of reduction contexts, and determinism is therefore not ensured. A counter example is the sample term from Chapter 1:

$$(\lambda x.(\lambda y.y)\ z)\ ((\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x)))$$

Both the whole term and the sub-term $(\lambda y.y)\, z$ can be the β-redex according to the grammar of reduction contexts. The problem is the second production in the naive grammar above: Normalizing an application starts by normalizing the left-hand sub-term. But because this sub-term is in operator position a contraction of the application is needed, if the left-hand sub-term (weak head) normalizes to an abstraction. Put differently, normalization of the left-hand sub-term should only proceed until either a normal form that is not an abstraction is obtained (the corresponding context now takes the form $a\ C_{\bar{n}}$) *or* an abstraction is obtained — the body of the abstraction should in that case not be normalized before the contraction of the application.

Second attempt We introduce an extra nonterminal to exclude the possibility to normalize the body of abstractions when the abstraction is the left-hand sub-term of an application:

$$\begin{array}{lrcl} & A_{\bar{n}} & ::= & [\,] \mid A_{\bar{n}}\ t \mid a\ C_{\bar{n}} \\ \textit{Context}_{\bar{n}} & C_{\bar{n}} & ::= & A_{\bar{n}} \mid \lambda x.C_{\bar{n}} \end{array}$$

This stratified grammar of contexts and the notion of reduction β imply that all terms can be uniquely decomposed into a reduction context and a β-redex. Equivalently to the definition in Section 1.5, the one-step reduction function can hence be defined by

$$C_{\bar{n}}[(\lambda x.t)\ t'] \quad \mapsto_{\bar{n}} \quad C_{\bar{n}}[t\{t'/x\}]$$

According to Section 4.5.3, with values defined as the β-normal forms, we have obtained a reduction semantics.

7.2 Deriving a corresponding abstract machine

The grammar of contexts specified in Section 7.1 represents contexts 'outside-in'. The same contexts can equivalently be represented 'inside-out':

$$\begin{array}{lrcl} & A_{\bar{n}}[\,] & ::= & [\,] \mid C_{\bar{n}}[a\ [\,]] \mid A_{\bar{n}}[\lambda x.[\,]] \\ \textit{Context}_{\bar{n}} & C_{\bar{n}}[\,] & ::= & A_{\bar{n}}[\,] \mid C_{\bar{n}}[[\,]\ t] \end{array}$$

It is straightforward to give explicit definitions of plugging a term into a context represented 'inside-out' and the corresponding decomposition function of a term not in normal form into a reduction context and a β-redex. The mechanical introduction of an efficient refocus function (as described in Section 5.2.5) directly let us switch from reduction-based to reduction-free strong normalization and obtain a corresponding 'machine' — exactly as demonstrated in Section 5.2 for weak head normalization.

Primarily for one reason, the derivation is not presented here: The obtained 'machine' is not exactly in the right shape, according to the definition of abstract machines found in Section 4.5.2. The problem is the actual substitution in forming the contractum of a β-redex. Even though the actual substitution is a total operation, it is not elementary according to our use: It consists of a recursive descent on a term, which can be an arbitrarily big tree structure.

To transform the obtained 'machine' into the shape of an abstract machine, the recursive descent of the substitution must be incorporated explicitly into the transition function of the machine, such that no meta construction is used. One possibility is to let the machine enter a 'substitution mode' when considering the contraction of a β-redex $(\lambda x.t)\ t'$. In that mode a complete substitution is performed (in elementary steps). When t has been traversed and every substitution performed the 'machine' leaves the 'substitution mode' and continues with the result of the complete substitution in the contraction-time context. Such a complete substitution would need to traverse the tree structure t to guarantee the substitution is properly implemented. In general, when normalizing a term some sub-terms will be repeatedly traversed before (possibly) a normal form is reached. Even when the substitution variable x is not used in t, the traversal will be performed. In other words, a better solution would be to delay the substitutions until needed to eliminate unneeded and repeated traversals of terms.

7.3 Summary

In this chapter we showed how to define a reduction semantics for strong normalization in the λ-calculus utilizing actual substitution in forming the contractum of a β-redex. We outlined that we via the syntactic correspondence could derive a 'machine'.

Migrating to a calculus with *explicit substitutions* instead of *actual substitution* the meta-construction for substitution is eliminated, and substitutions are delayed since the contraction of a β-redex explicitly represents the substitution in the term without performing it. Within such a calculus with explicit substitutions we can redo the definition of a reduction semantics for strong normalization and apply the syntactic correspondence to obtain an abstract machine for strong normalization. We go into details in the next chapter.

Chapter 8

Strong normalization with *explicit substitutions*

According to the discussion in the previous chapter we consider strong normalization in connection with explicit substitutions. The topic of this chapter is *motivated* calculi with explicit substitutions and reduction semantics for strong normalization of λ-terms via these calculi.

Roadmap We show how the syntactic correspondence (introduced for weak head normalization in Section 5.2) also apply considering strong normalization: Motivated by the implementation of the contraction rule β_{deB} for de Bruijn-indexed λ-terms (Section 8.1.1) we define a new calculus with singleton substitutions explicit in the term language — the $\lambda\hat{s}$-calculus (Section 8.1.2). The $\lambda\hat{s}$-calculus is very simple. Standard properties of the calculus are implied by construction.

With the $\lambda\hat{s}$-calculus as starting point we define a normal-order reduction semantics facilitating reduction-based strong normalization of λ-terms (Section 8.1.3). We then show that the syntactic correspondence between reduction semantics and abstract machines also apply considering strong normalization (Section 8.1.4).

Motivated by some properties of the derived abstract machine we revise the $\lambda\hat{s}$-calculus to operate on lists of substitutions instead of singletons. We denote the calculus $\lambda\hat{\hat{s}}$ (Section 8.2.1). We adjust the reduction semantics and present the corresponding abstract machine (Section 8.2.2).

8.1 Strong normalization via the $\lambda\hat{s}$-calculus

8.1.1 Motivation

In Section 1.6.2 we explicitly stated a definition of the substitution mechanism underlying the β_{deB}-contraction rule for de Bruijn-indexed λ-terms. For convenience we repeat the definition:

$$\beta_{deB}: \quad (\lambda t)\, t' \quad \to \quad \texttt{substitute}\,(t,\ (1, t'))$$

$$\begin{array}{rcl}
\texttt{substitute} & : & \mathit{Term}_{deB} \times (\mathit{Index} \times \mathit{Term}_{deB}) \to \mathit{Term}_{deB} \\
\texttt{substitute}\,(i,\ (j, t)) & = & \begin{cases} i, & \text{if } i < j \\ \texttt{reindex}\,(t,\ (1,\ j)) & \text{if } i = j \\ i - 1, & \text{if } i > j \end{cases} \\
\texttt{substitute}\,(\lambda t,\ (j, t')) & = & \lambda(\texttt{substitute}\,(t,\ (j+1, t'))) \\
\texttt{substitute}\,(t\,t',\ (j, t'')) & = & (\texttt{substitute}\,(t,\ (j, t''))) \,(\texttt{substitute}\,(t',\ (j, t'')))
\end{array}$$

$$\begin{array}{rcl}
\texttt{reindex} & : & \mathit{Term}_{deB} \times (\mathit{Index} \times \mathit{Index}) \to \mathit{Term}_{deB} \\
\texttt{reindex}\,(i,\ (j, g)) & = & \begin{cases} i, & \text{if } i < j \\ i + g - 1, & \text{if } i \geq j \end{cases} \\
\texttt{reindex}\,(\lambda t,\ (j, g)) & = & \lambda(\texttt{reindex}\,(t,\ (j+1, g))) \\
\texttt{reindex}\,(t\,t',\ (j, g)) & = & (\texttt{reindex}\,(t,\ (j, g))) \,(\texttt{reindex}\,(t',\ (j, g)))
\end{array}$$

Observing the common pattern in `reindex` and `substitute` in their treatment of abstractions and substitutions (and partly variables) a merging of `reindex` into the definition of `substitute` is immediate:

$$\begin{array}{rrcl}
\mathit{Index} & i, j, g & ::= & \{1, 2, 3, \ \ldots\} \\
 & d & ::= & t \mid G\ g \\
\mathit{Substitution}_{\hat{s}} & s & ::= & [j, d]
\end{array}$$

$$\beta_{deB}: \quad (\lambda t)\, t' \quad \to \quad \texttt{substitute}'\,(t,\ [1, t'])$$

$$\begin{array}{rcll}
\texttt{substitute}' & : & \mathit{Term}_{deB} \times \mathit{Substitution}_{\hat{s}} \to \mathit{Term}_{deB} & \\
\texttt{substitute}'\,(i,\ [j, d]) & = & i, & \text{if } i < j \\
\texttt{substitute}'\,(i,\ [j, t]) & = & \begin{cases} \boxed{\texttt{substitute}'\,(t,\ [1, G\ j])} & \text{if } i = j \\ i - 1, & \text{if } i > j \end{cases} & \\
\texttt{substitute}'\,(i,\ [j, G\ g]) & = & i + g - 1, & \text{if } i \geq j \\
\texttt{substitute}'\,(\lambda t,\ [j, d]) & = & \lambda(\texttt{substitute}'\,(t,\ [j+1, d])) & \\
\texttt{substitute}'\,(t\,t',\ [j, d]) & = & (\texttt{substitute}'\,(t,\ [j, d])) \,(\texttt{substitute}'\,(t',\ [j, d])) &
\end{array}$$

A substitution $[j, d]$ consists of a de Bruijn index and an associated term t or a 'tagged' index $G\ g$ (denoting a relative de Bruijn level). Instead of using `reindex` when a term is substituted for an index, `substitute'` is applied with a substitution $[1, G\ j]$ to reindex free variables in the substituted term (see the gray box). Such a substitution is eventually matched by the first or third clause of `substitute'`.

8.1.2 The $\lambda\hat{s}$-calculus

The term language Instead of having the above meta construction to describe the corresponding contractum of a β_{deB}-redex, we introduce a new construct in the term language. Together with contraction rules for transformation and elimination of instances of that construct according to the definition of the meta construction the need for `substitute'` is eliminated. The grammar of terms thus reads:

$$\mathit{Term}_{\hat{s}} \qquad t \quad ::= \quad i \mid \lambda t \mid t\,t \mid t[j, d]$$

where $[j, d]$ is a substitution as defined above. Compared to the λ-calculus one extra syntactic construct is added. This construct associates a substitution with a term, which itself can contain other substitutions.[1] The set of terms from the λ-calculus (with terms represented with de Bruijn indices) is included as a subset of $\mathit{Term}_{\hat{s}}$.

Notion of reduction Interpreting all occurrences of `substitute` as the use of the constructor for the new construct in the term language, the contraction rules of the calculus are evident: The new language construct is introduced in a new version of the β rule ($\beta_{\hat{s}}$). Distribution, transformation, and elimination of the new construct is performed according to the definition of `substitute'`:

$$
\begin{array}{rrcll}
\beta_{\hat{s}}: & (\lambda t)\, t' & \rightarrow & t[1, t'] & \\
\iota_d: & i[j, d] & \rightarrow & i, & \text{if } i < j \\
\iota_t: & i[j, t] & \rightarrow & \begin{cases} t[1, G\ j], & \text{if } i = j \\ i - 1, & \text{if } i > j \end{cases} & \\
\iota_g: & i[j, G\ g] & \rightarrow & i + g - 1, & \text{if } i \geq j \\
\xi: & (\lambda t)[j, d] & \rightarrow & \lambda t[j+1, d] & \\
\pi: & (t\, t')[j, d] & \rightarrow & t[j, d]\, t'[j, d] &
\end{array}
$$

The union of the six contraction rules constitutes the notion of reduction in the $\hat{s}$-calculus:[2] $\hat{s} := \beta_{\hat{s}} \cup \iota_d \cup \iota_t \cup \iota_g \cup \xi \cup \pi$.

One-step $\hat{s}$-reduction, $\hat{s}$-reduction, and equality We define *one-step $\hat{s}$-reduction* $\rightarrow_{\hat{s}}$ as the compatible closure of $\hat{s}$ according to a grammar of contexts:

$$\mathit{Context}_{\hat{s}} \qquad C \quad ::= \quad [\,] \mid C\ t \mid t\ C \mid \lambda C \mid C[j, d] \mid t[j, C]$$

The $\lambda\hat{s}$-calculus is strong as opposed to the $\lambda\hat{\rho}$-calculus from Section 5.2.1, which is weak: In the $\lambda\hat{\rho}$-calculus reductions are not allowed inside the body of abstractions. In the $\lambda\hat{s}$-calculus the grammar for compatibility include two productions related to the new construct in the term language: Reductions can be performed both in the term-part and in the associated substitution-part. Reductions can hence be performed in all parts of a term.

The reflexive transitive closure of one-step $\hat{s}$-reduction defines *$\hat{s}$-reduction* $\rightarrow^*_{\hat{s}}$. Taking the symmetric closure of $\hat{s}$-reduction defines *$\hat{s}$-equality* $=_{\hat{s}}$, in the calculus.

Properties of the $\lambda\hat{s}$-calculus As presented above the motivation for the contraction rules is an implementation of the β-substitution for de Bruijn-indexed λ-terms. Assume a term contains no substitution constructs and one such is introduced via the use of $\beta_{\hat{s}}$ on redex $(\lambda t)\, t'$. Via the rules ξ and π the substitution can be distributed to all variables of t. Uses of the rules ι_d and ι_t at all these variable places either eliminate the substitution or introduce a new substitution $t'[1, G\ j]$ with a tagged index instead. This substitution can be distributed to all variables of t' (again via ξ and π) and eliminated via ι_d or ι_g. At this point the term (say t'') is a de Bruijn-indexed λ-term containing no substitutions. All contractions are done according to `substitute'`. Hence, `substitute'`$(t,\ [1, t']) = t''$. In other words β-reductions can be simulated by $\hat{s}$-reductions. From the Church-Rosser property of β_{deB} (page 17 and page 10) it thus follows that $\hat{s}$ is Church-Rosser on the subset of terms containing no substitutions. We conjecture $\hat{s}$ is Church-Rosser on all terms in $\mathit{Term}_{\hat{s}}$.[3]

[1] In contrast, closures in the $\lambda\hat{\rho}$-calculus only associates substitutions with standard λ-terms.

[2] We denote the notion of reduction $\hat{s}$ to indicate that the $\lambda\hat{s}$-calculus belong to the 'λs-calculus family' [41,42].

[3] Being Church-Rosser on $\mathit{Term}_{\hat{s}}$ include Church-Rosser on terms containing free de Bruijn indices, i.e., $\hat{s}$ is then Church-Rosser also on open terms. This property is in an explicit-substitution setting usually called *meta confluence*.

Two substitutions cannot interfere with each other. In particular the order of substitutions never changes and it is clear from the contraction rules that $\hat{s}\backslash\beta_{\hat{s}}$ is strongly normalizing.

Normal forms The normal forms in the $\lambda\hat{s}$-calculus are the normal forms of the λ-calculus with de Bruijn-indexed λ-terms as defined on page 1.6.2:

$$\begin{array}{rccl} \mathit{ANForm}_{deB} & a & ::= & i \mid a\,n \\ \mathit{NForm}_{deB} & n & ::= & a \mid \lambda n \end{array}$$

Substitutions are introduced in the contraction of a $\beta_{\hat{s}}$-redex, and substitutions can always be eliminated according to the above discussion. With also no more $\beta_{\hat{s}}$-redexes left the term is a β_{deB}-normal form containing no substitutions.

A sample series of reductions A sample series of one-step reductions on the term $(\lambda(\lambda 34(\lambda 32))4)1$ obtaining the corresponding normal form reads as follows (underlining the contacted redex at each step):

$$\begin{array}{rcl}
(\lambda(\lambda 3\,4\,(\lambda 3\,2))\,4)\,1 & \to_{\hat{s}} & (\lambda(3\,4\,(\lambda 3\,2))[1,4])\,1 \\
& \to_{\hat{s}} & (\lambda(3\,4)[1,4]\,(\lambda 3\,2)[1,4])\,1 \\
& \to_{\hat{s}} & (\lambda(3\,4)[1,4]\,(\lambda(3\,2)[2,4]))\,1 \\
& \to_{\hat{s}} & (\lambda(3\,4)[1,4]\,(\lambda 3[2,4]\,2[2,4]))\,1 \\
& \to_{\hat{s}} & (\lambda(3\,4)[1,4]\,(\lambda 2\,2[2,4]))\,1 \\
& \to_{\hat{s}} & (\lambda(3\,4)[1,4]\,(\lambda 2\,4[1,G\,2]))\,1 \\
& \to_{\hat{s}} & (\lambda(3\,4)[1,4]\,(\lambda 2\,5))\,1 \\
& \to_{\hat{s}} & (\lambda 3[1,4]\,4[1,4]\,(\lambda 2\,5))\,1 \\
& \to_{\hat{s}} & (\lambda(2\,4[1,4])\,(\lambda 2\,5))\,1 \\
& \to_{\hat{s}} & (\lambda 2\,3\,(\lambda 2\,5))\,1 \\
& \to_{\hat{s}} & (2\,3\,(\lambda 2\,5))[1,1] \\
& \to_{\hat{s}} & (2\,3)[1,1]\,(\lambda 2\,5)[1,1] \\
& \to_{\hat{s}} & 2[1,1]\,3[1,1]\,(\lambda 2\,5)[1,1] \\
& \to_{\hat{s}} & 1\,3[1,1]\,(\lambda 2\,5)[1,1] \\
& \to_{\hat{s}} & 1\,2\,(\lambda 2\,5)[1,1] \\
& \to_{\hat{s}} & 1\,2\,(\lambda(2\,5)[2,1]) \\
& \to_{\hat{s}} & 1\,2\,(\lambda 2[2,1]\,5[2,1]) \\
& \to_{\hat{s}} & 1\,2\,(\lambda 1[1,G\,2]\,5[2,1]) \\
& \to_{\hat{s}} & 1\,2\,(\lambda 2\,5[2,1]) \\
& \to_{\hat{s}} & 1\,2\,(\lambda 2\,4)
\end{array}$$

Correspondence with the λ-calculus The set of terms Term_{deB} is a subset of the terms in the $\lambda\hat{s}$-calculus. A translation of terms from the λ-calculus to terms in the $\lambda\hat{s}$-calculus is therefore not needed.

Translating from terms in the $\lambda\hat{s}$-calculus to de Bruijn-indexed λ-terms is a bit more involved. Like in the translation from closures in the $\lambda\hat{\rho}$-calculus to de Bruijn-indexed λ-terms (Section 5.2.2), all the delayed substitutions contained as the substitution-part must be forced into actual substitutions. By construction, occurrences of the substitution construct can be eliminated by normalizing using the notion of reduction $\hat{s}\backslash\beta_{\hat{s}}$: Normalization without use of $\beta_{\hat{s}}$ yields terms with no substitutions.

Alternatively, we can map to de Bruijn-indexed λ-terms deterministically via a modified version of `substitute′`: `substitute′` was defined on terms containing no substitutions and therefore a 'driver' is needed to eliminate all substitutions in both the term and the substitution itself:

$$
\begin{array}{rcl}
\sigma_{\hat{s}} & : & \mathit{Term}_{\hat{s}} \to \mathit{Term}_{deB} \\
\sigma_{\hat{s}}\,(i) & = & i \\
\sigma_{\hat{s}}\,(\lambda t) & = & \lambda(\sigma_{\hat{s}}\,t) \\
\sigma_{\hat{s}}\,(t\,t') & = & (\sigma_{\hat{s}}\,t)\,(\sigma_{\hat{s}}\,t') \\
\sigma_{\hat{s}}\,(i[j,d]) & = & \mathtt{aux}\,(i,\ [j,d]) \\
\sigma_{\hat{s}}\,((\lambda t)[j,d]) & = & \lambda(\sigma_{\hat{s}}\,(t[j+1,d])) \\
\sigma_{\hat{s}}\,((t\,t')[j,d]) & = & (\sigma_{\hat{s}}\,(t[j,d]))\,(\sigma_{\hat{s}}\,(t'[j,d])) \\
\sigma_{\hat{s}}\,(t[j',d'][j,d]) & = & \mathtt{aux}\,(\sigma_{\hat{s}}\,(t[j',d']),\ [j,d])
\end{array}
$$

$$
\begin{array}{rcll}
\mathtt{aux} & : & \mathit{Term}_{deB} \times \mathit{Substitution}_{\hat{s}} \to \mathit{Term}_{deB} & \\
\mathtt{aux}\,(i,\ [j,d]) & = & i, & \text{if } i < j \\
\mathtt{aux}\,(i,\ [j,t]) & = & \sigma_{\hat{s}}\,(t[1, G\,j]), & \text{if } i = j \\
\mathtt{aux}\,(i,\ [j,t]) & = & i-1, & \text{if } i > j \\
\mathtt{aux}\,(i,\ [j,G\,g]) & = & i+g-1, & \text{if } i \geq j \\
\mathtt{aux}\,(\lambda t,\ [j,d]) & = & \lambda(\mathtt{aux}\,(t,\ [j+1,d])) & \\
\mathtt{aux}\,(t\,t',\ [j,d]) & = & (\mathtt{aux}\,(t,\ [j,d]))\,(\mathtt{aux}\,(t',\ [j,d])) &
\end{array}
$$

The auxiliary `aux` is handling the actual substitution of one delayed substitution in a de Bruijn-indexed λ-term. We conjecture the following property holds for $t \in \mathit{Term}_{deB}$ (with $\to^*_{\beta_{deB}}$ defined in Section 1.6.3):

$$t \to^*_{\hat{s}} t' \implies t \to^*_{\beta_{deB}} \sigma_{\hat{s}}(t')$$

8.1.3 A normal-order reduction semantics for the $\lambda\hat{s}$-calculus

A grammar of reduction contexts can be defined like in the case of actual substitution, presented in Chapter 7, to define a normal-order reduction strategy for $\lambda\hat{s}$-terms (performing strong normalization). Only one extra nonterminal is needed to cope with the introduction of explicit substitutions. The defining grammar (represented inside-out) reads:

$$
\begin{array}{lrcl}
\mathit{AContext}_{\hat{s}} & A[\,] & ::= & [\,] \mid C[a\,[\,]] \mid A[\lambda[\,]] \\
\mathit{CContext}_{\hat{s}} & C[\,] & ::= & A[\,] \mid C[[\,]\,t] \\
\mathit{DContext}_{\hat{s}} & D[\,] & ::= & C[\,] \mid D[[\,][j,d]]
\end{array}
$$

The hole of the context is immediately inside the term-part of a series (possibly empty) of the new substitution construct. This series is inside a reduction context, $C[\,] \in \mathit{CContext}_{\hat{s}}$, which was introduced in Section 7.2 considering strong normalization with actual substitution.

Values and a one-step reduction function We define a reduction semantics to normal forms and not only to weak head normal forms. Therefore the values of the reduction semantics are the normal forms of the calculus.

The notions of reduction (defined in Section 8.1.2) together with the above grammar of reduction contexts does not imply a unique decomposition of a term t not in normal form into a context $D[\,]$ and a $\hat{s}$-redex r such that $t = D[r]$. The term $t = ((\lambda 1)\ 2)[2,3]$ is an illustrative example:

$$
\begin{array}{ll}
t = D_1[r_1], & D_1 = [\,] \text{ and } r_1 = ((\lambda 1)\ 2)[2,3] \\
t = D_2[r_2], & D_2 = [[\,][2,3]] \text{ and } r_2 = (\lambda 1)\ 2
\end{array}
$$

Here r_1 is a π-redex and r_2 is a $\beta_{\hat{s}}$-redex. A one-step reduction function can nevertheless be defined by allowing the rule $\beta_{\hat{s}}$ to apply only in contexts from *CContext*$_{\hat{s}}$. In the above example this restriction rules out the second decomposition $t = D_2[r_2]$. A unique decomposition into a context $C[\,]$ and a β-redex *or* a context $D[\,]$ and a $\hat{s}$-redex that is not a $\beta_{\hat{s}}$-redex is ensured. By construction of the grammar and the contraction rules it is an *exclusive or*: both situations cannot occur. The following definition of a one-step reduction function is hence well-defined: (The format was introduced in Section 4.5.3)

$$\begin{array}{rcll}
C[(\lambda t)\ t'] & \overset{n}{\mapsto}_{\hat{s}} & C[t[1, t']] & \\
D[i[j, d]] & \overset{n}{\mapsto}_{\hat{s}} & D[i], & \text{if } i < j \\
D[i[j, t]] & \overset{n}{\mapsto}_{\hat{s}} & \begin{cases} D[t[1, G\ j]], & \text{if } i = j \\ D[i-1], & \text{if } i > j \end{cases} & \\
D[i[j, G\ g]] & \overset{n}{\mapsto}_{\hat{s}} & D[i+g-1], & \text{if } i \geq j \\
D[(\lambda t)[j, d]] & \overset{n}{\mapsto}_{\hat{s}} & D[\lambda t[j+1, d]] & \\
D[(t\ t')[j, d]] & \overset{n}{\mapsto}_{\hat{s}} & D[t[j, d]\ t'[j, d]] &
\end{array}$$

This function is partial because it is not defined on values, i.e., normal forms.

Normalization of de Bruijn-indexed λ-terms The implicit decomposition is into a context and an actual redex, and $\overset{n}{\mapsto}_{\hat{s}}$ is hence defined for all terms not in normal form. In other words, it is well-defined to define normalization in terms of the reflexive transitive closure of $\overset{n}{\mapsto}_{\hat{s}}$ (denoted $\overset{n}{\mapsto}^{*}_{\hat{s}}$):

$$\texttt{normalize}_{(\hat{s},n)}\ t = n \quad \text{iff} \quad t \overset{n}{\mapsto}^{*}_{\hat{s}} n$$

This function strongly normalizes terms by a series of one-step reductions following a normal-order reduction strategy. $\texttt{normalize}_{(\hat{s},n)}$ is partial because it is not defined on terms with no normal form. Because $\hat{s}\backslash\beta_{\hat{s}}$ is strongly normalizing — i.e., for all terms, there exists no infinite reduction sequences starting with that term when not using the $\beta_{\hat{s}}$ contraction rule — terms with no normal form must contain at least one $\beta_{\hat{s}}$-redex.

Normalization follows standard reduction in the λ-calculus extended to consume substitutions. Considering the subset *Term*$_{deB}$, we hence conjecture a standardization theorem holds saying that $\texttt{normalize}_{(\hat{s},n)}$ yields a normal form, when one exists.

Sample normalization following the normal-order strategy Normalization of the term from above, $(\lambda(\lambda 3\ 4\ (\lambda 3\ 2))\ 4)\ 1$, following the normal-order reduction strategy reads (underlining the contracted redex at each step):

$$
\begin{array}{rcl}
(\lambda(\lambda 3\,4\,(\lambda 3\,2))\,4)\,1 & \overset{n}{\mapsto}_{\hat{s}} & ((\lambda(3\,4)\,(\lambda 3\,2))\,4)[1,1] \\
& \overset{n}{\mapsto}_{\hat{s}} & ((\lambda(3\,4)\,(\lambda 3\,2))[1,1]\,4[1,1] \\
& \overset{n}{\mapsto}_{\hat{s}} & (\lambda((3\,4)\,(\lambda 3\,2))[2,1])\,4[1,1] \\
& \overset{n}{\mapsto}_{\hat{s}} & ((3\,4)\,(\lambda 3\,2))[2,1][1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & ((3\,4)[2,1]\,(\lambda 3\,2)[2,1])[1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & (3\,4)[2,1][1,4[1,1]]\,(\lambda 3\,2)[2,1][1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & (3[2,1]\,4[2,1])[1,4[1,1]]\,(\lambda 3\,2)[2,1][1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & 3[2,1][1,4[1,1]]\,4[2,1][1,4[1,1]]\,(\lambda 3\,2)[2,1][1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & 2[1,4[1,1]]\,4[2,1][1,4[1,1]]\,(\lambda 3\,2)[2,1][1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,4[2,1][1,4[1,1]]\,(\lambda 3\,2)[2,1][1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,3[1,4[1,1]]\,(\lambda 3\,2)[2,1][1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda 3\,2)[2,1][1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda(3\,2)[3,1])[1,4[1,1]] \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda(3\,2)[3,1][2,4[1,1]]) \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda(3[3,1]\,2[3,1])[2,4[1,1]]) \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda 3[3,1][2,4[1,1]]\,2[3,1][2,4[1,1]]) \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda 1[1,G\,3][2,4[1,1]]\,2[3,1][2,4[1,1]]) \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda 3[2,4[1,1]]\,2[3,1][2,4[1,1]]) \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda 2\,2[3,1][2,4[1,1]]) \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda 2\,2[2,4[1,1]]) \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda 2\,4[1,1][1,G\,2]) \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda 2\,3[1,G\,2]) \\
& \overset{n}{\mapsto}_{\hat{s}} & 1\,2\,(\lambda 2\,4)
\end{array}
$$

Remark: we have normalized the open de Bruijn-indexed λ-term to its equivalent de Bruijn-indexed normal form.

8.1.4 Reduction-free strong normalization in the $\lambda\hat{s}$-calculus

The normalization function from Section 8.1.3, is reformulated according to the description in Section 4.5.3:

$$
\begin{array}{rcll}
\texttt{normalize}_{(\hat{s},n)} & : & \mathit{Term}_{\hat{s}} \to \mathit{NForm}_{deB} & \\
\texttt{normalize}_{(\hat{s},n)}\ t & = & \texttt{iterate}\ t & \\
& & & \\
\texttt{iterate} & : & \mathit{Term}_{\hat{s}} \to \mathit{NForm}_{deB} & \\
\texttt{iterate}\ n & = & n & \\
\texttt{iterate}\ C[(\lambda t)\ t'] & = & \texttt{iterate}\ C[t[1,t']] & \\
\texttt{iterate}\ D[i[j,d]] & = & \texttt{iterate}\ D[i], & \text{if } i < j \\
\texttt{iterate}\ D[i[j,t]] & = & \texttt{iterate}\ D[t[1,G\ j]], & \text{if } i = j \\
\texttt{iterate}\ D[i[j,t]] & = & \texttt{iterate}\ D[i-1], & \text{if } i > j \\
\texttt{iterate}\ D[i[j,G\ g]] & = & \texttt{iterate}\ D[i+g-1], & \text{if } i \geq j \\
\texttt{iterate}\ D[(\lambda t)[j,d]] & = & \texttt{iterate}\ D[\lambda t[j+1,d]] & \\
\texttt{iterate}\ D[(t\ t')[j,d]] & = & \texttt{iterate}\ D[t[j,d]\ t'[j,d]] &
\end{array}
$$

Again an advanced pattern matching is exploited. In other words, the above definition implicitly on the left-hand side relies on a (unique) decomposition of terms not in normal form into a contexts and a redex. On the right-hand side it implicitly relies on a plugging of terms into contexts.

Introducing an explicit definition of the decomposition and the plugging, a refocus function is mechanically obtained by use of the standard method explained in Section 5.2.5. Fusing `iterate` and the refocus function yields an abstract machine. Optimization by short-circuiting transitions yields the following abstract machine:

$$
\begin{array}{rcll}
\texttt{normalize}_{(\hat{s},n)} & : & \mathit{Term}_{\hat{s}} \rightarrow \mathit{NForm}_{deB} & \\
\texttt{normalize}_{(\hat{s},n)}\ t & = & \texttt{refocus}\,(t,\ [\,]) & \\
\\
\texttt{refocus} & : & \mathit{Term}_{\hat{s}} \times \mathit{DContext}_{\hat{s}} \rightarrow \mathit{NForm}_{deB} & \\
\texttt{refocus}\,(i,\ D[\,]) & = & \texttt{aux}\,(D[\,],\ i) & \\
\texttt{refocus}\,(\lambda t,\ D[\,]) & = & \texttt{aux}\,(D[\,],\ \lambda t) & \\
\texttt{refocus}\,(t\,t',\ D[\,]) & = & \texttt{aux}\,(D[\,],\ t\,t') & \\
\texttt{refocus}\,(t[j,d],\ D[\,]) & = & \texttt{refocus}\,(t,\ D[[\,][j,d]]) & \\
\\
\texttt{aux}_D & : & \mathit{DContext}_{\hat{s}} \times \mathit{Term}_{\hat{s}} \rightarrow \mathit{NForm}_{deB} & \\
\texttt{aux}_D\,(C[\,],\ i) & = & \texttt{aux}_C\,(C[\,],\ i) & \\
\texttt{aux}_D\,(C[\,],\ \lambda t) & = & \texttt{aux}_C\,(C[\,],\ \lambda t) & \\
\texttt{aux}_D\,(C[\,],\ t\,t') & = & \texttt{refocus}\,(t,\ C[[\,]\,t']) & \\
\texttt{aux}_D\,(D[[\,][j,d]],\ t\,t') & = & \texttt{aux}_D\,(D[\,],\ t[j,d]\,t'[j,d]) & \\
\texttt{aux}_D\,(D[[\,][j,d]],\ \lambda t) & = & \texttt{aux}_D\,(D[\,],\ \lambda t[j+1,d]) & \\
\texttt{aux}_D\,(D[[\,][j,d]],\ i) & = & \texttt{aux}_D\,(D[\,],\ i), & \text{if } i < j \\
\texttt{aux}_D\,(D[[\,][j,t]],\ i) & = & \texttt{refocus}\,(t,\ D[[\,][1,G\,j]]), & \text{if } i = j \\
\texttt{aux}_D\,(D[[\,][j,t]],\ i) & = & \texttt{aux}_D\,(D[\,],\ i-1), & \text{if } i > j \\
\texttt{aux}_D\,(D[[\,][j,G\,g]],\ i) & = & \texttt{aux}_D\,(D[\,],\ i+g-1), & \text{if } i \geq j \\
\\
\texttt{aux}_C & : & \mathit{CContext}_{\hat{s}} \times (\mathit{ANForm}_{deB} + \mathit{Term}_{\hat{s}}) \rightarrow \mathit{NForm}_{deB} & \\
\texttt{aux}_C\,(A[\,],\ a) & = & \texttt{aux}_A\,(A[\,],\ a) & \\
\texttt{aux}_C\,(A[\,],\ \lambda t) & = & \texttt{refocus}\,(t,\ A[\lambda[\,]]) & \\
\texttt{aux}_C\,(C[[\,]\,t],\ a) & = & \texttt{refocus}\,(t,\ C[a\,[\,]]) & \\
\texttt{aux}_C\,(C[[\,]\,t'],\ \lambda t) & = & \texttt{refocus}\,(t,\ C[[\,][1,t']]) & \\
\\
\texttt{aux}_A & : & \mathit{AContext}_{\hat{s}} \times \mathit{NForm}_{deB} \rightarrow \mathit{NForm}_{deB} & \\
\texttt{aux}_A\,([\,],\ n) & = & n & \\
\texttt{aux}_A\,(C[a\,[\,]],\ n) & = & \texttt{aux}_C\,(C[\,],\ a\,n) & \\
\texttt{aux}_A\,(A[\lambda[\,]],\ n) & = & \texttt{aux}_A\,(A[\,],\ \lambda n) &
\end{array}
$$

Normalization no longer consists of explicitly constructing a series of intermediate terms equivalent to the initial term, i.e., normalization is reduction free.

8.2 Strong normalization via the $\lambda\hat{\hat{s}}$-calculus

Motivation All dispatches on contexts involving substitutions are performed in $\texttt{aux}_D$. According to the last rule of `refocus` and the rules of $\texttt{aux}_D$ all available substitutions are moved to the context when refocusing on a unknown term and all are moved back into the term, e.g., when distributing substitutions in applications and into the body of abstractions. This property indicates that the representation of substitutions as part of the context might not be ideal.

On the one hand each substitution is treated in isolation from the rest of the substitutions. This property is a direct consequence of the term language and the notion of reduction in the

$\lambda\widehat{s}$-calculus. On the other hand, e.g., when distributing a substitution all adjacent substitutions are distributed in the same way one at a time.

Instead of letting the above machine undergo surgery we adjust our starting point — the $\lambda\widehat{s}$-calculus: We introduce lists of substitutions in the term language and alter the contraction rules accordingly.

8.2.1 The $\lambda\widehat{\widehat{s}}$-calculus

The abstract syntax of the $\lambda\widehat{\widehat{s}}$-calculus only differ in the substitutions. Here substitutions are lists of substitution elements instead of exactly one substitution element as was the case in the $\lambda\widehat{s}$-calculus:

$$\begin{array}{lrcl} \mathit{Term}_{\widehat{\widehat{s}}} & t & ::= & i \mid \lambda t \mid t\,t' \mid t[s] \\ \mathit{Substitution}_{\widehat{\widehat{s}}} & s & ::= & \bullet \mid (j, d) \cdot s \end{array}$$

Here i, j, and d range over the same sets as in the $\lambda\widehat{s}$-calculus. The notions of reduction is defined as $\widehat{\widehat{s}} = \beta_{\widehat{\widehat{s}}} \cup \mu \cup \iota'_d \cup \iota'_t \cup \iota'_g \cup \xi' \cup \pi' \cup \gamma$, where the eight contraction rules are defined as follows:

$$\begin{array}{rrcll} \beta_{\widehat{\widehat{s}}}: & (\lambda t)\, t' & \to & t[(1, t') \cdot \bullet] & \\ \mu: & i[\bullet] & \to & i & \\ \iota'_d: & i[(j, d) \cdot s] & \to & i[s], & \text{if } i < j \\ \iota'_t: & i[(j, t) \cdot s] & \to & \begin{cases} t[(1, G\,j) \cdot s], \\ (i - 1)[s], \end{cases} & \begin{array}{l} \text{if } i = j \\ \text{if } i > j \end{array} \\ \iota'_g: & i[(j, G\,g) \cdot s] & \to & (i + g - 1)[s], & \text{if } i \geq j \\ \xi': & (\lambda t)[s] & \to & \lambda t[\Upsilon(s)] & \\ \pi': & (t\,t')[s] & \to & t[s]\, t'[s] & \\ \gamma: & t[s'][s] & \to & t[s' \oplus s] & \end{array}$$

The lifting operation of the contraction rule ξ' is the straightforward generalization of the lifting of a singleton substitution in rule ξ of the $\lambda\widehat{s}$-calculus:

$$\begin{array}{rcl} \Upsilon(\bullet) & = & \bullet \\ \Upsilon((j, d) \cdot s) & = & (j + 1, d) \cdot \Upsilon(s) \end{array}$$

Compared to $\widehat{s}$, basically two new rules are added: Rule μ consumes empty substitution lists, and γ merges two substitution lists into one. Concatenation of two lists is denoted by infix $\oplus$.

Each of the contraction rules of the $\lambda\widehat{s}$-calculus has a list-based version in the $\lambda\widehat{\widehat{s}}$-calculus. One-step $\widehat{\widehat{s}}$-reduction, $\widehat{\widehat{s}}$-reduction, and $\widehat{\widehat{s}}$-equality, and a correspondence with the λ-calculus are defined mutatis mutandis — we straightforwardly adjust the corresponding definition in the $\lambda\widehat{s}$-calculus to cope with the new set of contraction rules and terms.

Again, the order of substitutions never change. The difference from the $\lambda\widehat{\widehat{s}}$-calculus consists in the ability to distribute more substitutions in one contraction. In other words, the normal forms of the $\lambda\widehat{\widehat{s}}$-calculus are the β_{deB}-normal forms of the λ-calculus and various properties (e.g., Church-Rosser of $\widehat{\widehat{s}}$) are inherited from the $\lambda\widehat{s}$-calculus.

8.2.2 Obtaining an efficient abstract machine

Like in Section 8.1.2, we define a normal-order reduction strategy for the $\lambda\widehat{\widehat{s}}$-calculus that yields full normal forms, and we present the syntactically corresponding abstract machine derived via introduction of a refocus function and transition compression via short-circuiting. Afterwards, two inefficiencies are identified and removed from the obtained abstract machine.

Obtaining an abstract machine We define a normal-order reduction semantics again using the grammar of reduction contexts presented in Chapter 7 in connection to strong normalization with actual substitution:

$$\begin{array}{lrcl} \mathit{AContext}_{\hat{\hat{s}}} & A[\,] & ::= & [\,] \mid C[a\,[\,]] \mid A[\lambda[\,]] \\ \mathit{CContext}_{\hat{\hat{s}}} & C[\,] & ::= & A[\,] \mid C[[\,]\,t] \end{array}$$

For the $\lambda\hat{\hat{s}}$-calculus no extra nonterminal is needed. The values of the reduction semantics are again the normal forms.

The notions of reduction $\hat{\hat{s}}$ and the above grammar of reduction contexts together satisfy that a non-value term t, i.e., a term not in normal form, can be uniquely decomposed into a reduction context $C[\,]$ and a $\hat{\hat{s}}$-redex r, such that $t = C[r]$. Because $\hat{\hat{s}}$ is a function it is hence well-defined to specify a one-step reduction function as the compatible closure of $\hat{\hat{s}}$:

$$C[r] \overset{n}{\mapsto}_{\hat{\hat{s}}} C[r'] \quad \text{iff} \quad (r, r') \in \hat{\hat{s}}$$

and the corresponding normalization function (where $\overset{n}{\mapsto}{}^{*}_{\hat{\hat{s}}}$ denotes the reflexive transitive closure of $\overset{n}{\mapsto}_{\hat{\hat{s}}}$):

$$\texttt{normalize}_{(\hat{\hat{s}},n)}\ t = n \quad \text{iff} \quad t \overset{n}{\mapsto}{}^{*}_{\hat{\hat{s}}}\ n$$

An abstract machine is obtained via the syntactic correspondence (Section 5.2):

$$\begin{array}{rcll}
\texttt{normalize}_{(\hat{\hat{s}},n)} & : & \mathit{Term}_{\hat{\hat{s}}} \to \mathit{NForm}_{deB} & \\
\texttt{normalize}_{(\hat{\hat{s}},n)}\ t & = & \texttt{refocus}\,(t,\ [\,]) & \\
 & & & \\
\texttt{refocus} & : & \mathit{Term}_{\hat{\hat{s}}} \times \mathit{CContext}_{\hat{\hat{s}}} \to \mathit{NForm}_{deB} & \\
\texttt{refocus}\,(i,\ C[\,]) & = & \texttt{aux}_C\,(C[\,],\ i) & \\
\texttt{refocus}\,(\lambda t,\ C[\,]) & = & \texttt{aux}_C\,(C[\,],\ \lambda t) & \\
\texttt{refocus}\,(t\,t',\ C[\,]) & = & \texttt{refocus}\,(t,\ C[[\,]\,t']) & \\
\texttt{refocus}\,(i[\bullet],\ C[\,]) & = & \texttt{aux}_C\,(C[\,],\ i) & \\
\texttt{refocus}\,(i[(j,d)\cdot s],\ C[\,]) & = & \texttt{refocus}\,(i[s],\ C[\,]), & \text{if } i < j \\
\texttt{refocus}\,(i[(j,t)\cdot s],\ C[\,]) & = & \texttt{refocus}\,(t[(1, G\ j)\cdot s],\ C[\,]), & \text{if } i = j \\
\texttt{refocus}\,(i[(j,t)\cdot s],\ C[\,]) & = & \texttt{refocus}\,((i-1)[s],\ C[\,]), & \text{if } i > j \\
\texttt{refocus}\,(i[(j,G\ g)\cdot s],\ C[\,]) & = & \texttt{refocus}\,((i+g-1)[s],\ C[\,]), & \text{if } i \geq j \\
\texttt{refocus}\,((\lambda t)[s],\ C[\,]) & = & \texttt{aux}_C\,(C[\,],\ \lambda t[\Upsilon(s)]) & \\
\texttt{refocus}\,((t\,t')[s],\ C[\,]) & = & \texttt{refocus}\,(t[s],\ C[[\,]\,t'[s]]) & \\
\texttt{refocus}\,(t[s'][s],\ C[\,]) & = & \texttt{refocus}\,(t[s' \oplus s],\ C[\,]) & \\
 & & & \\
\texttt{aux}_C & : & \mathit{CContext}_{\hat{\hat{s}}} \times (\mathit{ANForm}_{deB} + \mathit{Term}_{\hat{\hat{s}}}) \to \mathit{NForm}_{deB} & \\
\texttt{aux}_C\,(A[\,],\ a) & = & \texttt{aux}_A\,(A[\,],\ a) & \\
\texttt{aux}_C\,(A[\,],\ \lambda t) & = & \texttt{refocus}\,(t,\ A[\lambda[\,]]) & \\
\texttt{aux}_C\,(C[[\,]\,t],\ a) & = & \texttt{refocus}\,(t,\ C[a\,[\,]]) & \\
\texttt{aux}_C\,(C[[\,]\,t'],\ \lambda t) & = & \texttt{refocus}\,(t[1,t'],\ C[\,]) & \\
 & & & \\
\texttt{aux}_A & : & \mathit{AContext}_{\hat{\hat{s}}} \times \mathit{NForm}_{deB} \to \mathit{NForm}_{deB} & \\
\texttt{aux}_A\,([\,],\ n) & = & n & \\
\texttt{aux}_A\,(C[a\,[\,]],\ n) & = & \texttt{aux}_C\,(C[\,],\ a\,n) & \\
\texttt{aux}_A\,(A[\lambda[\,]],\ n) & = & \texttt{aux}_A\,(A[\,],\ \lambda n) &
\end{array}$$

Here $\texttt{aux}_A$ remains unchanged from the abstract machine for strong normalization in the $\lambda\hat{s}$-calculus. $\texttt{aux}_C$ only differ in the last clause. These similarities come from the similar grammar of contexts.

A simple inspection of the transition rules justifies that if the initial term $\mathtt{t}$ is associated with an empty substitution list — i.e., exploring the equivalence $\mathtt{t} =_{\hat{\hat{s}}} \mathtt{t}[\bullet]$ — `refocus` is always applied to a term of the form $\mathtt{t}[\mathtt{s}]$. Hence the first three transition rules of `refocus` will never be matched and can safely be removed. Furthermore, restricting ourselves to normalization of terms that is also in the λ-calculus, i.e., the initial term does not contain any substitution-constructs, also the last transition rule of `refocus` can be removed by 'inlining' the rule in `refocus` where $\mathtt{i} = \mathtt{j}$ and in the third rule of $\mathtt{aux}_C$.

Hereafter `refocus` is always applied with the first component being a construction $\mathtt{t}[\mathtt{s}]$, where $\mathtt{t}$ is always a term of the λ-calculus, i.e., it itself never takes the form $\mathtt{t}[\mathtt{s}]$. Flattening the first component of `refocus` such that this transition function is always applied to a λ-term, a substitution, and a context, an abstract machine performing strong normalization of terms in the λ-calculus is obtained:

$$
\begin{array}{rcll}
\mathtt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} & \\
\mathtt{normalize}\,\mathtt{t} & = & \mathtt{refocus}\,(\mathtt{t},\ \bullet,\ [\,]) & \\
\\
\mathtt{refocus} & : & \mathit{Term}_{deB} \times \mathit{Substitution}_{\hat{\hat{s}}} \times \mathit{CContext}_{\hat{\hat{s}}} \to \mathit{NForm}_{deB} & \\
\mathtt{refocus}\,(\mathtt{i},\ \bullet,\ C[\,]) & = & \mathtt{aux}_C\,(C[\,],\ \mathtt{i}) & \\
\mathtt{refocus}\,(\mathtt{i},\ (\mathtt{j}, \mathtt{d}) \cdot \mathtt{s},\ C[\,]) & = & \mathtt{refocus}\,(\mathtt{i},\ \mathtt{s},\ C[\,]), & \text{if } \mathtt{i} < \mathtt{j} \\
\mathtt{refocus}\,(\mathtt{i},\ (\mathtt{j}, \mathtt{t}[\mathtt{s}']) \cdot \mathtt{s},\ C[\,]) & = & \mathtt{refocus}\,(\mathtt{t},\ \mathtt{s}' \oplus ((1, G\ \mathtt{j}) \cdot \mathtt{s}),\ C[\,]), & \text{if } \mathtt{i} = \mathtt{j} \\
\mathtt{refocus}\,(\mathtt{i},\ (\mathtt{j}, \mathtt{t}'[\mathtt{s}']) \cdot \mathtt{s},\ C[\,]) & = & \mathtt{refocus}\,(\mathtt{i} - 1,\ \mathtt{s},\ C[\,]), & \text{if } \mathtt{i} > \mathtt{j} \\
\mathtt{refocus}\,(\mathtt{i},\ (\mathtt{j}, G\ \mathtt{g}) \cdot \mathtt{s},\ C[\,]) & = & \mathtt{refocus}\,(\mathtt{i} + \mathtt{g} - 1,\ \mathtt{s},\ C[\,]), & \text{if } \mathtt{i} \geq \mathtt{j} \\
\mathtt{refocus}\,(\lambda \mathtt{t},\ \mathtt{s},\ C[\,]) & = & \mathtt{aux}_C\,(C[\,],\ \lambda \mathtt{t}[\Upsilon(\mathtt{s})]) & \\
\mathtt{refocus}\,(\mathtt{t}\,\mathtt{t}',\ \mathtt{s},\ C[\,]) & = & \mathtt{refocus}\,(\mathtt{t},\ \mathtt{s},\ C[[\,]\,\mathtt{t}'[\mathtt{s}]]) & \\
\\
\mathtt{aux}_C & : & \mathit{CContext}_{\hat{\hat{s}}} \times (\mathit{ANForm}_{deB} + \mathit{Term}_{\hat{\hat{s}}}) \to \mathit{NForm}_{deB} & \\
\mathtt{aux}_C\,(A[\,],\ \mathtt{a}) & = & \mathtt{aux}_A\,(A[\,],\ \mathtt{a}) & \\
\mathtt{aux}_C\,(A[\,],\ \lambda \mathtt{t}[\mathtt{s}]) & = & \mathtt{refocus}\,(\mathtt{t},\ \mathtt{s},\ A[\lambda[\,]]) & \\
\mathtt{aux}_C\,(C[[\,]\,\mathtt{t}[\mathtt{s}]],\ \mathtt{a}) & = & \mathtt{refocus}\,(\mathtt{t},\ \mathtt{s},\ C[\mathtt{a}\,[\,]]) & \\
\mathtt{aux}_C\,(C[[\,]\,\mathtt{t}'[\mathtt{s}']],\ \lambda \mathtt{t}[\mathtt{s}]) & = & \mathtt{refocus}\,(\mathtt{t},\ \mathtt{s} \oplus ((1, \mathtt{t}'[\mathtt{s}']) \cdot \bullet),\ C[\,]) & \\
\\
\mathtt{aux}_A & : & \mathit{AContext}_{\hat{\hat{s}}} \times \mathit{NForm}_{deB} \to \mathit{NForm}_{deB} & \\
\mathtt{aux}_A\,([\,],\ \mathtt{n}) & = & \mathtt{n} & \\
\mathtt{aux}_A\,(C[\mathtt{a}\,[\,]],\ \mathtt{n}) & = & \mathtt{aux}_C\,(C[\,],\ \mathtt{a}\,\mathtt{n}) & \\
\mathtt{aux}_A\,(A[\lambda[\,]],\ \mathtt{n}) & = & \mathtt{aux}_A\,(A[\,],\ \lambda \mathtt{n}) &
\end{array}
$$

The above machine has two drawbacks: (1) it relies in two transition rules on the inefficient concatenation of lists of substitutions, and (2) it relies on the inefficient lifting operation in one transition rule. Let us remove these inefficiencies.

Getting rid of the inefficient concatenation and lifting of substitutions The drawbacks cannot be removed mechanically. But it *is* possible to get rid of them via a nontrivial transformation. The drawbacks are introduced because the machine does not exploit its own normalization strategy: We can exploit invariants of substitution lists implied by the strategy.

In the rule where $\mathtt{i} = \mathtt{j}$ one of the concatenations of substitution lists occurs. One invariant gives that all real substitutions $(\mathtt{j}, \mathtt{t}')$ relevant for $\mathtt{t}$ can be found in $\mathtt{s}'$. That is, after introduction of $\mathtt{t}$ via a substitution only the adjustments of free indices of $\mathtt{t}$ are needed after the substitutions in $\mathtt{s}'$ has be consumed. Therefore, if substitution elements of the form $(\mathtt{j}, G\ \ \mathtt{g})$ (used to lift free indices) are not placed in the environment, the concatenation of substitution lists can be removed because s can be discarded. A dedicated component of the machine can hold the liftings of free variables.

The last clause of $\mathtt{aux}_C$ contains the second occurrence of the concatenation. Here a new element is added to the end of the substitution list. Immediately before that, the substitution list has been lifted in the second to last rule of `refocus`. Distributing the lifting to the two possible cases we can have an implicit lifting in the last clause of $\mathtt{aux}_C$ by placing the element in the *front* of the substitution list. By this change the second concatenation is removed.

Exploiting that the substitution lists now are never concatenated, the index in each element of the substitution elements is no longer needed: By indexing a whole list of substitutions using *one* index, the lifting degenerates to incrementing that index. Because the lifting of free indices has been removed from the substitutions, the lifting counter must also be incremented.

Instead of changing the substitution construct of terms we introduce closures and change substitution lists according to the above discussion:

$$\begin{array}{lrcl} \textit{Closure} & c & ::= & t[s, g] \\ & e & ::= & \bullet \mid c \cdot s \\ \textit{Substitution} & s & ::= & e^j \end{array}$$

where j and g are lifting indices and $t \in \textit{Term}_{deB}$. We state the abstract machine with the two drawbacks removed:[4]

$$\begin{array}{rcll}
\mathtt{normalize} & : & \textit{Term}_{deB} \to \textit{NForm}_{deB} & \\
\mathtt{normalize}\, t & = & \mathtt{refocus}\,(t,\ \bullet^1,\ 1,\ [\,]) & \\
\\
\mathtt{refocus} & : & \textit{Term}_{deB} \times \textit{Substitution} \times \textit{Index} \times \textit{Context} \to \textit{NForm}_{deB} & \\
\mathtt{refocus}\,(i,\ e^j,\ g,\ C[\,]) & = & \mathtt{aux}\,(C[\,],\ i), & \text{if } i < j \\
\mathtt{refocus}\,(i,\ (t[s', g'] \cdot s)^j,\ g,\ C[\,]) & = & \mathtt{refocus}\,(t,\ s',\ g',\ C[\,]), & \text{if } i = j \\
\mathtt{refocus}\,(i,\ (c \cdot s)^j,\ g,\ C[\,]) & = & \mathtt{refocus}\,(i - j,\ s,\ g,\ C[\,]), & \text{if } i > j \\
\mathtt{refocus}\,(i,\ \bullet^j,\ g,\ C[\,]) & = & \mathtt{aux}\,(C[\,],\ i + g - 1), & \text{if } i \geq j \\
\mathtt{refocus}\,(\lambda t,\ s,\ g,\ C[\,]) & = & \mathtt{aux}\,(C[\,],\ (\lambda t)[s, g]) & \\
\mathtt{refocus}\,(t\,t',\ s,\ g,\ C[\,]) & = & \mathtt{refocus}\,(t,\ s,\ g,\ C[[\,]\, t'[s, g]]) & \\
\\
\mathtt{aux}_C & : & \textit{Context} \times (\textit{ANForm}_{deB} + \textit{Closure}) \to \textit{NForm}_{deB} & \\
\mathtt{aux}_C\,(A[\,],\ a) & = & \mathtt{aux}_A\,(A[\,],\ a) & \\
\mathtt{aux}_C\,(A[\,],\ (\lambda t)[e^j, g]) & = & \mathtt{refocus}\,(t,\ e^{j+1},\ g + 1,\ A[\lambda[\,]]) & \\
\mathtt{aux}_C\,(C[[\,]\, t[s, g]],\ a) & = & \mathtt{refocus}\,(t,\ s,\ g,\ C[a\,[\,]]) & \\
\mathtt{aux}_C\,(C[[\,]\, c],\ (\lambda t)[s, g]) & = & \mathtt{refocus}\,(t,\ (c \cdot s)^1,\ g,\ C[\,]) & \\
\\
\mathtt{aux}_A & : & \textit{AContext} \times \textit{NForm}_{deB} \to \textit{NForm}_{deB} & \\
\mathtt{aux}_A\,([\,],\ n) & = & n & \\
\mathtt{aux}_A\,(C[a\,[\,]],\ n) & = & \mathtt{aux}_C\,(C[\,],\ a\,n) & \\
\mathtt{aux}_A\,(A[\lambda[\,]],\ n) & = & \mathtt{aux}_A\,(A[\,],\ \lambda n) &
\end{array}$$

Obtaining a corresponding normalization function The above abstract machine is identified to be in defunctionalized form wrt. the reduction contexts with $\mathtt{aux}_C$ and $\mathtt{aux}_A$ constituting the apply-function. Refunctionalization yields a normalization function in CPS.

[4]Here the definition of contexts has been adjusted to the above closures and is denoted by *Context* and *AContext*.

8.3 Summary

In this chapter the starting point was the $\lambda\hat{s}$-calculus which was defined directly from a specification of the meta construction `substitute'` for β-substitution in de Bruijn-indexed λ-terms. The contraction rules and the calculus as such are very simple: Each substitution is handled in isolation from other substitutions. The only difference from an actual use of `substitute'` is that the substitutions are delayed. This property immediately gives the simulation of β-contractions.

An analysis of the abstract machine, syntactically corresponding to a normal-order reduction strategy in the $\lambda\hat{s}$-calculus, pinpointed that the machine simulates operating on lists of substitutions. We hence adjusted the $\lambda\hat{s}$-calculus to use contraction rules for lists of substitutions and defined the corresponding normal-order reduction strategy as a reduction semantics in the $\lambda\hat{\hat{s}}$-calculus. The syntactically corresponding abstract machine has two major drawbacks:[5] (1) it *concatenates* lists of substitutions and (2) it *shifts* lists of substitutions. A final non-mechanical transformation yielded an improved abstract machine without the drawbacks.

We showed that the syntactic correspondence (previously only investigated in connection with weak normalization) also applies when considering strong normalization of λ-terms.

Applying the functional correspondence to the improved machine would let us mechanically extract the corresponding higher-order normalization function for λ-terms. This normalization function is compositional. In Chapter 10, we treat both the syntactic correspondence and the functional correspondence for strong normalization with an existing example.

[5]Because of the drawbacks the machine is actually not an abstract machine according to our definition in Section 4.5.2.

Chapter 9

Strong normalization starting from Lescanne's normalizer

Lescanne defines a normalizer performing strong normalization in his paper on explicit substitution and extensions to confluent calculi [53]. This normalizer uses an abstract machine (the $\mathtt{U}$-*machine*). This machine is weakly normalizing but can be loaded with an arbitrary environment and stack. To perform strong normalization of λ-terms Lescanne's normalizer repeatedly 'instantiates' the $\mathtt{U}$-machine by use of a 'driver loop'.[1]

Roadmap In this chapter our starting point is Lescanne's normalizer. We present this normalizer with a minimal change to let it be deterministic (Section 9.1). Simple transformations then put the complete specification into the shape of an abstract machine (Section 4.5.2). We relate that abstract machine to the derived abstract machines from Chapter 8.

9.1 Lescanne's specification made deterministic

The $\mathtt{U}$-machine The $\mathtt{U}$-machine utilizes two auxiliary structures: environments and stacks:

$$\begin{array}{lrcl} & \mathsf{i} & ::= & \{1, 2, 3, \ldots\} \\ & \mathsf{c} & ::= & \uparrow \mid (\mathsf{t}, e) \\ \mathit{Env}_{\mathtt{U}} & e & ::= & \bullet \mid (\mathsf{c}, \mathsf{i}) \cdot e \\ \mathit{Stack}_{\mathtt{U}} & \mathsf{s} & ::= & \bullet \mid (\mathsf{t}, e) \cdot \mathsf{s} \end{array}$$

Here $\mathsf{t} \in \mathit{Term}_{deB}$. In Lescanne's paper, i ranges over the natural numbers including zero. We exclude zero (because the transition rules become more clear) and adjust the definition of the $\mathtt{U}$-machine accordingly.

Configurations in the $\mathtt{U}$-machine consist of (1) a term, (2) an environment, and (3) a stack. The possible terms are de Bruijn-indexed λ-terms:

$$\mathit{Configuration}_{\mathtt{U}} := \mathit{Term}_{deB} \times \mathit{Env}_{\mathtt{U}} \times \mathit{Stack}_{\mathtt{U}}$$

[1]The use of the term 'instantiates' is taken from Lescanne's paper. The term 'driver loop' is standard.

Transliterated into our notation the U-machine reads:

$$\begin{array}{rcl} \mathtt{U} & : & \mathit{Configuration}_{\mathtt{U}} \to \mathit{Configuration}_{\mathtt{U}} \\ \mathtt{U}\,(i,\ \bullet,\ s) & = & (i,\ \bullet,\ s) \\ \mathtt{U}\,(i,\ (\uparrow,1)\cdot e,\ s) & = & \mathtt{U}\,(i+1,\ e,\ s) \\ \mathtt{U}\,(1,\ ((t,e'),1)\cdot e,\ s) & = & \mathtt{U}\,(t,\ e'\oplus e,\ s) \\ \mathtt{U}\,(i+1,\ ((t,e'),1)\cdot e,\ s) & = & \mathtt{U}\,(i,\ e,\ s) \\ \mathtt{U}\,(1,\ (c,j+1)\cdot e,\ s) & = & \mathtt{U}\,(1,\ e,\ s) \\ \mathtt{U}\,(i+1,\ (c,j+1)\cdot e,\ s) & = & \mathtt{U}\,(i,\ (c,j)\cdot(\uparrow,1)\cdot e,\ s) \\ \mathtt{U}\,(\lambda t,\ e,\ \bullet) & = & (\lambda t,\ e,\ \bullet) \\ \mathtt{U}\,(\lambda t,\ e,\ (t',e')\cdot s) & = & \mathtt{U}\,(t,\ (\mathtt{Lift\_env}\ e)\oplus(((t',e'),1)\cdot\bullet),\ s) \\ \mathtt{U}\,(t\,t',\ e,\ s) & = & \mathtt{U}\,(t,\ e,\ (t',e)\cdot s) \end{array}$$

Here Lift_env corresponds to the lifting operation Υ in the $\lambda\hat{\hat{s}}$-calculus from Section 8.2.1. The concatenation of two environments is denoted by $\oplus$.

The two kinds of terminal configurations $(\lambda t,\ e,\ \bullet)$ and $(i,\ \bullet,\ s)$ make it evident that normalization is not strong using the U-machine in isolation. A driver loop outside the machine is needed to achieve strong normalization.

Strong normalization via the U-machine Lescanne defines a driver loop for the U-machine. The complete specification for strong normalization is non-deterministic because the driver loop does not decide on an order of normalization of right-hand-side sub-terms in nested applications that will eventually appear in the final normal form. In other words, when the U-machine reaches a final state of the form $(i,\ \bullet,\ s)$, all elements on the stack s are normalized, but the order in which to perform these normalizations is not specified.

To match our specification in Section 4.5.2, abstract machines must be deterministic. We decide on left-to-right normalization order of the sub-terms in question, and implement determinism by adding an auxiliary function (aux):

$$\begin{array}{rcll} \mathtt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} & \\ \mathtt{normalize}\,t & = & \mathtt{nf}\,(t,\ \bullet) & \\ & & & \\ \mathtt{nf} & : & \mathit{Term}_{deB} \times \mathit{Env}_{\mathtt{U}} \to \mathit{NForm}_{deB} & \\ \mathtt{nf}\,(t,\ e) & = & \lambda(\mathtt{nf}\,(t',\ \mathtt{Lift\_env}\ e')), & \text{if } \mathtt{U}\,(t,\ e,\ \bullet) = (\lambda t',\ e',\ \bullet) \\ \mathtt{nf}\,(t,\ e) & = & \mathtt{aux}\,(s,\ i), & \text{if } \mathtt{U}\,(t,\ e,\ \bullet) = (i,\ \bullet,\ s) \\ & & & \\ \mathtt{aux} & : & \mathit{Stack}_{\mathtt{U}} \times \mathit{ANForm}_{deB} \to \mathit{NForm}_{deB} & \\ \mathtt{aux}\,(\bullet,\ a) & = & a & \\ \mathtt{aux}\,((t,e)\cdot s,\ a) & = & \mathtt{aux}\,(s,\ a\,(\mathtt{nf}\,(t,\ e))) & \end{array}$$

The above specification does still not match the shape of an abstract machine because it is not a state-transition system: non-tail calls exist in nf and aux.[2]

9.2 Obtaining a corresponding abstract machine

To eliminate the non-tail calls of the normalization function we follow in the steps of the functional correspondence (Section 5.3): (1) We eliminate nf by fusing it with U, (2) make the

[2]Here we ignore the lifting operation and the concatenation operation.

specification sequential by a CPS-transformation, and (3) make that CPS program first-order by defunctionalizing the continuation. The following data-structure continuation is used:

$$\textit{MCont}_{\mathtt{U}} \qquad M \;::=\; \bullet \mid (a, s) \cdot M \mid \Lambda \cdot M$$

The result is an abstract machine (where $\textit{Closure}_{\mathtt{U}}$ abbreviates $\textit{Term}_{deB} \times \textit{Env}_{\mathtt{U}}$):

$$\begin{array}{rcl}
\mathtt{normalize} & : & \textit{Term}_{deB} \rightarrow \textit{NForm}_{deB} \\
\mathtt{normalize}\, t & = & \mathtt{U}\,(t,\ \bullet,\ \bullet,\ \bullet) \\[1em]
\mathtt{U} & : & \textit{Term}_{deB} \times \textit{Env}_{\mathtt{U}} \times \textit{Stack}_{\mathtt{U}} \times \textit{MCont}_{\mathtt{U}} \rightarrow \textit{NForm}_{deB} \\
\mathtt{U}\,(i,\ \bullet,\ s,\ M) & = & \mathtt{aux}\,(s,\ i,\ M) \\
\mathtt{U}\,(i,\ (\uparrow, 1) \cdot e,\ s,\ M) & = & \mathtt{U}\,(i+1,\ e,\ s,\ M) \\
\mathtt{U}\,(1,\ ((t, e'), 1) \cdot e,\ s,\ M) & = & \mathtt{U}\,(t,\ e' \oplus e,\ s,\ M) \\
\mathtt{U}\,(i+1,\ ((t, e'), 1) \cdot e,\ s,\ M) & = & \mathtt{U}\,(i,\ e,\ s,\ M) \\
\mathtt{U}\,(1,\ (c, j+1) \cdot e,\ s,\ M) & = & \mathtt{U}\,(1,\ e,\ s,\ M) \\
\mathtt{U}\,(i+1,\ (c, j+1) \cdot e,\ s,\ M) & = & \mathtt{U}\,(i,\ (c, j) \cdot (\uparrow, 1) \cdot e,\ s,\ M) \\
\mathtt{U}\,(\lambda t,\ e,\ s,\ M) & = & \mathtt{aux}\,(s,\ (\lambda t, e),\ M) \\
\mathtt{U}\,(t\, t',\ e,\ s,\ M) & = & \mathtt{U}\,(t,\ e,\ (t', e) \cdot s,\ M) \\[1em]
\mathtt{aux} & : & \textit{Stack}_{\mathtt{U}} \times (\textit{ANForm}_{deB} + \textit{Closure}_{\mathtt{U}}) \rightarrow \textit{NForm}_{deB} \\
\mathtt{aux}\,(\bullet,\ a,\ M) & = & \mathtt{apply}\,(M,\ a) \\
\mathtt{aux}\,(\bullet,\ (\lambda t, e),\ M) & = & \mathtt{U}\,(t,\ \mathtt{Lift_env}\, e,\ \bullet,\ \Lambda \cdot M) \\
\mathtt{aux}\,((t, e) \cdot s,\ a,\ M) & = & \mathtt{U}\,(t,\ e,\ \bullet,\ (a, s) \cdot M) \\
\mathtt{aux}\,((t', e') \cdot s,\ (\lambda t, e),\ M) & = & \mathtt{U}\,(t,\ (\mathtt{Lift_env}\, e) \oplus (((t', e'), 1) \cdot \bullet),\ s,\ M) \\[1em]
\mathtt{apply} & : & \textit{MCont}_{\mathtt{U}} \times \textit{NForm}_{deB} \rightarrow \textit{NForm}_{deB} \\
\mathtt{apply}\,(\bullet,\ n) & = & n \\
\mathtt{apply}\,((a, s) \cdot M,\ n) & = & \mathtt{aux}\,(s,\ a\, n,\ M) \\
\mathtt{apply}\,(\Lambda \cdot M,\ n) & = & \mathtt{apply}\,(M,\ \lambda n)
\end{array}$$

Merging the data-structure continuation into the stack The defunctionalized continuation M is not changed in U. In aux and apply M is used as a stack. We thus incorporate the continuation into the existing stack s and thereby eliminate the unneeded component from configurations introduced by the CPS-transformation. The stack now represents four different 'contexts':

$$s \;::=\; \bullet \mid a \cdot s \mid \Lambda \cdot s \mid (t, e) \cdot s$$

Instead of using such a plain stack structure, we exploit that the first three productions are treated separately from the fourth, and represent the stack in a stratified fashion:

$$\begin{array}{lrcl}
\textit{MContext}_{\mathtt{U}} & M & ::= & \bullet \mid a \cdot s \mid \Lambda \cdot M \\
\textit{SContext}_{\mathtt{U}} & s & ::= & M \mid (t, e) \cdot s
\end{array}$$

This alternative representation lets us directly state the abstract machine without the extra component:

$$
\begin{array}{rcl}
\texttt{normalize} & : & \mathit{Term}_{deB} \rightarrow \mathit{NForm}_{deB} \\
\texttt{normalize}\, t & = & \texttt{U}\,(t,\ \bullet,\ \bullet) \\
\\
\texttt{U} & : & \mathit{Term}_{deB} \times \mathit{Env}_{\texttt{U}} \times \mathit{SContext}_{\texttt{U}} \rightarrow \mathit{NForm}_{deB} \\
\texttt{U}\,(i,\ \bullet,\ s) & = & \texttt{aux}_{S}\,(s,\ i) \\
\texttt{U}\,(i,\ (\uparrow, 1) \cdot e,\ s) & = & \texttt{U}\,(i+1,\ e,\ s) \\
\texttt{U}\,(1,\ ((t, e'), 1) \cdot e,\ s) & = & \texttt{U}\,(t,\ e' \oplus e,\ s) \\
\texttt{U}\,(i+1,\ ((t, e'), 1) \cdot e,\ s) & = & \texttt{U}\,(i,\ e,\ s) \\
\texttt{U}\,(1,\ (c, j+1) \cdot e,\ s) & = & \texttt{U}\,(1,\ e,\ s) \\
\texttt{U}\,(i+1,\ (c, j+1) \cdot e,\ s) & = & \texttt{U}\,(i,\ (c, j) \cdot (\uparrow, 1) \cdot e,\ s) \\
\texttt{U}\,(\lambda t,\ e,\ s) & = & \texttt{aux}_{S}\,(s,\ (\lambda t, e)) \\
\texttt{U}\,(t\, t',\ e,\ s) & = & \texttt{U}\,(t,\ e,\ (t', e) \cdot s) \\
\\
\texttt{aux}_{S} & : & \mathit{SContext}_{\texttt{U}} \times (\mathit{ANForm}_{deB} + \mathit{Closure}_{\texttt{U}}) \rightarrow \mathit{NForm}_{deB} \\
\texttt{aux}_{S}\,(M,\ a) & = & \texttt{aux}_{M}\,(M,\ a) \\
\texttt{aux}_{S}\,(M,\ (\lambda t, e)) & = & \texttt{U}\,(t,\ \texttt{Lift\_env}\, e,\ \Lambda \cdot M) \\
\texttt{aux}_{S}\,((t, e) \cdot s,\ a) & = & \texttt{U}\,(t,\ e,\ a \cdot s) \\
\texttt{aux}_{S}\,((t', e') \cdot s,\ (\lambda t, e)) & = & \texttt{U}\,(t,\ (\texttt{Lift\_env}\, e) \oplus (((t', e'), 1) \cdot \bullet),\ s) \\
\\
\texttt{aux}_{M} & : & \mathit{MContext}_{\texttt{U}} \times \mathit{NForm}_{deB} \rightarrow \mathit{NForm}_{deB} \\
\texttt{aux}_{M}\,(\bullet,\ n) & = & n \\
\texttt{aux}_{M}\,(a \cdot s,\ n) & = & \texttt{aux}_{S}\,(s,\ a\, n) \\
\texttt{aux}_{M}\,(\Lambda \cdot M,\ n) & = & \texttt{aux}_{M}\,(M,\ \lambda n)
\end{array}
$$

The above machine is closely related to the machine derived via refocusing starting from a normal-order reduction strategy in the $\lambda\hat{\hat{s}}$-calculus (page 95): The transition rules that dispatch on the two kinds of contexts are the same modulo the representations (and the distribution of the shift of environments). Also the elements in the substitution lists and environments are the same. Especially, both machines rely in the same way on liftings and concatenations of environments. The difference between the machines is the lookup of variables and reindexing of free variables: The U-machine treats the lookups and reindexings in 'small steps'.

9.3 Summary

Simplicity The following is a quote of Lescanne on the idea from which the machine has emanated:[3]

> If $\lambda\nu$ is simple then it should entail a conceptually simple machine for (weak and strong reduction) normalization of λ-calculus.

The paper does not describe how the calculus entails the specification of normalization using the U-machine. For a better understanding we transformed the specification into the shape of an abstract machine (Section 4.5.2) by use of simple meaning-preserving transformations. This abstract machine revealed that the specification of strong normalization via the U-machine is a cousin of the abstract machine from Section 8.2.2. Specifically, both machines rely in the same way on liftings and concatenations of environments.

[3] $\lambda\nu$ is an independently designed calculus from Lescanne's paper [53].

The cousin of Lescanne's machine was systematically derived from a normal-order reduction semantics for strong normalization in the $\lambda\hat{\hat{s}}$-calculus. The $\lambda\hat{\hat{s}}$-calculus was defined as a list-based version of the $\lambda\hat{s}$-calculus which in turn is closely related to the $\lambda\nu$-calculus. Both calculi are remarkably simple with the $\lambda\hat{s}$-calculus extracted directly from an implementation of β-substitution on de Bruijn-indexed λ-terms (Section 8.1.2). We conjecture that defining a lists-based version of the $\lambda\nu$-calculus and a normal-order reduction semantics for strong normalization in that calculus, the 'normalized' version of Lescanne's machine could be mechanically derived.

Efficiency The following is another quote of Lescanne from the paper on the drawbacks in relying on lifts and concatenations of substitution lists:

> In a good implementation, both `Lift_env` and $\oplus$ are called by need, that is they are evaluated on just the part of the environment that is necessary for enabling a further transition.

The use of liftings and concatenations is a reminiscence of treating all substitutions in the order they are introduced or detected and at the same time delaying distribution and consumption of them until needed. As demonstrated in Section 8.2.2, relying on lifting and concatenation is not essential. In fact, they are artifacts of the more general notion of terms including a construct $\mathtt{t}[s]$ in the term language. Both the abstract machine corresponding to Lescanne's normalizer and the abstract machine derived from the $\lambda\hat{\hat{s}}$-calculus do not exploit that the argument term for normalization is a de Bruijn-indexed λ-term with no substitutions, and that this knowledge together with the strategy of the machine implies important invariants for the lists of substitutions.

The same arguments as in Section 8.2.2 apply for the elimination of the inefficient artifacts from the abstract machine corresponding to Lescanne's normalizer. This elimination would give an efficient version of that abstract machine, which will be a cousin of the efficient machine from Section 8.2.2.

Chapter 10

Strong normalization starting from Curien's normalizer for strong left-most reduction

In Chapter 8, we derived abstract machines for strong normalization syntactically corresponding to normal-order strategies in the $\lambda\widehat{s}$-calculus and the $\lambda\widehat{\widehat{s}}$-calculus.

In Chapter 9, we presented Lescanne's $\mathtt{U}$-machine and strong normalization via this weak machine. We used standard transformations to put Lescanne's normalizer, once made deterministic, into the shape of an abstract machine. We noted that the machine has the drawbacks of relying on shiftings and concatenations like the machine directly derived from a reduction semantics in the $\lambda\widehat{\widehat{s}}$-calculus. The drawbacks are reminiscent to the nature of the $\lambda\nu$-calculus, the $\lambda\widehat{s}$-calculus, and the $\lambda\widehat{\widehat{s}}$-calculus: Each substitution from the root of the term to an index is consumed in the introduced order.

In this chapter our starting point is not a calculus but instead a normalizer defined independently from a calculus by Curien in his textbook [16, pages 65–66]. We show systematically, with Curien's normalizer as example, that the functional correspondence and the syntactic correspondence apply to strong normalization.

Roadmap Our roadmap is outlined in the following diagram. We start from Curien's normalizer (Section 10.1):

$$
\begin{tikzcd}
\text{Curien's normalizer} \arrow[d, "\text{Section 10.2}"'] & & & \\
\text{continuation-based normalization function} \arrow[r, "\text{functional correspondence}", "\text{Section 10.3}"'] \arrow[d, "\text{Section 10.5}"'] & \text{abstract machine w/ 2 stacks} & \text{reduction semantics w/ 2 layers of contexts} \arrow[l, "\text{syntactic correspondence}"', "\text{Section 10.4}"] \\
\text{normalization function in direct style} \arrow[r, "\text{functional correspondence}", "\text{Section 10.6}"'] & \text{abstract machine w/ 1 stack} & \text{reduction semantics w/ 1 layer of contexts} \arrow[l, "\text{syntactic correspondence}"', "\text{Section 10.7}"]
\end{tikzcd}
$$

In Section 10.2, we fuse the two components of Curien's normalizer (Section 10.2.1) using Ohori and Sasano's fixed-point promotion [58]. We then put this fused normalizer into defunctionalized form (Section 10.2.2) and refunctionalize it (Section 10.2.3). The result is a continuation-based normalization function. This normalization function is compositional. In Section 10.3, we CPS-transform the continuation-based normalization function and we defunctionalize the result, obtaining an abstract machine for strong normalization with two stacks. In Section 10.4, we present the corresponding reduction semantics. This reduction semantics uses two layers of contexts and it can be refocused. Fusing the refocusing function with the function iterating it and short-circuiting transitions, we obtain the abstract machine with two stacks. In Section 10.5, we express the compositional continuation-based normalization function from Section 10.2.3 into call-by-value direct style, using control delimiters [24]. The resulting normalization function is still compositional. Its control delimiters are moot since the normalization function does not capture continuations. In Section 10.6 we omit the control delimiters, CPS-transform this compositional normalization function in direct style and defunctionalize the result, obtaining an abstract machine for strong normalization with one stack. In Section 10.7, we present the corresponding reduction semantics; this reduction semantics uses one layer of contexts and it can be refocused into the abstract machine with one stack.

10.1 Curien's specification of strong normalization

Like Lescanne's normalizer in Chapter 9, Curien's specification of strong normalization consists of a 'weak' machine and a driver loop repeatedly initializing and running that weak machine.

10.1.1 The weak `CM`-machine

The weak machine is called 'A machine for strong leftmost reduction' in Curien's textbook. It is a Krivine-style abstract machine designed to be initialized with a term and a (lifted) environment to facilitate overall strong normalization of de Bruijn-indexed λ-terms. In the following we refer to the weak machine as the `CM`-*machine*.

Configurations of the `CM`-machine consist of a de Bruijn-indexed λ-term (Term_{deB}), a lifted environment ($\mathit{Subst}_{\mathtt{CM}}$), and a stack ($\mathit{Stack}_{\mathtt{CM}}$):[1]

$$
\begin{array}{llcl}
\mathit{Clos}_{\mathtt{CM}} & c & ::= & t[\sigma] \\
\mathit{Subst}_{\mathtt{CM}} & \sigma & ::= & \rho^{m} \\
\mathit{Env}_{\mathtt{CM}} & \rho & ::= & \mathsf{id} \mid c \cdot \sigma \\
\mathit{Stack}_{\mathtt{CM}} & S & ::= & \bullet \mid c \cdot S \\
\\
\mathit{Configuration}_{\mathtt{CM}} & & := & \mathit{Term}_{deB} \times \mathit{Subst}_{\mathtt{CM}} \times \mathit{Stack}_{\mathtt{CM}}
\end{array}
$$

Curien's definition of weak normalization with the `CM`-machine is transliterated into our

[1] Curien calls these components 'codes', 'terms', and 'stacks', respectively. We differ from Curien's terms and follow the rest of this text.

notation:

$$\begin{array}{rcl}
\texttt{CM} & : & \textit{Configuration}_{\texttt{CM}} \to \textit{Configuration}_{\texttt{CM}} \\
\texttt{CM}\,(i,\ \mathsf{id}^m,\ S) & = & (i,\ \mathsf{id}^m,\ S) \\
\texttt{CM}\,(i+1,\ (c \cdot \rho^m)^{m'},\ S) & = & \texttt{CM}\,(i,\ \rho^{m+m'},\ S) \\
\texttt{CM}\,(1,\ (t[\rho^m] \cdot \sigma)^{m'},\ S) & = & \texttt{CM}\,(t,\ \rho^{m+m'},\ S) \\
\texttt{CM}\,(t\,t',\ \sigma,\ S) & = & \texttt{CM}\,(t,\ \sigma,\ t'[\sigma] \cdot S) \\
\texttt{CM}\,(\lambda t,\ \sigma,\ \bullet) & = & (\lambda t,\ \sigma,\ \bullet) \\
\texttt{CM}\,(\lambda t,\ \sigma,\ c \cdot S) & = & \texttt{CM}\,(t,\ (c \cdot \sigma)^0,\ S)
\end{array}$$

Two possible kinds of terminal configurations exits: (1) Where the term part is an abstraction and the stack is empty $(\lambda t,\ \sigma,\ \bullet)$, and (2) where the term part is an index and the environment is empty $(i,\ \mathsf{id}^m,\ S)$. Without the lexical adjustments of environments the CM-machine coincides with the Krivine machine which defined the semantics for **TLLF** on page 53.

10.1.2 Strong normalization via the CM-machine

The driver loop initializing and repeatedly running the CM-machine to achieve strong normalization is specified via two rules:

$$\frac{\texttt{CM}\,(t,\ \sigma,\ \bullet) = (\lambda t',\ \rho^m,\ \bullet) \qquad \texttt{nf}\,(t',\ (1[\mathsf{id}^0] \cdot \rho^{m+1})^0) = n}{\texttt{nf}\,(t,\ \sigma) = \lambda n}$$

$$\frac{\texttt{CM}\,(t,\ \sigma,\ \bullet) = (i,\ \mathsf{id}^m,\ t_1[\sigma_1] \cdots t_k[\sigma_k]) \qquad \texttt{nf}\,(t_j,\ \sigma_j) = n_j,\ \text{for } 1 \le j \le k}{\texttt{nf}\,(t,\ \sigma) = (i+m)\, n_1 \ldots n_k}$$

Normalization of de Bruijn-indexed λ-term t is then defined by applying nf to t paired with the unadjusted identity environment:

$$\begin{array}{rcl}
\texttt{normalize} & : & \textit{Term}_{deB} \to \textit{NForm}_{deB} \\
\texttt{normalize}\,t & = & \texttt{nf}\,(t,\ \mathsf{id}^0)
\end{array}$$

The second rule of nf makes the normalizer non-deterministic because that rule does not decide on the order of normalization of closures on the stack S of terminal configurations of the CM-machine on the form $(i,\ \mathsf{id}^m,\ S)$. We decide (like for Lescanne's normalizer in Section 9.1) on left-to-right normalization order of the closures in question, and again implement determinism by adding an auxiliary function (aux):

$$\begin{array}{rcll}
\texttt{normalize} & : & \textit{Term}_{deB} \to \textit{NForm}_{deB} & \\
\texttt{normalize}\,t & = & \texttt{nf}\,(t,\ \mathsf{id}^0) & \\
 & & & \\
\texttt{nf} & : & \textit{Term}_{deB} \times \textit{Subst}_{\texttt{CM}} \to \textit{NForm}_{deB} & \\
\texttt{nf}\,(t,\ \sigma) & = & \lambda(\texttt{nf}\,(t',\ (1[\mathsf{id}^0] \cdot \rho^{m+1})^0)), & \text{if } \texttt{CM}\,(t,\ \sigma,\ \bullet) = (\lambda t',\ \rho^m,\ \bullet) \\
\texttt{nf}\,(t,\ \sigma) & = & \texttt{aux}\,(S,\ i+m), & \text{if } \texttt{CM}\,(t,\ \sigma,\ \bullet) = (i,\ \mathsf{id}^m,\ S) \\
 & & & \\
\texttt{aux} & : & \textit{Stack}_{\texttt{CM}} \times \textit{ANForm}_{deB} \to \textit{NForm}_{deB} & \\
\texttt{aux}\,(\bullet,\ a) & = & a & \\
\texttt{aux}\,(t[\sigma] \cdot S,\ a) & = & \texttt{aux}\,(S,\ a\,(\texttt{nf}\,(t,\ \sigma))) &
\end{array}$$

10.2 From Curien's normalizer to a continuation-based normalization function

Following in the steps of the functional correspondence (Section 5.3) we (1) fuse `nf` and `CM` to eliminate the driver loop, (2) put it in defunctionalized form with respect to the stack, and (3) refunctionalize.

10.2.1 A fused normalizer

With `nf` and `CM` fused using Ohori and Sasano's fixed-point promotion [58], the applications of `nf` are replaced by the equivalent call of the `CM`-machine with an empty stack:

$$
\begin{array}{rcl}
\texttt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{normalize}\,\mathtt{t} & = & \texttt{CM}\,(\mathtt{t},\ \mathsf{id}^0,\ \bullet) \\
\\
\texttt{CM} & : & \mathit{Configuration}_{\texttt{CM}} \to \mathit{NForm}_{deB} \\
\texttt{CM}\,(\mathtt{i},\ \mathsf{id}^m,\ \mathtt{S}) & = & \texttt{aux}\,(\mathtt{S},\ \mathtt{i}+m) \\
\texttt{CM}\,(\mathtt{i}+1,\ (\mathtt{c}\cdot\rho^m)^{m'},\ \mathtt{S}) & = & \texttt{CM}\,(\mathtt{i},\ \rho^{m+m'},\ \mathtt{S}) \\
\texttt{CM}\,(1,\ (\mathtt{t}[\rho^m]\cdot\sigma)^{m'},\ \mathtt{S}) & = & \texttt{CM}\,(\mathtt{t},\ \rho^{m+m'},\ \mathtt{S}) \\
\texttt{CM}\,(\mathtt{t}\,\mathtt{t}',\ \sigma,\ \mathtt{S}) & = & \texttt{CM}\,(\mathtt{t},\ \sigma,\ \mathtt{t}'[\sigma]\cdot\mathtt{S}) \\
\texttt{CM}\,(\lambda\mathtt{t},\ \rho^m,\ \bullet) & = & \lambda(\texttt{CM}\,(\mathtt{t},\ (1[\mathsf{id}^0]\cdot\rho^{m+1})^0,\ \bullet)) \\
\texttt{CM}\,(\lambda\mathtt{t},\ \sigma,\ \mathtt{c}\cdot\mathtt{S}) & = & \texttt{CM}\,(\mathtt{t},\ (\mathtt{c}\cdot\sigma)^0,\ \mathtt{S}) \\
\\
\texttt{aux} & : & \mathit{Stack}_{\texttt{CM}} \times \mathit{ANForm}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{aux}\,(\bullet,\ a) & = & a \\
\texttt{aux}\,(\mathtt{t}[\sigma]\cdot\mathtt{S},\ a) & = & \texttt{aux}\,(\mathtt{S},\ a\,(\texttt{CM}\,(\mathtt{t},\ \sigma,\ \bullet)))
\end{array}
$$

The non-tail call for normalizing the body of a residual abstraction and the non-tail call for normalizing the right sub-term of a residual application are the reasons why the fused normalizer is not in the shape of an abstract machine.

10.2.2 A normalizer in defunctionalized form

The fused normalizer is not in defunctionalized form with respect to its stack. The characteristic of a data type in defunctionalized form is that it is consumed solely by its apply function [26, 27]. The stack, however, is consumed both in the two clauses of `CM` for abstractions, and in `aux`. We can combine these two dispatch sites into `aux` by changing the domain to contain a sum type:

$$
\texttt{aux} \ : \ \mathit{Stack}_{\texttt{CM}} \times (\mathit{Clos}_{\texttt{CM}} + \mathit{ANForm}_{deB}) \to \mathit{NForm}_{deB}
$$

in order to yield a normalizer in defunctionalized form with respect to its stack.

We take this opportunity to split `CM` into two function `eval` and `lookup` where `lookup` interprets indices in an environment. In the definition of `lookup`, a call to `eval` with a known first argument has been short-circuited:

$$
\begin{array}{rcl}
\texttt{lookup}\,(\mathtt{i}+1,\ (\mathtt{c}\cdot\rho^m)^{m'},\ \mathtt{S}) & = & \texttt{eval}\,(\mathtt{i},\ \rho^{m+m'},\ \mathtt{S}) \\
 & = & \texttt{lookup}\,(\mathtt{i},\ \rho^{m+m'},\ \mathtt{S})
\end{array}
$$

The normalizer in defunctionalized form reads:

$$\begin{array}{rcl}
\texttt{eval} & : & \mathit{Term}_{deB} \times \mathit{Subst}_{\texttt{CM}} \times \mathit{Stack}_{\texttt{CM}} \to \mathit{NForm}_{deB} \\
\texttt{eval}\,(i,\ \sigma,\ S) & = & \texttt{lookup}\,(i,\ \sigma,\ S) \\
\texttt{eval}\,(\lambda t,\ \sigma,\ S) & = & \texttt{aux}\,(S,\ (\lambda t)[\sigma]) \\
\texttt{eval}\,(t\,t',\ \sigma,\ S) & = & \texttt{eval}\,(t,\ \sigma,\ t'[\sigma] \cdot S) \\[1ex]
\texttt{lookup} & : & \mathit{Index} \times \mathit{Subst}_{\texttt{CM}} \times \mathit{Stack}_{\texttt{CM}} \to \mathit{NForm}_{deB} \\
\texttt{lookup}\,(i,\ \mathsf{id}^m,\ S) & = & \texttt{aux}\,(S,\ i+m) \\
\texttt{lookup}\,(1,\ (t[\rho^m] \cdot \sigma)^{m'},\ S) & = & \texttt{eval}\,(t,\ \rho^{m+m'},\ S) \\
\texttt{lookup}\,(i+1,\ (c \cdot \rho^m)^{m'},\ S) & = & \texttt{lookup}\,(i,\ \rho^{m+m'},\ S) \\[1ex]
\texttt{aux} & : & \mathit{Stack}_{\texttt{CM}} \times (\mathit{Clos}_{\texttt{CM}} + \mathit{ANForm}_{deB}) \to \mathit{NForm}_{deB} \\
\texttt{aux}\,(\bullet,\ a) & = & a \\
\texttt{aux}\,(\bullet,\ (\lambda t)[\rho^m]) & = & \lambda(\texttt{eval}\,(t,\ (1[\mathsf{id}^0] \cdot \rho^{m+1})^0,\ \bullet)) \\
\texttt{aux}\,(t[\sigma] \cdot S,\ a) & = & \texttt{aux}\,(S,\ a\,(\texttt{eval}\,(t,\ \sigma,\ \bullet))) \\
\texttt{aux}\,(c \cdot S,\ (\lambda t)[\sigma]) & = & \texttt{eval}\,(t,\ (c \cdot \sigma)^0,\ S) \\[1ex]
\texttt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{normalize}\,t & = & \texttt{eval}\,(t,\ \mathsf{id}^0,\ \bullet)
\end{array}$$

10.2.3 A higher-order, continuation-based normalization function

We are now in position to refunctionalize the stack and its apply function. They represent higher-order functions of type $\mathit{Clos}_{\texttt{CM}} + \mathit{ANForm}_{deB} \to \mathit{NForm}_{deB}$. The function represented by the empty stack is recursive, because the stack $\bullet$ appears on both the left-hand and right-hand sides of the second clause of aux. Refunctionalizing the empty stack requires naming this recursive function. We recognize it as performing reification [20], and we write it as '$\downarrow$' to follow the tradition.

We can also refunctionalize the closures in substitutions. They are only consumed by the second clause of lookup: $\texttt{lookup}\,(1,\ (t[\rho^m] \cdot \sigma)^{m'},\ S) = \texttt{eval}\,(t,\ \rho^{m+m'},\ S)$. Thus, the closure $t[\rho^m]$ represents $\texttt{eval}\,(t,\ \rho^{m+m'},\ S)$ for some index m' and stack S, i.e., it represents

$$\lambda(m',\ S).\texttt{eval}\,(t,\ \rho^{m+m'},\ S).$$

One of the closures represents a function that can be simplified. The closure $1[\mathsf{id}^0]$ represents the function

$$\lambda(m',\ S).\texttt{eval}\,(1,\ \mathsf{id}^{0+m'},\ S).$$

The application of eval in the body can be short-circuited via:

$$\begin{array}{rcl}
\texttt{eval}\,(1,\ \mathsf{id}^{0+m'},\ S) & = & \texttt{lookup}\,(1,\ \mathsf{id}^{m'},\ S) \\
 & = & S\,(m'+1)
\end{array}$$

After refunctionalizing stacks and closures, source terms only appear as the arguments of continuations and they are always paired with substitutions. They are consumed in two different places, but in both places the pair of term and substitution $(\lambda t)[\rho^m]$ represents $\texttt{eval}\,(t,\ (c \cdot \rho^{m+m'})^0,\ S)$ for some refunctionalized closure c, index m', and stack S.

So, all in all, refunctionalizing the stack into the function space *Cont*, environment closures into *Thunk*, and argument closures for continuations into *Fun* yields the mutually recursive types:

$$\begin{array}{rcl} \mathit{Cont} & = & \mathit{Fun} + \mathit{ANForm}_{deB} \to \mathit{NForm}_{deB} \\ \mathit{Thunk} & = & \mathit{Index} \times \mathit{Cont} \to \mathit{NForm}_{deB} \\ \mathit{Fun} & = & \mathit{Thunk} \times \mathit{Index} \times \mathit{Cont} \to \mathit{NForm}_{deB} \end{array}$$

We let k range over *Cont*, d range over *Thunk*, and f range over *Fun*. These definitions are used in the higher-order continuation-based normalization function:

$$\begin{array}{rcl} \texttt{eval} & : & \mathit{Term}_{deB} \times \mathit{Subst}_{\texttt{CM}} \times \mathit{Cont} \to \mathit{NForm}_{deB} \\ \texttt{eval}\,(i,\ \sigma,\ k) & = & \texttt{lookup}\,(i,\ \sigma,\ k) \\ \texttt{eval}\,(\lambda t,\ \rho^m,\ k) & = & k\,(\lambda(d,\ m',\ k').\texttt{eval}\,(t,\ (d \cdot \rho^{m+m'})^0,\ k')) \\ \texttt{eval}\,(t\,t',\ \rho^m,\ k) & = & \texttt{eval}\,(t, \rho^m, \lambda v.\mathit{case}\ v \\ & & \quad \mathit{of}\ f \Rightarrow f\,(\lambda(m',\ k').\texttt{eval}\,(t',\ \rho^{m+m'},\ k'),\ 0,\ k) \\ & & \quad \mid a \Rightarrow k\,(a\,(\texttt{eval}\,(t',\ \rho^m,\ \downarrow)))) \end{array}$$

$$\begin{array}{rcl} \texttt{lookup} & : & \mathit{Index} \times \mathit{Subst}_{\texttt{CM}} \times \mathit{Cont} \to \mathit{NForm}_{deB} \\ \texttt{lookup}\,(i,\ \mathsf{id}^m,\ k) & = & k\,(i+m) \\ \texttt{lookup}\,(1,\ (d \cdot \sigma)^{m'},\ k) & = & d\,(m',\ k) \\ \texttt{lookup}\,(i+1,\ (d \cdot \rho^m)^{m'},\ k) & = & \texttt{lookup}\,(i,\ \rho^{m+m'},\ k) \end{array}$$

$$\begin{array}{rcl} \downarrow & : & \mathit{Cont} \\ \downarrow f & = & \lambda(f\,(\lambda(m,\ k).k\,(m+1),\ 1,\ \downarrow)) \\ \downarrow a & = & a \end{array}$$

$$\begin{array}{rcl} \texttt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\ \texttt{normalize}\,t & = & \texttt{eval}\,(t,\ \mathsf{id}^0,\ \downarrow) \end{array}$$

The normalization function is continuation-based, but not in continuation-passing style due to the non-tail calls to `eval`, and it is compositional. The type *Thunk* of 'denoted values' (i.e., the values associated with indices by substitutions) are recognized as CPS thunks taking a lexical adjustment in addition to a continuation. The type *Fun* of functional values are recognized as CPS functions taking a denoted value and a lexical adjustment in addition to a continuation.

Conclusion We have obtained the normalization function for untyped, de Bruijn-indexed λ-terms that underlies Curien's normalizer. It is compositional. All of the steps after the first one (making the normalizer deterministic) were mechanical, following in the steps of the functional correspondence between abstract machines and interpreters (Section 5.3).

10.3 From a continuation-based normalization function to an abstract machine with two stacks

Through the functional correspondence, we closure convert, CPS-transform and then defunctionalize the continuation-based normalization function from Section 10.2. The result is an abstract machine utilizing two stacks:

$$\begin{array}{llcl} \mathit{Cont} & C & ::= & \bullet \mid c \cdot C \\ \mathit{MetaCont} & M & ::= & \bullet \mid (a, C) \cdot M \mid \Lambda \cdot M \end{array}$$

The machine normalizes a term t by starting in the configuration $(t,\ \mathsf{id}^0,\ \bullet,\ \bullet)$. It halts with a normal form n if it reaches a configuration $(\bullet,\ n)$:

$$
\begin{array}{rcl}
\texttt{eval} & : & \mathit{Term}_{deB} \times \mathit{Subst}_{\texttt{CM}} \times \mathit{Cont} \times \mathit{MetaCont} \to \mathit{NForm}_{deB} \\
\texttt{eval}\ (i,\ \sigma,\ C,\ M) & = & \texttt{lookup}\ (i,\ \sigma,\ C,\ M) \\
\texttt{eval}\ (\lambda t,\ \sigma,\ C,\ M) & = & \texttt{apply}_C\ (C,\ (\lambda t)[\sigma],\ M) \\
\texttt{eval}\ (t\, t',\ \sigma,\ C,\ M) & = & \texttt{eval}\ (t,\ \sigma,\ t'[\sigma] \cdot C,\ M) \\
\\
\texttt{lookup} & : & \mathit{Term}_{deB} \times \mathit{Subst}_{\texttt{CM}} \times \mathit{Cont} \times \mathit{MetaCont} \to \mathit{NForm}_{deB} \\
\texttt{lookup}\ (i,\ \mathsf{id}^m,\ C,\ M) & = & \texttt{apply}_C\ (C,\ i+m,\ M) \\
\texttt{lookup}\ (1,\ (t[\rho^m] \cdot \sigma)^{m'},\ C,\ M) & = & \texttt{eval}\ (t,\ \rho^{m+m'},\ C,\ M) \\
\texttt{lookup}\ (i+1,\ (c \cdot \rho^m)^{m'},\ C,\ M) & = & \texttt{lookup}\ (i,\ \rho^{m+m'},\ C,\ M) \\
\\
\texttt{apply}_C & : & \mathit{Cont} \times (\mathit{Clos}_{\texttt{CM}} \times \mathit{ANForm}_{deB}) \times \mathit{MetaCont} \to \mathit{NForm}_{deB} \\
\texttt{apply}_C\ (\bullet,\ a,\ M) & = & \texttt{apply}_M\ (M,\ a) \\
\texttt{apply}_C\ (\bullet,\ (\lambda t)[\rho^m],\ M) & = & \texttt{eval}\ (t,\ (1[\mathsf{id}^0] \cdot \rho^{m+1})^0,\ \bullet,\ \Lambda \cdot M) \\
\texttt{apply}_C\ (t[\sigma] \cdot C,\ a,\ M) & = & \texttt{eval}\ (t,\ \sigma,\ \bullet,\ (a, C) \cdot M) \\
\texttt{apply}_C\ (t'[\sigma'] \cdot C,\ (\lambda t)[\sigma],\ M) & = & \texttt{eval}\ (t,\ (t'[\sigma'] \cdot \sigma)^0,\ C,\ M) \\
\\
\texttt{apply}_M & : & \mathit{MetaCont} \times \mathit{NForm}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{apply}_M\ (\bullet,\ n) & = & n \\
\texttt{apply}_M\ (\Lambda \cdot M,\ n) & = & \texttt{apply}_M\ (M,\ \lambda n) \\
\texttt{apply}_M\ ((a, C) \cdot M,\ n) & = & \texttt{apply}_C\ (C,\ a\, n,\ M) \\
\\
\texttt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{normalize}\, t & = & \texttt{eval}\ (t,\ \mathsf{id}^0,\ \bullet,\ \bullet)
\end{array}
$$

In Section 9.2 we CPS-transformed and defunctionalized the continuation of Lescanne's normalizer put on defunctionalized form wrt. the stack and derived an abstract machine with two stacks. Likewise, CPS-transforming and defunctionalizing the continuation of Curien's normalizer in defunctionalized form (Section 10.2.2) yields the above abstract machine with two stacks.

10.4 From a reduction semantics with two layers of contexts to an abstract machine with two stacks

The goal of this section is to present the reduction semantics that corresponds to the abstract machine derived in Section 10.3. We introduce such a reduction semantics and through the syntactic correspondence (Section 5.2), we derive the abstract machine.

10.4.1 A reduction semantics for Curien's calculus of closures

There is a reduction semantics that embodies the reduction strategy of Curien's normalizer. It is a calculus equipped with a deterministic reduction strategy that leads to a β-normal form when one exists. Though the underlying calculus will not necessarily be confluent, we can still use the reduction strategy to reduce terms to their β-normal forms.

The term language, contraction rules, and reduction contexts are mutually dependent. Together, they are chosen to ensure *unique decomposition*, i.e., that a non-normal-form term can be decomposed into a reduction context and *potential redex* (either an actual redex or a stuck term) in exactly one way. The syntax of terms is chosen so that they are closed under the contraction rules. We present first the term language, then the contraction rules, then the reduction contexts, but they do depend on each other.

Term language The configurations of the abstract machine from Section 10.3 should correspond to terms in context in the reduction semantics. The data-structure continuations of the machine are isomorphic to the reduction contexts, and thus the other components of the configurations must correspond to the terms of the reduction semantics.

The de Bruijn-indexed λ-term and substitution in the `eval` and `lookup` configurations represent closures, so the terms of the reduction semantics certainly includes the closures, c. The right-hand side of the transition

$$\texttt{eval}\ (t\,t',\ \sigma,\ C,\ M) = \texttt{eval}\ (t,\ \sigma,\ t'[\sigma]\cdot C,\ M)$$

represents the application (in context) of closures, $t[\sigma]\ t'[\sigma]$, so the terms include the application $c\ c'$ of a pair of closures. The $\texttt{apply}_C$ configurations can contain atomic forms (in context), a, so these are included in the set of terms.[2] The right-hand side of the transition

$$\texttt{apply}_C\ (\bullet,\ (\lambda t)[\rho^m],\ M) = \texttt{eval}\ (t,\ (1[\mathsf{id}^0]\cdot\rho^{m+1})^0,\ \bullet,\ \Lambda\cdot M)$$

represents an abstraction (in context) of a closure, $\lambda t[(1[\mathsf{id}^0]\cdot\rho^{m+1})^0]$, so the syntax of terms includes abstractions of closures, λc. These considerations suggest that the terms of the reduction semantics are given by an extended grammar of closures: $c ::= t[\sigma] \mid \lambda c \mid c\ c \mid a$. The $\texttt{apply}_M$ configurations contain β-normal forms (in context), n, but as can easily be checked, the β-normal forms are already given by this grammar.

This grammar of terms (together with the contraction rules and contexts) creates an overlap that causes unique decomposition to fail. Consider the term in context $C[(\lambda t)[\sigma]\ c]$. This could be a *Beta*-contraction in the context C, or it could be a contraction that moves the substitution σ inside the abstraction in the context $C[[\,]\ c]$, i.e., an *Abs*-contraction. The same idea used in defining the grammar of normal forms (page 17) resolves this overlap. We use a stratified syntax of terms that prevents abstractions from occurring as operators in applications. The term language of the reduction semantics is as follows:

$$\begin{array}{llll}
\textit{Atom} & d & ::= & t[\sigma] \mid d\ c \mid a \\
\textit{Clos}_{\texttt{CM}} & c & ::= & d \mid \lambda c \\
\textit{Subst}_{\texttt{CM}} & \sigma & ::= & \rho^m \\
\textit{Env}_{\texttt{CM}} & \rho & ::= & \mathsf{id} \mid c\cdot\sigma
\end{array}$$

Substitutions and environments are defined like for the `CM`-machine but with the above more general notion of closures. The set of values (i.e., terms without redexes) is exactly that of β-normal forms.

Notion of reduction If the configurations of the machine utilizing two stacks (Section 10.3) represent terms in context, then the contraction rules must be contained in the transitions of the machine. Specifically, when the left-hand and right-hand sides of a transition do not

[2] An atomic form, a, is a β-normal form that is not an abstraction (page 17).

represent the same term when plugged into their contexts, then that transition implements one of the contraction rules. The other transitions, where the left-hand and right-hand sides represent the same term when plugged into their contexts, implement the compatibility rules of the reduction semantics.

The following six rules constitute the possible ways to perform reductions in the term language.

$$
\begin{array}{rrcl}
\textit{Reindex}: & i[\mathsf{id}^m] & \rightarrow & i + m \\
\textit{Subst}: & 1[(t[\rho^m] \cdot \sigma)^{m'}] & \rightarrow & t[\rho^{m+m'}] \\
\textit{Subst}': & (i+1)[(c \cdot \rho^m)^{m'}] & \rightarrow & i[\rho^{m+m'}] \\
\textit{App}: & (t\ t')[\sigma] & \rightarrow & t[\sigma]\ t'[\sigma] \\
\textit{Beta}: & (\lambda t)[\sigma]\ c & \rightarrow & t[(c \cdot \sigma)^0] \\
\textit{Abs}: & (\lambda t)[\rho^m] & \rightarrow & \lambda t[(1[\mathsf{id}^0] \cdot \rho^{m+1})^0]
\end{array}
$$

We define $\textit{Weak} = \textit{Reindex} \cup \textit{Subst} \cup \textit{Subst}' \cup \textit{App} \cup \textit{Beta}$, and $\textit{Strong} = \textit{Weak} \cup \textit{Abs}$.

Without the rule *Abs* all m's would carry no information and they could be discarded. The rules *Subst*, *Subst′*, *App*, and *Beta* would then constitute the contraction rules of Biernacka and Danvy's $\lambda\hat{\rho}$-calculus (Section 5.2.1).[3] This calculus, paired with standard reduction strategies, e.g., normal order and (left-to-right) applicative order, syntactically corresponds to well-known machines: the Krivine machine [13, 15] and the CEK machine [32], respectively. In Section 5.2 we showed how to derive the Krivine machine extended to cope with basic constants. In Section 6.4 we outlined how to derive the CEK machine extended to cope with state variables.

In the next section, we will show that with the rules *Reindex* and *Abs*, this calculus (considering a specific strategy) syntactically corresponds to the abstract machine in Section 10.3.

Reduction contexts The reduction contexts are (when represented inside-out) isomorphic to the stacks of the abstract machine in Section 10.3:

$$
\begin{array}{llcl}
\textit{Cont} & C & ::= & [\,]\ \mid\ C[[\,]\ c] \\
\textit{MetaCont} & M & ::= & [\,]\ \mid\ M[C[a\ [\,]]]\ \mid\ M[\lambda[\,]]
\end{array}
$$

The contexts, C, are the reduction contexts of normal-order reduction to weak-head normal form. These contexts can only be plugged with an atom $a \in \textit{Atom}$ (i.e., not with an abstraction) according to the term syntax. The contraction rule *Abs* is the only rule that produces non-atom terms, so it cannot be applied in such a context. The meta-contexts, M, are exactly the contexts where the rule *Abs* can be applied: at the top of a term, or else in the operand position of an application of an atomic form that is in a normal-order weak-head reduction context in a meta-context, or else in the body of an abstraction in a meta-context.

A one-step reduction function The one-step reduction relation is defined according to the reduction contexts and contraction rules. We let $M\#C$ denote $M[C]$, where the hole of the meta-context M has been filled with the context C; the result is a term with a hole. Plugging this hole with the closure c is denoted by $M\#C[c]$.

We are now in position to define one-step reduction as the following relation on non-value terms:

$$
\begin{array}{rcll}
M\#C[c] & \mapsto & M\#C[c'], & \text{if } (c,\ c') \in \textit{Weak} \\
M\#[c] & \mapsto & M\#[c'], & \text{if } (c,\ c') \in \textit{Abs}
\end{array}
$$

[3]Rules *Subst* and *Subst′* together constitute rule ι of the $\lambda\hat{\rho}$-calculus.

We will see that the relation $\mapsto$ is a function. The following property is needed to prove that it is well-defined:

Proposition 5
A closure c *is either a* β*-normal form or it can be uniquely decomposed into a context* $M\#C$ *and a potential redex* p_a *such that* $c = M\#C[p_a]$ or *a context* $M\#[\,]$ *and a potential redex* p_c *such that* $c = M\#[p_c]$*, where potential redexes are defined as follows:*

$$\begin{array}{lcll} \mathit{PotRedex}_a & p_a & ::= & i[\sigma] \ \mid\ (t\ t)[\sigma] \ \mid\ (\lambda t)[\sigma]\ c \\ \mathit{PotRedex}_c & p_c & ::= & (\lambda t)[\sigma] \\ \mathit{PotRedex} & & = & \mathit{PotRedex}_a \cup \mathit{PotRedex}_c \end{array}$$

The unique decomposition of a non-value closure into a context and a potential redex is ensured by Proposition 5. As a function, $\mapsto$ maps any closure to at most one other closure and is hence well-defined. It is partial because it is not defined on β-normal forms, since they contain no potential redexes, and also because the potential redex $1[(c \cdot \sigma)^m]$ is only an actual redex when the closure c is of the form $t[\sigma']$.

Normalization of λ-terms The normalization of a de Bruijn-indexed λ-term t is defined by iterating the one-step reduction function until (possibly) a normal form is obtained, starting with the closure composed of t paired with the unadjusted identity environment id^0. In other words, the normalization of t is defined if the reflexive transitive closure of the one-step reduction function $\mapsto^*$ relates this initial closure with a β-normal form:

$$\texttt{normalize}\, t = n \quad \text{iff} \quad t[\mathsf{id}^0] \mapsto^* n.$$

The function `normalize` is well-defined because $\mapsto$ is a partial function not defined on β-normal forms.

10.4.2 Derivation of a corresponding abstract machine

The one-step reduction function is implicitly defined in terms of the operations of *decomposing* a term into a context and potential redex, *contracting* the redex if possible, and *plugging* the contractum back into the context. We give an explicit definition of these three operations:

Decomposing Directed by the grammar of reduction contexts and potential redexes, we explicitly define a function `decompose` performing the unique decomposition of Proposition 5 given a closure. We let all values (i.e., β-normal forms) be fixed points:

$$
\begin{array}{rcl}
\mathit{Decomp} & = & \mathit{PotRedex} \times \mathit{Cont} \times \mathit{MetaCont} \\
\\
\texttt{decompose} & : & \mathit{Clos}_{\texttt{CM}} \rightarrow \mathit{NForm}_{\mathit{deB}} + \mathit{Decomp} \\
\texttt{decompose}\ c & = & \texttt{decompose}'\ (c,\ [\,],\ [\,]) \\
\\
\texttt{decompose}' & : & \mathit{Clos}_{\texttt{CM}} \times \mathit{Cont} \times \mathit{MetaCont} \rightarrow \mathit{NForm}_{\mathit{deB}} + \mathit{Decomp} \\
\texttt{decompose}'\ (i[\sigma],\ C,\ M) & = & (i[\sigma],\ C,\ M) \\
\texttt{decompose}'\ ((\lambda t)[\sigma],\ C,\ M) & = & \texttt{aux}_C\ (C,\ (\lambda t)[\sigma],\ M) \\
\texttt{decompose}'\ ((t\ t')[\sigma],\ C,\ M) & = & ((t\ t')[\sigma],\ C,\ M) \\
\texttt{decompose}'\ (d\ c,\ C,\ M) & = & \texttt{decompose}'\ (d,\ C[[\,]\ c],\ M) \\
\texttt{decompose}'\ (a,\ C,\ M) & = & \texttt{aux}_C\ (C,\ a,\ M) \\
\texttt{decompose}'\ (\lambda c,\ [\,],\ M) & = & \texttt{decompose}'\ (c,\ [\,],\ M[\lambda[\,]]) \\
\\
\texttt{aux}_C & : & \mathit{Cont} \times (\mathit{PotRedex}_c + \mathit{ANForm}_{\mathit{deB}}) \times \mathit{MetaCont} \\
& & \rightarrow \mathit{NForm}_{\mathit{deB}} + \mathit{Decomp} \\
\texttt{aux}_C\ ([\,],\ a,\ M) & = & \texttt{aux}_M\ (M,\ a) \\
\texttt{aux}_C\ ([\,],\ (\lambda t)[\sigma],\ M) & = & ((\lambda t)[\sigma],\ [\,],\ M) \\
\texttt{aux}_C\ (C[[\,]\ c],\ a,\ M) & = & \texttt{decompose}'\ (c,\ [\,],\ M[C[a\ [\,]]]) \\
\texttt{aux}_C\ (C[[\,]\ c],\ (\lambda t)[\sigma],\ M) & = & ((\lambda t)[\sigma]\ c,\ C,\ M) \\
\\
\texttt{aux}_M & : & \mathit{MetaCont} \times \mathit{NForm}_{\mathit{deB}} \rightarrow \mathit{NForm}_{\mathit{deB}} + \mathit{Decomp} \\
\texttt{aux}_M\ ([\,],\ n) & = & n \\
\texttt{aux}_M\ (M[\lambda[\,]],\ n) & = & \texttt{aux}_M\ (M,\ \lambda n) \\
\texttt{aux}_M\ (M[C[a\ [\,]]],\ n) & = & \texttt{aux}_C\ (C,\ a\ n,\ M)
\end{array}
$$

Decomposition is a total function here because all closures not in β-normal form are mapped to a potential redex and the corresponding context and meta-context, and all β-normal forms are fixed points.

Contracting We define the function `contract` performing the contraction of an actual redex as follows:

$$
\begin{array}{rcl}
\texttt{contract} & : & \mathit{PotRedex} \rightarrow \mathit{Clos}_{\texttt{CM}} \\
\texttt{contract}\ (i[\mathsf{id}^m]) & = & i + m \\
\texttt{contract}\ (1[(t[\rho^m] \cdot \sigma)^{m'}]) & = & t[\rho^{m+m'}] \\
\texttt{contract}\ ((i+1)[(c \cdot \rho^m)^{m'}]) & = & i[\rho^{m+m'}] \\
\texttt{contract}\ ((\lambda t)[\sigma]\ c) & = & t[(c \cdot \sigma)^0] \\
\texttt{contract}\ ((t\ t')[\sigma]) & = & t[\sigma]\ t'[\sigma] \\
\texttt{contract}\ ((\lambda t)[\rho^m]) & = & \lambda t[(1[\mathsf{id}^0] \cdot \rho^{m+1})^0]
\end{array}
$$

This function is partial on potential redexes because the second contraction rule does not match all kind of closures in the substitution.

Plugging We also define a function `plug` to plug a closure into a context. This function is total and is defined by structural induction as follows:

$$\begin{array}{rcl}
\mathtt{plug} & : & \mathit{Clos}_{\mathtt{CM}} \times \mathit{Cont} \times \mathit{MetaCont} \to \mathit{Clos}_{\mathtt{CM}} \\
\mathtt{plug}\,(c,\ [\,],\ M) & = & \mathtt{plug}_M\,(c,\ M) \\
\mathtt{plug}\,(d,\ C[[\,]\ c],\ M) & = & \mathtt{plug}\,(d\ c,\ C,\ M)
\end{array}$$

$$\begin{array}{rcl}
\mathtt{plug}_M & : & \mathit{Clos}_{\mathtt{CM}} \times \mathit{MetaCont} \to \mathit{Clos}_{\mathtt{CM}} \\
\mathtt{plug}_M\,(c,\ [\,]) & = & c \\
\mathtt{plug}_M\,(c,\ M[\lambda[\,]]) & = & \mathtt{plug}_M\,(\lambda c,\ M) \\
\mathtt{plug}_M\,(c,\ M[C[a\ [\,]]]) & = & \mathtt{plug}\,(a\ c,\ C,\ M)
\end{array}$$

Normalization defined via more explicit one-step reduction Finally, we reformulate normalization from Section 10.4.1 based on the above concrete definitions:

$$\begin{array}{rcl}
\mathtt{iterate} & : & \mathit{NForm}_{deB} + \mathit{Decomp} \to \mathit{NForm}_{deB} \\
\mathtt{iterate}\,n & = & n \\
\mathtt{iterate}\,(p,\ C,\ M) & = & \mathtt{iterate}\,(\mathtt{decompose}\,(\mathtt{plug}\,(\mathtt{contract}\,p,\ C,\ M)))
\end{array}$$

$$\begin{array}{rcl}
\mathtt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\
\mathtt{normalize}\,t & = & \mathtt{iterate}\,(\mathtt{decompose}\,(t[\mathsf{id}^0]))
\end{array}$$

Introducing a refocus function Danvy and Nielsen [28] observed that when one-step reduction is iterated, decomposition is always performed on the result of plugging. The detour to the root of the term via plugging and back down to the next redex site via decomposition can be eliminated. Danvy and Nielsen gave an algorithm to construct a *refocus* function that navigated in a term directly from redex site to redex site. Furthermore, they observed that a refocused specification of reduction avoids constructing intermediate terms, which is in essence the difference between reduction semantics, which are ‘reduction-based’, and abstract machines, which are ‘reduction-free’.

Refocusing the above normalization function inlining the `contract` function, and fusing `refocus` with the function iterating it yields an abstract machine. We have introduced an auxiliary function `lookup` to match the three cases for the decomposition $(i[\sigma],\ C,\ M)$

corresponding to the *Reindex*, *Subst*, and *Subst'* contraction rules:

$$
\begin{array}{rcl}
\texttt{refocus} & : & \mathit{Clos}_{\mathtt{CM}} \times \mathit{Cont} \times \mathit{MetaCont} \to \mathit{NForm}_{deB} \\
\texttt{refocus}\ (i[\sigma],\ C,\ M) & = & \texttt{lookup}\ (i[\sigma],\ C,\ M) \\
\texttt{refocus}\ ((\lambda t)[\sigma],\ C,\ M) & = & \texttt{aux}_C\ (C,\ (\lambda t)[\sigma],\ M) \\
\texttt{refocus}\ ((t\ t')[\sigma],\ C,\ M) & = & \texttt{refocus}\ (t[\sigma]\ t'[\sigma],\ C,\ M) \\
\texttt{refocus}\ (d\ c,\ C,\ M) & = & \texttt{refocus}\ (d,\ C[[\,]\ c],\ M) \\
\texttt{refocus}\ (a,\ C,\ M) & = & \texttt{aux}_C\ (C,\ a,\ M) \\
\texttt{refocus}\ (\lambda c,\ [\,],\ M) & = & \texttt{refocus}\ (c,\ [\,],\ M[\lambda[\,]])
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{lookup} & : & \mathit{Clos}_{\mathtt{CM}} \times \mathit{Cont} \times \mathit{MetaCont} \to \mathit{NForm}_{deB} \\
\texttt{lookup}\ (i[\mathsf{id}^m],\ C,\ M) & = & \texttt{refocus}\ (i+m,\ C,\ M) \\
\texttt{lookup}\ (1[(t[\rho^m]\cdot\sigma)^{m'}],\ C,\ M) & = & \texttt{refocus}\ (t[\rho^{m+m'}],\ C,\ M) \\
\texttt{lookup}\ ((i+1)[(c\cdot\rho^m)^{m'}],\ C,\ M) & = & \texttt{refocus}\ (i[\rho^{m+m'}],\ C,\ M)
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{aux}_C & : & \mathit{Cont} \times (\mathit{PotRedex}_c \times \mathit{ANForm}_{deB}) \times \mathit{MetaCont} \to \mathit{NForm}_{deB} \\
\texttt{aux}_C\ ([\,],\ a,\ M) & = & \texttt{aux}_M\ (M,\ a) \\
\texttt{aux}_C\ ([\,],\ (\lambda t)[\rho^m],\ M) & = & \texttt{refocus}\ (\lambda t[(1[\mathsf{id}^0]\cdot\rho^{m+1})^0],\ [\,],\ M) \\
\texttt{aux}_C\ (C[[\,]\ c],\ a,\ M) & = & \texttt{refocus}\ (c,\ [\,],\ M[C[a\ [\,]]]) \\
\texttt{aux}_C\ (C[[\,]\ c],\ (\lambda t)[\sigma],\ M) & = & \texttt{refocus}\ (t[(c\cdot\sigma)^0],\ C,\ M)
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{aux}_M & : & \mathit{MetaCont} \times \mathit{NForm}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{aux}_M\ ([\,],\ n) & = & n \\
\texttt{aux}_M\ (M[\lambda[\,]],\ n) & = & \texttt{aux}_M\ (M,\ \lambda n) \\
\texttt{aux}_M\ (M[C[a\ [\,]]],\ n) & = & \texttt{aux}_C\ (C,\ a\ n,\ M)
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{normalize}\ t & = & \texttt{refocus}\ (t[\mathsf{id}^0],\ [\,],\ [\,])
\end{array}
$$

This specification defines an abstract machine using four transition functions. The initial transition maps a term t to $(t[\mathsf{id}^0],\ [\,],\ [\,])$, and the final transition extracts the β-normal form n from $([\,],\ n)$. This machine uses two stacks to represent the context. Also this machine cannot get stuck.

Short-circuiting transitions Looking, e.g., at the third clause of `refocus`, where the first component takes the form $(t\ t')[\sigma]$, we observe that an application to `refocus` can be short-circuited via:

$$
\begin{array}{rcl}
\texttt{refocus}\ ((t\ t')[\sigma],\ C,\ M) & = & \texttt{refocus}\ (t[\sigma]\ t'[\sigma],\ C,\ M) \\
& = & \texttt{refocus}\ (t[\sigma],\ C[[\,]\ t'[\sigma]],\ M)
\end{array}
$$

The application of refocus that has just been short-circuited was the only place where closures of the form $c\ c$ was constructed. Thus, the fourth clause of the definition is not needed and can be removed. We also see that this clause is the only place where the contexts are extended. This means that the closure in the context will always take the form of a term paired with a substitution, demonstrating that the machine cannot get stuck.

In a similar manner, the work of the last two clauses of `refocus` can also be short-circuited and because afterward the corresponding constructions are never built, the two clauses can be elided. Likewise, we can short-circuit the application of `refocus` in the last clause of `lookup`. The abstract machine after simplifications reads as follows:

$$
\begin{array}{rcl}
\texttt{refocus} & : & \mathit{Clos}_{\texttt{CM}} \times \mathit{Cont} \times \mathit{MetaCont} \rightarrow \mathit{NForm}_{deB} \\
\texttt{refocus}\ (i[\sigma],\ C,\ M) & = & \texttt{lookup}\ (i[\sigma],\ C,\ M) \\
\texttt{refocus}\ ((\lambda t)[\sigma],\ C,\ M) & = & \texttt{aux}_C\ (C,\ (\lambda t)[\sigma],\ M) \\
\texttt{refocus}\ ((t\ t')[\sigma],\ C,\ M) & = & \texttt{refocus}\ (t[\sigma],\ C[[\,]\ t'[\sigma]],\ M) \\
\\
\texttt{lookup} & : & \mathit{Clos}_{\texttt{CM}} \times \mathit{Cont} \times \mathit{MetaCont} \rightarrow \mathit{NForm}_{deB} \\
\texttt{lookup}\ (i[\mathsf{id}^m],\ C,\ M) & = & \texttt{aux}_C\ (C,\ i+m,\ M) \\
\texttt{lookup}\ (1[(t[\rho^m]\cdot\sigma)^{m'}],\ C,\ M) & = & \texttt{refocus}\ (t[\rho^{m+m'}],\ C,\ M) \\
\texttt{lookup}\ ((i+1)[(c\cdot\rho^m)^{m'}],\ C,\ M) & = & \texttt{lookup}\ (i[\rho^{m+m'}],\ C,\ M) \\
\\
\texttt{aux}_C & : & \mathit{Cont} \times (\mathit{PotRedex}_c \times \mathit{ANForm}_{deB}) \times \mathit{MetaCont} \rightarrow \mathit{NForm}_{deB} \\
\texttt{aux}_C\ ([\,],\ a,\ M) & = & \texttt{aux}_M\ (M,\ a) \\
\texttt{aux}_C\ ([\,],\ (\lambda t)[\rho^m],\ M) & = & \texttt{refocus}\ (t[(1[\mathsf{id}^0]\cdot\rho^{m+1})^0],\ [\,],\ M[\lambda[\,]]) \\
\texttt{aux}_C\ (C[[\,]\ c],\ a,\ M) & = & \texttt{refocus}\ (c,\ [\,],\ M[C[a\ [\,]]]) \\
\texttt{aux}_C\ (C[[\,]\ c],\ (\lambda t)[\sigma],\ M) & = & \texttt{refocus}\ (t[(c\cdot\sigma)^0],\ C,\ M) \\
\\
\texttt{aux}_M & : & \mathit{MetaCont} \times \mathit{NForm}_{deB} \rightarrow \mathit{NForm}_{deB} \\
\texttt{aux}_M\ ([\,],\ n) & = & n \\
\texttt{aux}_M\ (M[\lambda[\,]],\ n) & = & \texttt{aux}_M\ (M,\ \lambda n) \\
\texttt{aux}_M\ (M[C[a\ [\,]]],\ n) & = & \texttt{aux}_C\ (C,\ a\ n,\ M) \\
\\
\texttt{normalize} & : & \mathit{Term}_{deB} \rightarrow \mathit{NForm}_{deB} \\
\texttt{normalize}\ t & = & \texttt{refocus}\ (t[\mathsf{id}^0],\ [\,],\ [\,])
\end{array}
$$

Flattening closures At this point only one of the constructions of closures is used — the construction with a de Bruijn indexed λ-term paired with a substitution. By splitting such a closure into a pair of a term and a substitution, `refocus` and `lookup` can operate directly on quadruples composed of a λ-term, a substitution, a context, and a meta-context. The first component of `lookup` is simplified to just being an index.

Conclusion A direct comparison with the two-stack abstract machine from Section 10.3 is possible: `refocus` coincides with `eval`, `lookup` coincides with `lookup`, $\texttt{aux}_C$ coincides with $\texttt{apply}_C$, and $\texttt{aux}_M$ coincides with $\texttt{apply}_M$. In other words, the final abstract machine is the abstract machine with two stacks from Section 10.3. The only difference is the representations of the two stacks.

10.5 From a continuation-based normalization function to a normalization function in direct style

The continuation-based normalization function from Section 10.2.3 can be transformed to direct style using the control delimiter *reset* (noted $\langle\cdot\rangle$ in the machine below) to account for the three occurrences of the initialization of the continuation [23]. The types *Fun* and *Thunk* change to:

$$
\begin{array}{rcl}
\mathit{Thunk} & = & \mathit{Index} \rightarrow \mathit{Fun} + \mathit{ANForm}_{deB} \\
\mathit{Fun} & = & \mathit{Index} \times \mathit{Cont} \rightarrow \mathit{Fun} + \mathit{ANForm}_{deB}
\end{array}
$$

The resulting direct-style normalization function has no control effects. Therefore all three occurrences of the delimiter *reset* can be erased [23]. The normalization function with the delimiters using a call-by-value metalanguage reads as follows (The occurrences of *reset* are shaded):

$$
\begin{array}{rcl}
\mathtt{eval} & : & \mathit{Term}_{deB} \times \mathit{Subst}_{\mathtt{CM}} \to \mathit{Fun} + \mathit{ANForm}_{deB} \\
\mathtt{eval}\,(i,\ \sigma) & = & \mathtt{lookup}\,(i,\ \sigma) \\
\mathtt{eval}\,(\lambda t,\ \rho^{m}) & = & \lambda(c,\ m').\mathtt{eval}\,(t,\ (c \cdot \rho^{m+m'})^{0}) \\
\mathtt{eval}\,(t\ t',\ \rho^{m}) & = & \mathit{case}\ \mathtt{eval}\,(t,\ \rho^{m}) \\
 & & \quad \mathit{of}\ f \Rightarrow f\,(\lambda m'.\mathtt{eval}\,(t',\ \rho^{m+m'}),\ 0) \\
 & & \quad\ \mid a \Rightarrow a\ \langle\downarrow (\mathtt{eval}\,(t',\ \rho^{m}))\rangle \\
\\
\mathtt{lookup} & : & \mathit{Index} \times \mathit{Subst}_{\mathtt{CM}} \to \mathit{Fun} + \mathit{ANForm}_{deB} \\
\mathtt{lookup}\,(i,\ \mathsf{id}^{m}) & = & i + m \\
\mathtt{lookup}\,(1,\ (c \cdot \sigma)^{m'}) & = & c\ m' \\
\mathtt{lookup}\,(i+1,\ (c \cdot \rho^{m})^{m'}) & = & \mathtt{lookup}\,(i,\ \rho^{m+m'}) \\
\\
\downarrow & : & \mathit{Fun} + \mathit{ANForm}_{deB} \to \mathit{NForm}_{deB} \\
\downarrow f & = & \lambda\ \langle\downarrow (f\,(\lambda m.m+1,\ 1))\rangle \\
\downarrow a & = & a \\
\\
\mathtt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\
\mathtt{normalize}\,t & = & \langle\downarrow (\mathtt{eval}\,(t,\ \mathsf{id}^{0}))\rangle
\end{array}
$$

10.6 From a normalization function in direct style to an abstract machine with one stack

Through the functional correspondence, we closure convert, CPS-transform and then defunctionalize the normalization function from Section 10.5 *without* the control delimiters. The result is an abstract machine utilizing one stack for the data-structure continuation:

$$
\begin{array}{lccl}
\mathit{MCont} & M & ::= & \bullet \mid a \cdot C \mid \Lambda \cdot M \\
\mathit{Cont} & C & ::= & M \mid c \cdot C
\end{array}
$$

This abstract machine is thus similar to the abstract machine corresponding to a reduction semantics in the $\lambda\hat{\hat{s}}$-calculus (Section 8.2.2) and to the abstract machine obtained from Lescanne's normalizer after merging the data-structure continuation into the stack (Section 9.2):

$$
\begin{array}{rcl}
\texttt{eval} & : & \mathit{Term}_{deB} \times \mathit{Subst}_{\texttt{CM}} \times \mathit{Cont} \to \mathit{NForm}_{deB} \\
\texttt{eval}\,(i,\ \sigma,\ C) & = & \texttt{lookup}\,(i,\ \sigma,\ C) \\
\texttt{eval}\,(\lambda t,\ \sigma,\ C) & = & \texttt{apply}_C\,(C,\ (\lambda t)[\sigma]) \\
\texttt{eval}\,(t\,t',\ \sigma,\ C) & = & \texttt{eval}\,(t,\ \sigma,\ t'[\sigma] \cdot C) \\
\\
\texttt{lookup} & : & \mathit{Term}_{deB} \times \mathit{Subst}_{\texttt{CM}} \times \mathit{Cont} \to \mathit{NForm}_{deB} \\
\texttt{lookup}\,(i,\ \mathsf{id}^m,\ C) & = & \texttt{apply}_C\,(C,\ i+m) \\
\texttt{lookup}\,(1,\ (t[\rho^m] \cdot \sigma)^{m'},\ C) & = & \texttt{eval}\,(t,\ \rho^{m+m'},\ C) \\
\texttt{lookup}\,(i+1,\ (c \cdot \rho^m)^{m'},\ C) & = & \texttt{lookup}\,(i,\ \rho^{m+m'},\ C) \\
\\
\texttt{apply}_C & : & \mathit{Cont} \times (\mathit{PotRedex}_c \times \mathit{ANForm}_{deB}) \to \mathit{NForm}_{deB} \\
\texttt{apply}_C\,(M,\ a) & = & \texttt{apply}_M\,(M,\ a) \\
\texttt{apply}_C\,(M,\ (\lambda t)[\rho^m]) & = & \texttt{eval}\,(t,\ (1[\mathsf{id}^0] \cdot \rho^{m+1})^0,\ \Lambda \cdot M) \\
\texttt{apply}_C\,(t[\sigma] \cdot C,\ a) & = & \texttt{eval}\,(t,\ \sigma,\ a \cdot C) \\
\texttt{apply}_C\,(c \cdot C,\ (\lambda t)[\sigma]) & = & \texttt{eval}\,(t,\ (c \cdot \sigma)^0,\ C) \\
\\
\texttt{apply}_M & : & \mathit{MCont} \times \mathit{NForm}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{apply}_M\,(\bullet,\ n) & = & n \\
\texttt{apply}_M\,(\Lambda \cdot M,\ n) & = & \texttt{apply}_M\,(M,\ \lambda n) \\
\texttt{apply}_M\,(a \cdot C,\ n) & = & \texttt{apply}_C\,(C,\ a\,n) \\
\\
\texttt{normalize} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{normalize}\,t & = & \texttt{eval}\,(t,\ \mathsf{id}^0,\ \bullet)
\end{array}
$$

This machine normalizes a term t by starting in the configuration $(t,\ \mathsf{id}^0,\ \bullet)$. It halts with a β-normal form n if it reaches a configuration $(\bullet,\ n)$.

10.7 From a reduction semantics with one layer of contexts to an abstract machine with one stack

In this section, we define a reduction semantics with one-layer contexts and outline how the abstract machine with one stack from Section 10.6 can be mechanically derived. Again the derivation hinges on the transformations of the syntactic correspondence.

A one-layer reduction semantics Most of the definitions are the same as in the two-layer reduction semantics of Section 10.4.1. The syntactic units are again closures and the term language is unchanged from the one defined in Section 10.4.1. Values are still the β-normal forms. The contraction rules are also unchanged from the ones defined in Section 10.4.1.

The sole difference from the two-layer reduction semantics is in the grammar of reduction contexts. The reduction semantics uses the one-layer (but stratified) grammar of contexts seen several times in the previous chapters:

$$
\begin{array}{llcl}
\mathit{AContext} & A[\,] & ::= & [\,] \mid C[a\,[\,]] \mid A[\lambda[\,]] \\
\mathit{CContext} & C[\,] & ::= & A[\,] \mid C[[\,]\,c]
\end{array}
$$

Plugging a closure c into a context $A[\,]$ (and context $C[\,]$) is denoted $A[c]$ (and $C[c]$).

One-step reduction is now defined as the following relation on non-value closures:

$$\begin{array}{lcll} C[c] & \mapsto & C[c'], & \text{if } (c,\, c') \in \mathit{Weak} \\ A[c] & \mapsto & A[c'], & \text{if } (c,\, c') \in \mathit{Abs} \end{array}$$

Again, $\mapsto$ is a function. The following unique decomposition property and the contraction rules prove the well-definedness of $\mapsto$ (where potential redexes are defined as in Section 10.4.1):

Proposition 6
A closure c *is either a* β*-normal form or it can be decomposed uniquely into context* $C[\,]$ *and potential redex* p_a *such that* $c = C[p_a]$ or *context* $A[\,]$ *and potential redex* p_c *such that* $c = A[p_c]$.

Derivation of an abstract machine Through the syntactic correspondence, we construct a refocused specification of the one-step reduction function and fuse it with a function iterating it. The result is an abstract machine. After short-circuiting transitions and eliminating unreachable configurations, we obtain the one-stack abstract machine from Section 10.6.

10.8 Summary

Curien's normalizer is defined with a driver loop initializing the weak `CM`-machine. As seen in Section 9.1, Lescanne's normalizer has the same structure. We simplified the abstract machine corresponding to Lescanne's normalizer by merging the data-structure continuation into the existing stack. We identify this simplification as corresponding to erasing moot control delimiters in the corresponding direct-style normalization function.

We showed that both the functional correspondence between abstract machines and evaluation functions and the syntactic correspondence between reduction semantics and abstract machines can be applied to strong normalization. In so doing, we established connections between an array of semantic artifacts: Curien's normalizer, a continuation-based normalization function, a normalization function in direct style, a traditional-style abstract machine, and a reduction semantics. A small variation on the direct-style normalization function leads to alternate but equivalent abstract machines and reduction semantics.

The normalization function corresponding to Curien's normalizer is compositional. The first abstract machine (Section 10.3) uses two stacks, one corresponding to the original stack of Curien's machine and one corresponding to the non-tail calls introduced in the driver loop. These two stacks are a consequence of our mechanical development of the abstract machine. The abstract machines corresponding to known normalization functions would normally have the one-stack architecture of the second abstract machine (Section 10.6). We do note, however, that this two-stack architecture is also found in Landin's SECD machine (Section 4.5.2).

This chapter is based on joint work with Olivier Danvy and Kevin Millikin, but mistakes are mine.

Chapter 11

Strong normalization starting from Crégut's KN-machine

In the previous chapter our starting point was Curien's normalizer, implementing normalization of de Bruijn-indexed λ-terms to β-normal forms. The normalizer has a driver loop, which repeatedly instantiates the weak CM-machine.

Crégut's first specification for strong normalization also consists of a driver loop and a weak machine, with the driver loop repeatedly instantiating the weak machine [13]. Crégut has recently superseded this normalizer with his definition of the KN-*machine* [14]. This machine takes the form of an abstract machine (Section 4.5.2).

Roadmap In this chapter our starting point is Crégut's KN-machine (Section 11.1). In previous chapters we have seen the value of having abstract machines in disentangled form. We hence disentangle the KN-machine in Section 11.2. Again, the disentangled version is in the image of defunctionalization. We present the higher-order direct-style compositional normalization function functionally corresponding to this version of the KN-machine (Section 11.3). We present the reduction semantics syntactically corresponding to that abstract machine in Section 11.4.

11.1 Crégut's definitional specification of the KN-machine

Crégut's machine performs strong normalization of closed de Bruijn-indexed λ-terms. In the paper Crégut lets de Bruijn indices start at 0. For uniformity with the rest of this text, we 'shift' the de Bruijn indices such that they start at 1 and we adjust the transition rules accordingly.

Nonterminal configurations of the KN-machine are composed of four parts: (1) a 'term' element u, (2) an environment e, (3) a stack s, and (4) a global de Bruijn level m holding the current abstraction-depth:[1]

$$\begin{array}{llcl}
\textit{Level} & m & ::= & \{0, 1, 2, \ldots\} \\
P & p & ::= & \{t, m\} \\
U & u & ::= & t \mid \mathtt{V}\ m \mid p \\
 & c & ::= & (u, e) \\
\textit{Env}_{\mathtt{KN}} & e & ::= & \bullet \mid c \cdot e \\
\textit{Stack}_{\mathtt{KN}} & s & ::= & \bullet \mid c \cdot s \mid \Lambda \cdot s \mid p \cdot s \\
\textit{Configuration}_{\mathtt{KN}} & & := & U \times \textit{Env}_{\mathtt{KN}} \times \textit{Stack}_{\mathtt{KN}} \times \textit{Level}
\end{array}$$

Here $t \in \textit{Term}_{deB}$ as defined in Section 1.6. The start configuration of the KN-machine is composed of the argument term for normalization t, an empty environment, an empty stack, and 0 as the current abstraction-depth: $(t,\ \bullet,\ \bullet,\ 0)$. The machine always stops with a use of the 'unload transition' when the term-part is a λ-term with an associated de Bruijn level and the stack is empty:[2]

$$\mathtt{KN}\,(\{t, m'\},\ e,\ \bullet,\ m) \quad \to \quad t$$

Crégut's definition of the KN-machine is transliterated into our notation:

$$\begin{array}{rcll}
\mathtt{normalize}_{\mathtt{KN}} & : & \textit{Term}_{deB} \to \textit{Term}_{deB} & \\
\mathtt{normalize}_{\mathtt{KN}}\, t & = & \mathtt{KN}\,(t,\ \bullet,\ \bullet,\ 0) & \\
 & & & \\
\mathtt{KN} & : & \textit{Configuration}_{\mathtt{KN}} \to \textit{Term}_{deB} & \\
\mathtt{KN}\,(1,\ (u, e') \cdot e,\ s,\ m) & = & \mathtt{KN}\,(u,\ e',\ s,\ m) & \\
\mathtt{KN}\,(i+1,\ c \cdot e,\ s,\ m) & = & \mathtt{KN}\,(i,\ e,\ s,\ m) & \\
\mathtt{KN}\,(t\,t',\ e,\ s,\ m) & = & \mathtt{KN}\,(t,\ e,\ (t', e) \cdot s,\ m) & \\
\mathtt{KN}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{KN}\,(t,\ c \cdot e,\ s',\ m), & \text{if } s = c \cdot s' \\
\mathtt{KN}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{KN}\,(t,\ (\mathtt{V}\ m, \bullet) \cdot e,\ \Lambda \cdot s,\ m+1), & \text{if } s \neq c \cdot s' \\
\mathtt{KN}\,(\mathtt{V}\ m',\ e,\ s,\ m) & = & \mathtt{KN}\,(\{m - m', m\},\ e,\ s,\ m) & \\
\mathtt{KN}\,(\{t, m'\},\ e,\ \bullet,\ m) & = & t & \\
\mathtt{KN}\,(\{t, m'\},\ e,\ (t', e') \cdot s,\ m) & = & \mathtt{KN}\,(t',\ e',\ \{t, m'\} \cdot s,\ m') & \\
\mathtt{KN}\,(\{t, m'\},\ e,\ \{t', m''\} \cdot s,\ m) & = & \mathtt{KN}\,(\{t'\,t, m''\},\ e,\ s,\ m) & \\
\mathtt{KN}\,(\{t, m'\},\ e,\ \Lambda \cdot s,\ m) & = & \mathtt{KN}\,(\{\lambda t, m'\},\ e,\ s,\ m) &
\end{array}$$

Crégut proves the correctness of the KN-machine, i.e., that the KN-machine actually computes normal forms when they exist:[3]

Theorem 6 (Crégut)
Let t *a be closed term and* t' *be a* β*-normal form.*
Then $t =_\beta t' \iff \mathtt{normalize}_{\mathtt{KN}}\, t = t'$

11.1.1 Normalization method

When the current term element is an index i, the second rule removes the first $i - 1$ closures in the environment and the first rule then selects the i-th closure. Because Crégut only considers closed terms the environment always contains enough closures.

[1]The set of de Bruijn levels *Level* contain 0 as opposed to the set of de Bruijn indices *Index* used in λ-terms.

[2]For now, we differ from using the metavariable n ranging over β-normal forms to follow Crégut's definition. We thus also define the co-domain of the normalization function to be $\textit{Term}_{deB}$ instead of $\textit{NForm}_{deB}$.

[3]Lescanne and Curien do not present corresponding theorems for their machines.

Considering an application tt' the machine continues normalization of the left sub-term t with a new closure (consisting of the right sub-term t' paired with the current environment) pushed on the stack.

Meeting an abstraction the top element of the stack is inspected with two different outcomes: (1) If that element is a closure c the abstraction is the left sub-term of an application, and c is moved to the environment. That is, de Bruijn index 1 is bound to that closure and the rest of the environment is 'lifted' by one. (2) If the top element of the stack is not a closure, the abstraction is not the left sub-term of an application and the body of the abstraction must be normalized with an indication Λ on the stack that the normalized body is the body of an abstraction in the residual normal form. Also the current de Bruijn level is incremented and 1 is bound to a special closure $(\mathtt{V}\ m, \bullet)$.

When such a special closure $(\mathtt{V}\ m', \bullet)$ is substituted for a de Bruijn index, that index is (as explained above) bound by an abstraction that is part of the normal form. That is, the result should be an adjusted de Bruijn index. By construction the abstraction has level m' and the current de Bruijn level is m, i.e., the resulting index is the difference $m - m'$. This residual de Bruijn index is introduced in the construction $\{m - m', m\}$.

A construction $\{t, m'\}$ indicates that a partial result t is found and its de Bruijn level is m'. The next configuration depends on the stack. If the stack is empty t is the complete normal form. If the top element of the stack is a closure (t', e'), the left sub-term of an application could not be reduced to an abstraction, and the right sub-term t' must be normalized in the application-time environment e'. $\{t, m'\}$ is pushed on the stack to indicate an application must be constructed with t as left sub-term when normalization of the right sub-term is finished. Finally, if the top element of the stack indicates a residual abstraction (Λ) the partial result t is the normalized body of an abstraction, and the partial result is λt.

A de Bruijn level m is included in $\{t, m\}$-constructions such that right sub-terms of residual applications are normalized with the correct current de Bruijn level.

11.1.2 Normalization of open terms

Because not all terms have a normal form, normalization is partial. But as the KN-machine is defined above normalization is also not defined on terms with free variables. In this text all machines so far normalize also such open terms. Only one transition rule is needed to extend the KN-machine to also normalize open terms:

$$\mathtt{KN}\,(i,\ \bullet,\ s,\ m) \quad = \quad \mathtt{KN}\,(\{i + m, m\},\ \bullet,\ s,\ m)$$

Considering an index that in the argument term corresponds to the i-th free variable, the second rule of KN subtracts the length of the environment in small steps. This length is the number of abstractions from the root of the term to the index, because the environment is always extended when normalizing the body of abstractions and nowhere else. The obtained index is therefore i. The above rule then adds the current abstraction-depth m, i.e., the number or *constructed* abstractions from the root to the index where it occurs in the residual term. In result, the de Bruijn index $i + m$ at that place in the normal form, then corresponds to the i-th free variable of the overall term. Also m is the current level of the residual term $i + m$. We continue with the above rule included.

11.2 Simplifications of the $\mathtt{KN}$-machine

Splitting the set of configurations in two When the first component of a configuration is a $\{t, m\}$-construction the level m in the fourth component is never used. This observation applies to the last four rules of $\mathtt{KN}$. The same property holds for the environment component. We split the transition rules by introducing an auxiliary function $\mathtt{aux}$ in which the level and the environment components have been discarded according to the above observation:

$$
\begin{array}{rcll}
\mathtt{normalize}_{\mathtt{KN}} & : & \mathit{Term}_{deB} \to \mathit{Term}_{deB} & \\
\mathtt{normalize}_{\mathtt{KN}}\, t & = & \mathtt{KN}\,(t,\ \bullet,\ \bullet,\ 0) & \\
\\
\mathtt{KN} & : & \mathit{Configuration}_{\mathtt{KN}} \to \mathit{Term}_{deB} & \\
\mathtt{KN}\,(i,\ \bullet,\ s,\ m) & = & \mathtt{aux}\,(s,\ \{i+m, m\}) & \\
\mathtt{KN}\,(1,\ (u, e')\cdot e,\ s,\ m) & = & \mathtt{KN}\,(u,\ e',\ s,\ m) & \\
\mathtt{KN}\,(i+1,\ c\cdot e,\ s,\ m) & = & \mathtt{KN}\,(i,\ e,\ s,\ m) & \\
\mathtt{KN}\,(t\,t',\ e,\ s,\ m) & = & \mathtt{KN}\,(t,\ e,\ (t', e)\cdot s,\ m) & \\
\mathtt{KN}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{KN}\,(t,\ c\cdot e,\ s',\ m), & \text{if } s = c\cdot s' \\
\mathtt{KN}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{KN}\,(t,\ (\mathtt{V}\,m, \bullet)\cdot e,\ \Lambda\cdot s,\ m+1), & \text{if } s \neq c\cdot s' \\
\mathtt{KN}\,(\mathtt{V}\,m',\ e,\ s,\ m) & = & \mathtt{aux}\,(s,\ \{m-m', m\}) & \\
\\
\mathtt{aux} & : & \mathit{Stack}_{\mathtt{KN}} \times P \to \mathit{Term}_{deB} & \\
\mathtt{aux}\,(\bullet,\ \{t, m\}) & = & t & \\
\mathtt{aux}\,(\{t', m'\}\cdot s,\ \{t, m\}) & = & \mathtt{aux}\,(s,\ \{t'\,t, m'\}) & \\
\mathtt{aux}\,(\Lambda\cdot s,\ \{t, m\}) & = & \mathtt{aux}\,(s,\ \{\lambda t, m\}) & \\
\mathtt{aux}\,((t', e)\cdot s,\ \{t, m\}) & = & \mathtt{KN}\,(t',\ e,\ \{t, m\}\cdot s,\ m) &
\end{array}
$$

We follow the convention and let the stack be the first argument to emphasize that $\mathtt{aux}$ dispatches on the possible stacks.

Eliminating P A simple analysis of the transformation rules gives that the first part t of a $\{t, m\}$-construction is always a normal form. Furthermore as explained in Section 11.1.1, the second part m is only included to proceed normalization with the correct de Bruijn level in the right sub-term of an application that eventually will be constructed in the normal form (if it exists). By storing that level in the closures on the stack, the level paired with normal forms in $\{t, m\}$-constructions can be discarded. It is hence implied that P is no longer needed; the set of normal forms NForm_{deB} is sufficient.

Stacks are now defined such that stack closures contain a de Bruijn index. Also according to the above analysis a non-abstraction normal form on the stack a denotes a residual application with a as left sub-term:

$$
\mathit{Stack}'_{\mathtt{KN}} \qquad s \quad ::= \quad \bullet \mid (t, e, m)\cdot s \mid \Lambda\cdot s \mid a\cdot s
$$

The adjusted version of the KN-machine reads as follows:

$$
\begin{array}{rcll}
\mathtt{normalize}_{\mathtt{KN}} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} & \\
\mathtt{normalize}_{\mathtt{KN}}\ t & = & \mathtt{KN}\,(t,\ \bullet,\ \bullet,\ 0) & \\
\\
\mathtt{KN} & : & U \times \mathit{Env}_{\mathtt{KN}} \times \mathit{Stack}'_{\mathtt{KN}} \times \mathit{Level} \to \mathit{NForm}_{deB} & \\
\mathtt{KN}\,(i,\ \bullet,\ s,\ m) & = & \mathtt{aux}\,(s,\ i+m) & \\
\mathtt{KN}\,(1,\ (u,e')\cdot e,\ s,\ m) & = & \mathtt{KN}\,(u,\ e',\ s,\ m) & \\
\mathtt{KN}\,(i+1,\ (u,e')\cdot e,\ s,\ m) & = & \mathtt{KN}\,(i,\ e,\ s,\ m) & \\
\mathtt{KN}\,(t\,t',\ e,\ s,\ m) & = & \mathtt{KN}\,(t,\ e,\ (t',e,m)\cdot s,\ m) & \\
\mathtt{KN}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{KN}\,(t,\ (t',e')\cdot e,\ s',\ m), & \text{if } s = (t',e',m')\cdot s' \\
\mathtt{KN}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{KN}\,(t,\ (\mathtt{V}\ m,\bullet)\cdot e,\ \Lambda\cdot s,\ m+1), & \text{if } s \neq (t',e',m')\cdot s' \\
\mathtt{KN}\,(\mathtt{V}\ m',\ e,\ s,\ m) & = & \mathtt{aux}\,(s,\ m-m') & \\
\\
\mathtt{aux} & : & \mathit{Stack}'_{\mathtt{KN}} \times \mathit{NForm}_{deB} \to \mathit{NForm}_{deB} & \\
\mathtt{aux}\,(\bullet,\ n) & = & n & \\
\mathtt{aux}\,(a\cdot s,\ n) & = & \mathtt{aux}\,(s,\ a\,n) & \\
\mathtt{aux}\,(\Lambda\cdot s,\ n) & = & \mathtt{aux}\,(s,\ \lambda n) & \\
\mathtt{aux}\,((t,e,m)\cdot s,\ a) & = & \mathtt{KN}\,(t,\ e,\ a\cdot s,\ m) &
\end{array}
$$

Eliminate *U* from configurations and disentangle with respect to the stack Except for the last rule, KN operates on standard λ-terms. By splitting the second rule into two and match on the first element in the environment the last rule of KN can be removed, which eliminates the need for *U*. The definition of environments is adjusted, such that only actual λ-terms are associated with an environment, because retrieving a $(\mathtt{V}\ m, e')$ from the environment never uses e'.

We move the two conditional rules of KN to aux, such that only aux dispatches on the structure of the stack. Like for the abstract machine corresponding to Lescanne's normalizer (Chapter 9.2), we define the stack in a stratified fashion to facilitate pattern matching. The revised definitions of $\mathit{Env}_{\mathtt{KN}}$ and $\mathit{Stack}'_{\mathtt{KN}}$ read:

$$
\begin{array}{lrcl}
 & v & ::= & (t,e) \mid \mathtt{V}\ m \\
\mathit{Env}'_{\mathtt{KN}} & e & ::= & \bullet \mid v\cdot e \\
\mathit{MStack}_{\mathtt{KN}} & M & ::= & \bullet \mid a\cdot s \mid \Lambda\cdot M \\
\mathit{Stack}''_{\mathtt{KN}} & s & ::= & M \mid (t,e,m)\cdot s
\end{array}
$$

In all but one call to aux the second argument is in ANForm_{deB}. All applications of aux in KN and the second clause in aux are of this kind. The exception is the third clause of aux. Here the argument takes the form λn.

The simplified KN-machine finally reads (where $\mathit{Clos}_{\mathtt{KN}} = \mathit{Term}_{deB} \times \mathit{Env}'_{\mathtt{KN}} \times \mathit{Level}$ denotes the set of closures):

$$\begin{array}{rcl}
\texttt{normalize}_{\texttt{KN}} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{normalize}_{\texttt{KN}}\, t & = & \texttt{KN}\,(t,\ \bullet,\ \bullet,\ 0) \\
\\
\texttt{KN} & : & \mathit{Term}_{deB} \times \mathit{Env}'_{\texttt{KN}} \times \mathit{Stack}''_{\texttt{KN}} \times \mathit{Level} \to \mathit{NForm}_{deB} \\
\texttt{KN}\,(i,\ \bullet,\ s,\ m) & = & \texttt{aux}\,(s,\ i+m) \\
\texttt{KN}\,(1,\ \mathsf{V}\,m' \cdot e,\ s,\ m) & = & \texttt{aux}\,(s,\ m-m') \\
\texttt{KN}\,(1,\ (t, e') \cdot e,\ s,\ m) & = & \texttt{KN}\,(t,\ e',\ s,\ m) \\
\texttt{KN}\,(i+1,\ v \cdot e,\ s,\ m) & = & \texttt{KN}\,(i,\ e,\ s,\ m) \\
\texttt{KN}\,(t\,t',\ e,\ s,\ m) & = & \texttt{KN}\,(t,\ e,\ (t', e, m) \cdot s,\ m) \\
\texttt{KN}\,(\lambda t,\ e,\ s,\ m) & = & \texttt{aux}\,(s,\ (\lambda t, e, m)) \\
\\
\texttt{aux} & : & \mathit{Stack}''_{\texttt{KN}} \times (\mathit{ANForm}_{deB} \times \mathit{Cl}) \to \mathit{NForm}_{deB} \\
\texttt{aux}\,(M,\ a) & = & \texttt{aux}'\,(M,\ a) \\
\texttt{aux}\,((t, e, m) \cdot s,\ a) & = & \texttt{KN}\,(t,\ e,\ a \cdot s,\ m) \\
\texttt{aux}\,(M,\ (\lambda t, e, m)) & = & \texttt{KN}\,(t,\ \mathsf{V}\,m \cdot e,\ \Lambda \cdot M,\ m+1) \\
\texttt{aux}\,((t', e', m') \cdot s,\ (\lambda t, e, m)) & = & \texttt{KN}\,(t,\ (t', e') \cdot e,\ s,\ m) \\
\\
\texttt{aux}' & : & \mathit{MStack}_{\texttt{KN}} \times \mathit{NForm}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{aux}'\,(\bullet,\ n) & = & n \\
\texttt{aux}'\,(a \cdot s,\ n) & = & \texttt{aux}\,(s,\ a\,n) \\
\texttt{aux}'\,(\Lambda \cdot M,\ n) & = & \texttt{aux}'\,(M,\ \lambda n)
\end{array}$$

This simplified version of the KN-machine is in defunctionalized form with respect to the stack.

A direct comparison with the initial KN-machine Let us summarize the 'distance' from Crégut's definition of the KN-machine (extended to cope with open terms) to the above simplified version:

1. By including the level in stack closures, the need to pair the residual normal forms with a de Bruijn level was eliminated.
2. To follow the use of the stack we slightly modified the representation of the stack and disentangled the machine accordingly by separating the rules dispatching on the stack from the rest. The separation of the rules emphasized that the environment and the de Bruijn level are only used in some of the rules.
3. By pattern matching on the top element in the environment, one transition step was eliminated by short-circuiting and KN is adjusted to operate directly on λ-terms.

The correctness of the first simplification was informally justified. The other simplifications are immediate.

11.3 A corresponding higher-order direct-style normalization function

In this section we derive a higher-order direct-style normalization function that corresponds to the simplified version of the KN-machine. The normalization function is compositional. The derivation is yet another application of the standard transformations of the functional correspondence (Section 5.3).

Refunctionalizing the continuation The simplified version of the KN-machine is in defunctionalized form wrt. the stack: KN is an evaluation function in CPS; aux and aux′ together constitute the apply function for the data-structure continuation — the stack component of KN. Refunctionalizing the continuation yields a higher-order normalization function in CPS. We leave out this normalization function.

Direct-style transformation of the normalization function Applying a (call-by-value) direct-style transformation yields the (call-by-value) direct-style counter-part (where $ExpVal = ANForm_{deB} + Clos_{\mathtt{KN}}$):

$$\begin{array}{rcl}
\mathtt{normalize_{KN}} & : & Term_{deB} \rightarrow NForm_{deB} \\
\mathtt{normalize_{KN}}\ t & = & \mathtt{reify}\,(\mathtt{KN}\,(t,\ \bullet,\ 0)) \\
\\
\mathtt{KN} & : & Term_{deB} \times Env'_{\mathtt{KN}} \times Level \rightarrow ExpVal \\
\mathtt{KN}\,(i,\ \bullet,\ m) & = & i + m \\
\mathtt{KN}\,(1,\ \mathtt{V}\ m' \cdot e,\ m) & = & m - m' \\
\mathtt{KN}\,(1,\ (t, e') \cdot e,\ m) & = & \mathtt{KN}\,(t,\ e',\ m) \\
\mathtt{KN}\,(i + 1,\ v \cdot e,\ m) & = & \mathtt{KN}\,(i,\ e,\ m) \\
\mathtt{KN}\,(\lambda t,\ e,\ m) & = & (t,\ e,\ m) \\
\mathtt{KN}\,(t\,t',\ e,\ m) & = & \mathit{case}\ \mathtt{KN}\,(t,\ e,\ m) \\
 & & \quad \mathit{of}\ a \qquad\qquad \Rightarrow a\,(\mathtt{reify}\,(\mathtt{KN}\,(t',\ e,\ m))) \\
 & & \quad\ \ \mid (t,\ e',\ m') \Rightarrow \mathtt{KN}\,(t,\ (t', e) \cdot e',\ m') \\
\\
\mathtt{reify} & : & ExpVal \rightarrow NForm_{deB} \\
\mathtt{reify}\,a & = & a \\
\mathtt{reify}\,(t,\ e,\ m) & = & \lambda(\mathtt{reify}\,(\mathtt{KN}\,(t,\ \mathtt{V}\ m \cdot e,\ m + 1)))
\end{array}$$

This normalization function is evaluation-order dependent and assumes applicative-order evaluation of the metalanguage.

Converting denotable and expressible values In the previous paragraph denotable and expressible values are first-order. They can be seen as the result of a closure conversion, which is an in-place defunctionalization [27] of higher-order values. Refunctionalizing the expressible values and the denotable values requires new definitions. Both expressible and denotable values now contain function spaces:

$$\begin{array}{rcl}
DenVal & := & Level \rightarrow ExpVal \\
ExpVal & := & ANForm_{deB} \times ((Level \rightarrow DenVal) \rightarrow Int \rightarrow ExpVal) \\
Env''_{\mathtt{KN}} & := & DenVal\ list
\end{array}$$

We make the resulting normalization function compositional by separating the lookup of a variable from KN. We emphasize the denotational nature of this compositional function by letting KN map a λ-term to its denotational meaning:

$$
\begin{array}{rcl}
\texttt{normalize}_{\texttt{KN}} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} \\
\texttt{normalize}_{\texttt{KN}}\, t & = & \texttt{reify}\,(\texttt{KN}\, t \bullet 0) \\
\\
\texttt{KN} & : & \mathit{Term}_{deB} \to \mathit{Env}''_{\texttt{KN}} \to \mathit{Level} \to \mathit{ExpVal} \\
\texttt{KN}\, i\, e\, m & = & \texttt{lookup}\,(i,\ e,\ m) \\
\texttt{KN}\,(\lambda t)\, e\, m & = & \lambda g.\lambda i.\texttt{KN}\, t\,((g\ m)\cdot e)\,(m+i) \\
\texttt{KN}\,(t\, t')\, e\, m & = & \mathit{case}\ \texttt{KN}\, t\, e\, m \\
 & & \quad \mathit{of}\ a \Rightarrow a\,(\texttt{reify}\,(\texttt{KN}\, t'\, e\, m)) \\
 & & \quad\ \ \mid f \ \Rightarrow f\,(\lambda m'.\lambda m.\texttt{KN}\, t'\, e\, m)\, 0 \\
\\
\texttt{lookup} & : & \mathit{Index} \times \mathit{Env}''_{\texttt{KN}} \times \mathit{Level} \to \mathit{ExpVal} \\
\texttt{lookup}\,(i,\ \bullet,\ m) & = & i+m \\
\texttt{lookup}\,(1,\ v\cdot e,\ m) & = & v\, m \\
\texttt{lookup}\,(i+1,\ v\cdot e,\ m) & = & \texttt{lookup}\,(i,\ e,\ m) \\
\\
\texttt{reify} & : & \mathit{ExpVal} \to \mathit{NForm}_{deB} \\
\texttt{reify}\, a & = & a \\
\texttt{reify}\, f & = & \lambda(\texttt{reify}\,(f\,(\lambda m'.\lambda m.m-m')\, 1))
\end{array}
$$

The correctness of the resulting higher-order direct-style normalization function comes for free as a corollary of the correctness of each of the transformations in the functional correspondence.

11.4 A corresponding reduction semantics in a calculus of closures

In this section we apply the transformations of the syntactic correspondence (Section 5.2) starting from a reduction semantics and deriving the version from Section 11.2 of Crégut's KN-machine.

11.4.1 Definition of a reduction semantics

To define the reduction semantics, we must define the syntax, values, and a one-step reduction function on non-value terms.

Term language The syntactic units of the language are closures. The defining grammar is defined in a stratified fashion:

$$
\begin{array}{rrcl}
 & v & ::= & (t, s) \mid \mathtt{V}\ m \\
 & s & ::= & \bullet \mid v \cdot s \\
 & d & ::= & t[s, m] \mid d\ c \mid a \\
\mathit{Closure} & c & ::= & d \mid \lambda c
\end{array}
$$

Here $m \in \mathit{Level}$ denotes a de Bruijn level, $a \in \mathit{ANForm}_{deB}$ denotes as usual a normal form that is not an abstraction, and t is a de Bruijn-indexed λ-term.

Contraction rules Seven rules constitute the possible ways to perform reductions in the term language:

$$\begin{array}{rrcl}
\textit{Reindex}: & \mathtt{i}[\bullet, m] & \to & \mathtt{i} + m \\
\textit{Subst}: & \mathtt{1}[t[s', m'] \cdot s, m] & \to & t[s', m] \\
\textit{Subst}_{\mathtt{V}}: & \mathtt{1}[\mathtt{V}\ m' \cdot s, m] & \to & m - m' \\
\textit{Subst}': & (\mathtt{i}+\mathtt{1})[c \cdot s, m] & \to & \mathtt{i}[s, m] \\
\textit{App}: & (t\,t')[s, m] & \to & (t[s, m])\,(t'[s, m]) \\
\textit{Beta}: & ((\lambda t)[s, m])\,c & \to & t[c \cdot s, m] \\
\textit{Abs}: & (\lambda t)[s, m] & \to & \lambda(t[\mathtt{V}\ m \cdot s, m + \mathtt{1}])
\end{array}$$

Because the syntactic units of the language are closures, the contraction rules constitute a binary reduction relation on closures which is also a function. We exclude the contraction rule *Abs* to define a 'partition' of this function:

$$\textit{Weak} \quad := \quad \textit{Reindex} \cup \textit{Subst} \cup \textit{Subst}_{\mathtt{V}} \cup \textit{Subst}' \cup \textit{App} \cup \textit{Beta}$$

A one-step reduction function A strategy for reductions in closures is defined via a grammar of reduction contexts:

$$\begin{array}{rcl}
A[\,] & ::= & [\,] \mid A[\lambda[\,]] \mid C[a\,[\,]] \\
C[\,] & ::= & A[\,] \mid C[[\,]\,c]
\end{array}$$

This grammar represents reduction contexts inside-out and is the standard grammar presented and used in Section 7.2 and thereafter used in Section 8.2.2 and Section 10.7. The only difference is that c here ranges over the definition of closures of this chapter. We can define a one-step reduction function directly via the reduction contexts because the following unique decomposition property holds:

Proposition 7 (Unique decomposition)
A closure c *is either a* β*-normal form or it can be uniquely decomposed into context* $C[\,]$ *and potential redex* p_a *such that* $c = C[p_a]$ or *context* $A[\,]$ *and potential redex* p_c *such that* $c = A[p_c]$*, where potential redexes are defined as follows:*

$$\begin{array}{lrcl}
\textit{PotRedex}_a & p_a & ::= & \mathtt{i}[s, m] \mid (t\,t')[s, m] \mid (\lambda t)[s, m]\,c \\
\textit{PotRedex}_c & p_c & ::= & (\lambda t)[s, m]
\end{array}$$

We define the one-step reduction function as a restricted compatible closure of the contraction rules according to the above reduction contexts:

$$\begin{array}{rcll}
A[r] & \mapsto_{\mathtt{KN}} & A[r'], & \text{where } (r, r') \in \textit{Abs} \\
C[r] & \mapsto_{\mathtt{KN}} & C[r'], & \text{where } (r, r') \in \textit{Weak}
\end{array}$$

Well-definedness of the one-step reduction function follows immediately from the above unique decomposition property together with the fact that *Weak* is a function. $\mapsto_{\mathtt{KN}}$ is partial on *Closure* for two reasons: (1) the function is not defined on β-normal forms, and (2) not all potential redexes are actual redexes.

Normalization of de Bruijn-indexed λ-terms Iterating the one-step reduction function $\mapsto_{\mathtt{KN}}$ until (possibly) a normal form is obtained defines normalization of λ-terms.

11.4.2 Derivation of the KN-machine on defunctionalized form

With an explicit definition of a function to perform the unique decomposition and to plug a closure into a reduction context we can define the iterated one-step reduction more explicitly. Identifying an auxiliary function of the decomposition function to be a refocus function according to the grammar of contexts and the contraction rules we can change the normalization of terms from being reduction-based to be reduction-free.

Fusing the iterating function with the refocus function the specification becomes a state-transition system, i.e., an abstract machine.

The notion of closures is not needed after optimization by short-circuiting transitions. We can hence make the machine operate directly on λ-terms. The resulting abstract machine is the normalized version of Crégut's KN-machine from Section 11.2.

11.5 Summary

In this chapter the starting point was Crégut's KN-machine. This abstract machine performs strong normalization of closed λ-terms. We extended the KN-machine to handle all of $Term_{deB}$ and performed simple adjustments to put the machine on defunctionalized form.

We used the KN-machine as yet another example to illustrate that the *functional correspondence* applies also in the context of strong normalization: Via the standard transformations of the functional correspondence we were able to derive a corresponding higher-order direct-style normalization function following the operational framework of normalization-by-evaluation. This normalization function is compositional. The correctness of the normalization function follows immediately from the correctness of the individual transformations.

Furthermore, we used the KN-machine as yet another example to illustrate that the *syntactic correspondence* applies also in the context of strong normalization: We specified a reduction semantics that defined a reduction-based way to normalize λ-terms. The syntactic correspondence immediately justifies the correctness of the reduction semantics because the sound transformations let us derive the defunctionalized version of the KN-machine.

Chapter 12

An abstract head-order reduction machine

Kluge presents an abstract machine for strong reduction of closed λ-terms in his textbook [45, Section 6.4].[1] We refer to this machine as the `HOR`-*machine*. Kluge informally introduces three 'extended' contraction rules as the underlying normalization mechanism of the `HOR`-machine to justify the correctness of this machine.

Roadmap In Section 12.1, we introduce the three 'extended' contraction rules. We do that formally and justify their soundness and relate them to rules of the $\lambda\hat{\rho}$-calculus. In Section 12.2, we formally define the `HOR`-machine and extend it to also normalize open terms. In Section 12.3, we disentangle the `HOR`-machine and show that the `HOR`-machine as a clever implementation of the underlying normalization mechanism in essence coincide with Crégut's `KN`-machine from Section 11.2: The two machines coincide when specified in a disentangled way.

12.1 The normalization mechanism underlying the `HOR`-machine

In his textbook, Kluge introduces three rules to transform de Bruijn-indexed λ-terms. He lets de Bruijn indices start from `0`. For uniformity with the rest of this text, we shift the de Bruijn indices to start from `1`. The three rules are named

```
identity_reduction_in_the_large
β-distribution_in_the_large
η-extension_in_the_large
```

For simplicity, we denote these three rules by ι_l, π_l, and η_l, respectively.

The contraction rule ι_l The rule ι corresponds to the contraction of a series of consecutive β-redexes where the body of the innermost redex is an index:

$$\iota_l : \qquad (\overbrace{\lambda \ldots \lambda}^{n \text{ times}} i)\ t_n\ \ldots\ t_1 \quad \rightarrow \quad t_i$$

[1] Kluge credits Troullinos for the formal definition of this abstract machine.

No reindexing is needed in t_i. This rule is sound in the λ-calculus, i.e., the left-hand side is β-equivalent to the right-hand side for all i, and $t_1, \ldots, t_n$.

The contraction rule π_l The rule π_l distributes a series of consecutive β-redexes into an application:

$$\pi_l : \qquad (\overbrace{\lambda\ldots\lambda}^{n \text{ times}} t\, t')\, t_n \,\ldots\, t_1 \quad \rightarrow \quad ((\overbrace{\lambda\ldots\lambda}^{n \text{ times}} t)\, t_n \,\ldots\, t_1)\, ((\overbrace{\lambda\ldots\lambda}^{n \text{ times}} t')\, t_n \,\ldots\, t_1)$$

No β-reductions are performed. This rule is used to eventually be able to perform ι_l-reductions. Also the rule π_l is sound in the λ-calculus.

The contraction rule η_l It does not hold that it is always possible to either use ι_l or π_l on all λ-terms that are not β-normal forms.[2] Consider a term t_a which is a left-hand side of the rule π_l but with fewer argument terms t_i than the number of abstractions immediately outside the inner application. On the one hand it holds (according to the definition of β_{deB} in Section 1.6.2) that:[3]

$$\begin{array}{rcl} t_a & = & (\overbrace{\lambda\ldots\lambda}^{n \text{ times}} t'\, t'')\, t_n \,\ldots\, t_{n'}, \qquad n' > 1 \\ & =_\beta & \overbrace{\lambda\ldots\lambda}^{n'-1 \text{ times}} \mathtt{substitute}\,(\ldots(\mathtt{substitute}\,(t\,t',\ (n,\ t_n)))\ldots,\ (n',\ t_{n'})) \end{array}$$

On the other hand, letting $\lceil t \rceil^n$ denote t with all free indices lifted n it holds that:

$$\begin{array}{rcl} t_b & := & \overbrace{\lambda\ldots\lambda}^{n'-1 \text{ times}} \lceil t_a \rceil^{n'-1}\, (n'-1)\, \ldots 1 \\ & =_\beta & \overbrace{\lambda\ldots\lambda}^{n'-1 \text{ times}} \lceil \overbrace{\lambda\ldots\lambda}^{n'-1 \text{ times}} \mathtt{substitute}\,(\ldots(\mathtt{substitute}\,(t\,t',\ (n,\ t_n)))\ldots,\ (n',\ t_{n'}))\rceil^{n'-1}\, (n'-1)\, \ldots 1 \\ & =_\beta & \overbrace{\lambda\ldots\lambda}^{n'-1 \text{ times}} \mathtt{substitute}\,(\ldots(\mathtt{substitute}\,(t\,t',\ (n,\ t_n)))\ldots,\ (n',\ t_{n'})) \\ & =_\beta & t_a \end{array}$$

Because all free variables initially is greater than $n' - 1$ each application to an index corresponds to decrementing all free variables. In all $n' - 1$ times the free variables is decremented, which make t_b equivalent to t_a. By construction of t_b the distribution rule π_l can be applied to its body:

$$\lceil t_a \rceil^{n'-1}\, (n'-1)\, \ldots 1$$

The rule η_l just performs the construction of t_b in such cases to facilitate further distribution via π_l:

$$\eta_l : \qquad t_a \quad \rightarrow \quad \overbrace{\lambda\ldots\lambda}^{n'-1 \text{ times}} \lceil t_a \rceil^{n'-1}\, (n'-1)\, \ldots 1$$

Kluge presents a more general version of this rule which performs η-extensions on general terms instead of only the terms ranged over by t_a. By construction the rule as defined above is sound in the λ-calculus even when only considering β. Kluge's version is sound when adding η as presented in Section 3.1.

[2] Overall the terms in consideration are closed. This property implies that terms cannot take the form

$$\iota_l : \qquad (\overbrace{\lambda\ldots\lambda}^{n \text{ times}} i)\, t_{n'} \,\ldots\, t_1, \quad i > n$$

[3] Such a term corresponds to a partial application of a curried function.

Observation It is our observation that the rule π_l corresponds to the rule π of the $\lambda\widehat{\rho}$-calculus and ι_l corresponds to the rule ι of that calculus (Section 5.2.1). In the $\lambda\widehat{\rho}$-calculus the syntactic units are closures, i.e., there are explicit constructs for closures. With the closures seen as abbreviations of terms in $Term_{deB}$, π_l *is* π and ι_l *is* ι. No rule corresponding to a generalized version of β in the $\lambda\widehat{\rho}$-calculus is needed, when closures are abbreviations. The generalized version of β is the β^+ rule introduced by Biernacka and Danvy [8, Section 5.1], which is presented as a context-sensitive rule corresponding to the optimized evaluation used in Krivine's original specification of a call-by-name abstract machine for the λ-calculus [48,49].

Because the $\lambda\widehat{\rho}$-calculus is weak and Krivine's original machine is weakly normalizing they do not contain elements corresponding to η_l, which facilitates strong normalization.

12.2 The HOR-machine

The HOR-machine normalizes de Bruijn-indexed λ-terms where indices start from 0. For uniformity with the rest of this text, the de Bruijn indices are shifted to start from 1. The rules of the machine are adjusted accordingly.

Configurations Configurations of the HOR-machine are composed of five parts: (1) a λ-term t, (2) an environment e, (3) a stack s, (4) a level m indicating current de Bruijn level, and (5) a direction directive d. The λ-terms are the de Bruijn-indexed λ-terms $Term_{deB}$ from Section 1.6. Definitions of the rest of the entities (transliterated into our notation) read as follows:

$$\begin{array}{lrcl} Clos_{\mathtt{HOR}} & c & ::= & (t, e) \\ Env_{\mathtt{HOR}} & e & ::= & \bullet \mid c \cdot e \mid \mathtt{V}\ m \cdot e \\ Stack_{\mathtt{HOR}} & s & ::= & \bullet \mid @ \cdot t \cdot s \mid \Lambda \cdot s \mid c \cdot s \\ Level & m & ::= & \{0, 1, 2, \ldots\} \\ Dir & d & ::= & \uparrow \mid \downarrow \mid done \end{array}$$

An environment is a list, where the elements are closures (t, e) or tagged levels $\mathtt{V}\ m$.[4] Possible elements on the stack are closures, the special symbol Λ, and the special symbol @ followed by a λ-term. The possible directions indicate that the machine is either on its way 'down' normalizing the term-part ($\downarrow$), or on its way 'up' with a partial result ($\uparrow$), or in a final state with the term part being a normal form (*done*). Configurations are formally defined as follows:

$$Configuration_{\mathtt{HOR}} \quad := \quad Stack_{\mathtt{HOR}} \times Env_{\mathtt{HOR}} \times Term_{deB} \times Level \times Dir$$

The transition rules Kluge states the transition rules of the HOR-machine in an order-dependent way: The rules must be 'tried' in a specific order on actual machine configurations. In other words, some configurations are matched by several of the transition rules, and determinism of the machine is ensured by the specified order. Only two rules turn out to be influenced by the order-dependency. We eliminate this dependency by letting these two

[4] Kluge uses the terms *suspensions* and *unapplied lambdas counts (ULCs)* to denote closures and de Bruijn levels respectively.

rules be conditional. We transliterate the HOR-machine into our notation:[5]

$$
\begin{array}{rcll}
\mathtt{HOR} & : & \mathit{Configuration}_{\mathtt{HOR}} \to \mathit{Configuration}_{\mathtt{HOR}} & \\
\mathtt{HOR}\,(s,\ \mathtt{V}\ m' \cdot e,\ 1,\ m,\ \downarrow) & = & \mathtt{HOR}\,(s,\ e,\ m - m',\ m,\ \uparrow) & \\
\mathtt{HOR}\,(s,\ (t, e') \cdot e,\ 1,\ m,\ \downarrow) & = & \mathtt{HOR}\,(s,\ e',\ t,\ m,\ \downarrow) & \\
\mathtt{HOR}\,(s,\ v \cdot e,\ i+1,\ m,\ \downarrow) & = & \mathtt{HOR}\,(s,\ e,\ i,\ m,\ \downarrow) & \\
\mathtt{HOR}\,(s,\ e,\ t\,t',\ m,\ \downarrow) & = & \mathtt{HOR}\,((t', e) \cdot s,\ e,\ t,\ m,\ \downarrow) & \\
\mathtt{HOR}\,(s,\ e,\ \lambda t,\ m,\ \downarrow) & = & \mathtt{HOR}\,(s',\ (t', e') \cdot e,\ t,\ m,\ \downarrow), & \text{if } s = (t', e') \cdot s' \\
\mathtt{HOR}\,(s,\ e,\ \lambda t,\ m,\ \downarrow) & = & \mathtt{HOR}\,(\Lambda \cdot s,\ \mathtt{V}\ m \cdot e,\ t,\ m+1,\ \downarrow), & \text{if } s \neq (t', e') \cdot s' \\
\mathtt{HOR}\,(\bullet,\ e,\ t,\ m,\ \uparrow) & = & (\bullet,\ e,\ t,\ m,\ \mathit{done}) & \\
\mathtt{HOR}\,(@ \cdot t' \cdot s,\ e,\ t,\ m,\ \uparrow) & = & \mathtt{HOR}\,(s,\ e,\ t'\,t,\ m,\ \uparrow) & \\
\mathtt{HOR}\,(\Lambda \cdot s,\ e,\ t,\ m,\ \uparrow) & = & \mathtt{HOR}\,(s,\ e,\ \lambda t,\ m-1,\ \uparrow) & \\
\mathtt{HOR}\,((t', e') \cdot s,\ e,\ t,\ m,\ \uparrow) & = & \mathtt{HOR}\,(@ \cdot t \cdot s,\ e',\ t',\ m,\ \downarrow) &
\end{array}
$$

From these transition rules it is seen that the machine implements both ι_l and η_l in small step. The first conditional rule in small steps navigates in the term to find the form where either ι_l or π_l apply. If it is not possible to directly obtain such a form a use of η_l followed by π_l is needed. The second conditional rule in small steps perform the introduction of the added abstractions and implicitly performs the navigation of the following use of π_l by directly placing $\mathtt{V}\ m$ in the environment. Also the the free variables are incremented (lazily) in small steps by incrementing the de Bruijn level.

Strong normalization via the HOR-machine Normalization of a closed λ-term t via the HOR-machine is defined as (1) initializing the machine with t by constructing an initial configuration, (2) running the machine starting form the initial configuration, and (3) extracting the term part from the terminal configuration. The initial configuration is $(\bullet,\ \bullet,\ t,\ 0,\ \downarrow)$ where the stack and the environment is empty, the term is t, the current de Bruijn level is 0, and the direction directive is $\downarrow$ indicating that t in general is not a normal form. Normalization of closed t thus reads as follows:

$$
\begin{array}{rcl}
\mathtt{normalize}_{\mathtt{HOR}} & : & \mathit{Term}_{deB} \to \mathit{Term}_{deB} \\
\mathtt{normalize}_{\mathtt{HOR}}\ t & = & t', \quad \text{if } \mathtt{HOR}\,(\bullet,\ \bullet,\ t,\ 0,\ \downarrow) = (s,\ e,\ t',\ m,\ \mathit{done})
\end{array}
$$

Kluge informally justifies the following correctness theorem of the HOR-machine:

Theorem 7
Let t *a be closed term and* t' *be a* β*-normal form.*
Then $t =_\beta t' \iff \mathtt{normalize}_{\mathtt{HOR}}\ t = t'$

$\mathtt{normalize}_{\mathtt{HOR}}$ is partial because it is not defined for open λ-terms and for closed terms only when the term is β-equivalent to a normal form. To be defined on *open* terms (β-equivalent to a normal form) the HOR-machine must include a rule matching a de Bruijn index occurring in a configuration with an empty environment. We add one transition rule to facilitate normalization of open terms with normal forms:

$$\mathtt{HOR}\,(s,\ \bullet,\ i,\ m,\ \downarrow) \quad = \quad \mathtt{HOR}\,(s,\ \bullet,\ i+m,\ m,\ \uparrow)$$

[5] In the sixth transition rule $\mathtt{V}\ m$ is placed in the environment. In Kluge's specification $\mathtt{V}\ (m+1)$ is used. The difference comes from letting de Bruijn indices start from 0 instead of 1.

12.3 Simplifying the HOR-machine

Eliminating the direction directive and emphasizing partial results We inline the 'unloading' to short-circuit one transition:

$$\begin{array}{rcl} \mathtt{HOR}\,(\bullet,\ e,\ t,\ m,\ \uparrow) & = & (\bullet,\ e,\ t,\ m,\ \mathit{done}) \\ & = & t \end{array}$$

In every state where the direction directive is $\downarrow$ the term component is a general λ-term. When the directive is $\uparrow$, the term component is a normal form. We separate the rules into two, to make the distinction between general λ-terms and terms on normal form more explicit. This separation also makes the direction directive in configurations superfluous.

The above observation also gives that the term placed on the stack in the last rule of HOR is a normal form. A simple analysis gives that this normal formal is not an abstraction, i.e., it is an $a \in \mathit{ANForm}_{deB}$ defined on page 16. We change the representation of stacks accordingly:

$$\mathit{Stack}'_{\mathtt{HOR}} \qquad s \quad ::= \quad \bullet \mid a \cdot s \mid \Lambda \cdot s \mid (t, e) \cdot s$$

Finally the first three parts of the machine are rearranged to follow the rest of this text:

$$\begin{array}{rcll} \mathtt{normalize_{HOR}} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} & \\ \mathtt{normalize_{HOR}}\ t & = & \mathtt{HOR}\,(t,\ \bullet,\ \bullet,\ 0) & \\ & & & \\ \mathtt{HOR} & : & \mathit{Term}_{deB} \times \mathit{Env}_{\mathtt{HOR}} \times \mathit{Stack}'_{\mathtt{HOR}} \times \mathit{Level} \to \mathit{NForm}_{deB} & \\ \mathtt{HOR}\,(i,\ \bullet,\ s,\ m) & = & \mathtt{aux}\,(s,\ \bullet,\ i+m,\ m) & \\ \mathtt{HOR}\,(1,\ \mathtt{V}\ m' \cdot e,\ s,\ m) & = & \mathtt{aux}\,(s,\ e,\ m-m',\ m) & \\ \mathtt{HOR}\,(1,\ (t, e') \cdot e,\ s,\ m) & = & \mathtt{HOR}\,(t,\ e',\ s,\ m) & \\ \mathtt{HOR}\,(i+1,\ v \cdot e,\ s,\ m) & = & \mathtt{HOR}\,(i,\ e,\ s,\ m) & \\ \mathtt{HOR}\,(t\,t',\ e,\ s,\ m) & = & \mathtt{HOR}\,(t,\ e,\ (t', e) \cdot s,\ m) & \\ \mathtt{HOR}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{HOR}\,(t,\ (t', e') \cdot e,\ s',\ m), & \text{if } s = (t', e') \cdot s' \\ \mathtt{HOR}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{HOR}\,(t,\ \mathtt{V}\ m \cdot e,\ \Lambda \cdot s,\ m+1), & \text{if } s \neq (t', e') \cdot s' \\ & & & \\ \mathtt{aux} & : & \mathit{Stack}'_{\mathtt{HOR}} \times \mathit{Env}_{\mathtt{HOR}} \times \mathit{NForm}_{deB} \times \mathit{Level} \to \mathit{NForm}_{deB} & \\ \mathtt{aux}\,(\bullet,\ e,\ n,\ m) & = & n & \\ \mathtt{aux}\,(a \cdot s,\ e,\ n,\ m) & = & \mathtt{aux}\,(s,\ e,\ a\,n,\ m) & \\ \mathtt{aux}\,(\Lambda \cdot s,\ e,\ n,\ m) & = & \mathtt{aux}\,(s,\ e,\ \lambda n,\ m-1) & \\ \mathtt{aux}\,((t', e') \cdot s,\ e,\ a,\ m) & = & \mathtt{HOR}\,(t',\ e',\ a \cdot s,\ m) & \end{array}$$

Disentangling with respect to the stack We observe that the argument environment in aux is never used. That argument can hence be removed from aux. Also, the de Bruijn level is only used in aux when the right-hand sub-term of an application must be normalized. The de Bruijn level is incremented when entering an abstraction for normalization and is decremented when generating the corresponding abstraction as part of the resulting normal form. By storing the level together with the right-hand sub-term and the environment, the level is available if that term must be normalized. In other words, the level is also superfluous in aux and can be discarded. The modified abstract machine reads as follows (where $\mathit{Stack}''_{\mathtt{HOR}}$ is an adjusted definition of stacks to cope with the above mentioned changes).

$$
\begin{array}{rcll}
\mathtt{normalize}_{\mathtt{HOR}} & : & \mathit{Term}_{deB} \to \mathit{NForm}_{deB} & \\
\mathtt{normalize}_{\mathtt{HOR}}\ t & = & \mathtt{HOR}\,(t,\ \bullet,\ \bullet,\ 0) & \\
& & & \\
\mathtt{HOR} & : & \mathit{Term}_{deB} \times \mathit{Env}_{\mathtt{HOR}} \times \mathit{Stack}''_{\mathtt{HOR}} \times \mathit{Level} \to \mathit{NForm}_{deB} & \\
\mathtt{HOR}\,(i,\ \bullet,\ s,\ m) & = & \mathtt{aux}\,(s,\ i+m) & \\
\mathtt{HOR}\,(1,\ V\ m' \cdot e,\ s,\ m) & = & \mathtt{aux}\,(s,\ m-m') & \\
\mathtt{HOR}\,(1,\ (t,e') \cdot e,\ s,\ m) & = & \mathtt{HOR}\,(t,\ e',\ s,\ m) & \\
\mathtt{HOR}\,(i+1,\ v \cdot e,\ s,\ m) & = & \mathtt{HOR}\,(i,\ e,\ s,\ m) & \\
\mathtt{HOR}\,(t\,t',\ e,\ s,\ m) & = & \mathtt{HOR}\,(t,\ e,\ (t',e,m) \cdot s,\ m) & \\
\mathtt{HOR}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{HOR}\,(t,\ (t',e') \cdot e,\ s',\ m), & \text{if } s = (t',e',m') \cdot s' \\
\mathtt{HOR}\,(\lambda t,\ e,\ s,\ m) & = & \mathtt{HOR}\,(t,\ V\ m \cdot e,\ \Lambda \cdot s,\ m+1), & \text{if } s \neq (t',e',m') \cdot s' \\
& & & \\
\mathtt{aux}\,(\bullet,\ n) & = & n & \\
\mathtt{aux}\,(a \cdot s,\ n) & = & \mathtt{aux}\,(s,\ a\,n) & \\
\mathtt{aux}\,(\Lambda \cdot s,\ n) & = & \mathtt{aux}\,(s,\ \lambda n) & \\
\mathtt{aux}\,((t,e,m) \cdot s,\ a) & = & \mathtt{HOR}\,(t,\ e,\ a \cdot s,\ m) &
\end{array}
$$

If we move the dispatch on the stack from HOR to aux only aux dispatch on the stack and the HOR-machine becomes in defunctionalized form with respect to the stack. We identify that after such a final adjustment, the disentangled HOR-machine coincides with the disentangled KN-machine from Section 11.2. We hence leave further analysis of the HOR-machine and refer to Chapter 11 where we presented a functionally corresponding normalization function and a syntactically corresponding reduction semantics.

12.4 Summary

In this chapter our starting point was three sound rules, which are informally explained by Kluge [45, Section 6.4] to be the foundation or strategy of an abstract machine for strong normalization — the HOR-machine. The rule η_l is presented in relation to π_l: If not either ι_l or π_l apply one use of η_l prepare for a use of π_l. We observed the relation between these three rules, the contraction rules in the (generalized) $\lambda\hat{\rho}$-calculus and Krivine's original machine for weak normalization.

We presented strong normalization via the HOR-machine and extended the specification to cope with open terms. We disentangled the HOR-machine which let us eliminate unneeded parts in the new kind of configurations. We saw in Section 11.2 the result of this simple transformation: Disentangling the HOR-machine yields the disentangled KN-machine. In other words, two independently presented abstract machines for strong normalization are the same abstract machine when stated in a disentangled way.

Crégut presents the KN-machine as an extension of what we now call the Krivine machine (Section 4.5.2). In other words, Crégut's starting point is normal-order reduction to weak head normal forms, where β-redexes are contracted according the possible evaluation contexts $E ::= [\,] \mid E\ t$. From Kluge's perspective, β-redexes in these contexts should not be contracted but instead distributed towards the indices, where the contractum of the generalized contraction rule ι_l is trivial. An abstract machine should then perform an implementation of the navigation to π_l-redexes or preparation of such redexes by implementing η_l. The HOR-machine is such an abstract machine implementing the rules cleverly including a lazy implementation of liftings of free variables. We showed that the KN-machine and the

HOR-machine — regardless of the different starting points — in essence are the same abstract machine. We observe that the clever (lazy) implementation of β-reductions by extending an environment in the KN-machine corresponds to constructing a π_1-redex in the HOR-machine.

Chapter 13

Conclusion, future work, and perspectives

Conclusion In this thesis, we have studied (1) Church's λ-calculus, which is *standard material*, (2) the functional correspondence and the syntactic correspondence in relation to weak normalization, which is *known material*, and (3) the functional correspondence and the syntactic correspondence in relation to strong normalization, which is *new material*.

Within the standard material, we were able to identify a general relationship between Church numerals and Scott numerals, which is new. This general relationship emphasized a course-of-value representation underlying both the Church numerals and the Scott numerals. This representation then enabled us to improve on existing representations of functions.

With the known material, we started from a calculus that includes state variables. We defined an applicative-order one-step reduction function for weak normalization in this calculus and we showed that evaluation by iterating this function syntactically corresponds to the well-known CEK machine extended with state variables, an assignment construct, and a component holding the global state.

The second part of the thesis was dedicated to the new material: we considered the syntactic correspondence and the functional correspondence in relation to strong normalization.

- We presented a simple calculus with explicit substitution as a direct offspring of an implementation of β-contraction. By construction, this calculus inherits the standard properties from the λ-calculus. We then showed that the syntactic correspondence also applies to strong normalization in that we derived an abstract machine corresponding to a strategy in the calculus and we connected it to another one independently due to Lescanne.

- Starting from Crégut's `KN`-machine and Curien's normalizer, we showed that the functional correspondence mechanically links normalization functions and abstract machines and that the syntactic correspondence mechanically links abstract machines and calculi with deterministic strategies.

- Finally we identified that two independently presented abstract machines for strong normalization — Crégut's `KN`-machine and the `HOR`-machine — in essence are the same machine.

Future work and perspectives In Chapter 6, we showed that the syntactic correspondence also applies in the presence of mutable variables. In the contraction rules of the λ_v-S($\mathbf{t}$)-calculus, reduction contexts were used in a non-standard fashion, but still the syntactic correspondence applied, and we were able to derive a corresponding abstract machine. Moreau uses also reduction contexts in such a non-standard fashion in his definition of a reduction semantics for a language including dynamic bindings [55, page 251]. Before Moreau's work, it was only implementational folklore that there exists a translation, the *dynamic-environment passing transformation*, which translates programs using dynamic variables into programs using lexical variables only. After such a translation, evaluation can be implemented with an abstract machine using a dynamic stack of bindings. Entering the body of a 'dynamic let'-expression the new binding is pushed onto the dynamic stack, and leaving the body the binding is popped off again. Lookups of dynamic variables are then performed according to the current dynamic stack. Such an abstract machine implements the idea of threading the dynamic environment. We conjecture that Moreau's reduction semantics gives rise to that abstract machine, using the syntactic correspondence.

Other possibilities include applying the syntactic correspondence to Maraist, Odersky, and Wadler's call-by-need λ-calculus [54] and to Ariola and Felleisen's call-by-need calculus [6]. The result would be an abstract machine for call-by-need, a topic that has only been partially explored [4] [9, Section 9].

So, all in all, studying functional programming and the λ-calculus has put us in position to contribute to λ-definability and to syntactic and semantic artifacts about weak normalization and strong normalization, and to understand them better.

Acknowledgements

First and foremost I would like to thank my supervisor Olivier Danvy. From Day One of interacting with him about this thesis, I have felt how much he engages himself in work, even with a MSc student. His infectious commitment also to my work made me join the game on Day One. I am also grateful to Olivier for his patience with my English writing and for his encouragements. Greetings like 'Happy working' and 'Happy hacking' I will never forget.

Secondly, I would like to thank Kevin Millikin for our joint work on strong normalization. His patient and thorough explanation of especially refunctionalization of closures in normalization functions clarified things a lot. I also owe Andrzej Filinski much for his insightful and useful comments.

Furthermore, I am grateful to Kristian Støvring for letting me borrow his copy of Barendregt's treatise on the λ-calculus, and to the administrative staff at DAIMI. Barendregt's book, an office in the 'right' corridor of the Turing building, and our seemingly never-failing computing facilities were instrumental for writing this thesis.

Finally, I would like to thank my wife Annemette and our daughter Mabel for their love.

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 1985.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.

[3] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the research report BRICS RS-04-3.

[5] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the research report BRICS RS-04-28.

[6] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[7] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[8] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 2006. To appear. Available as the research report BRICS RS-06-3.

[9] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375:76–108, 2007. Extended version available as the research report BRICS RS-06-18.

[10] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[11] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[12] William D. Clinger. Proper tail recursion and space efficiency. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998. ACM Press.

[13] Pierre Crégut. An abstract machine for lambda-terms normalization. In Wand [69], pages 333–340.

[14] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3), 2007. To appear. A preliminary version was presented at the 1990 ACM Conference on Lisp and Functional Programming.

[15] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[16] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhaüser, 1993.

[17] Haskell B. Curry, J. Roger Hindley, and Robert Feys. *Combinatory Logic: Volume II*. North Holland, 1972.

[18] Oliver Danvy and Kevin Millikin. A simple application of lightweight fusion to proving the equivalence of abstract machines. Technical Report BRICS RS-07-8, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2007.

[19] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).

[20] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[21] Olivier Danvy. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

[22] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk.

[23] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [69], pages 151–160.

[24] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[25] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's J operator. Research Report BRICS RS-06-17, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2006. A preliminary version appears in the proceedings of 17th International Workshop on the Implementation and Application of Functional Languages. Accepted to Logical Methods in Computer Science.

[26] Olivier Danvy and Kevin Millikin. Refunctionalization at work. Research Report BRICS RS-07-7, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2007. Invited talk at the 8th International Conference on Mathematics of Program Construction, MPC '06.

[27] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.

[28] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[29] Nicholas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

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

[31] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. <http://www.cs.utah.edu/plt/publications/pllc.pdf>, 1989-2006.

[32] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[33] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[34] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.

[35] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 18–27, Paris, France, September 1999. ACM Press.

[36] Mayer Goldberg. *Recursive Application Survival in the λ-Calculus*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, May 1996.

[37] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[38] Chris Hankin. *Lambda Calculi, a guide for computer scientists*, volume 1 of *Graduate Texts in Computer Science*. Oxford University Press, 1994.

[39] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[40] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997. Extended version available as the research report BRICS RS-97-7.

[41] Fairouz Kamareddine and Alejandro Rìos. A lambda-calculus à la de Bruijn with explicit substitutions. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Seventh International Symposium on Programming Language Implementation and Logic Programming*, number 982 in Lecture Notes in Computer Science, pages 45–62, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[42] Fairouz Kamareddine and Alejandro Rìos. Extending a lambda-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, 1997.

[43] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[44] Stephen C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52–67, January 1981.

[45] Werner E. Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005.

[46] Dexter C. Kozen. *Automata and Computability*. Springer, 1997.

[47] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Brown University, third edition, 2003–2006. Available online at `<http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/>`.

[48] Jean-Louis Krivine. Un interprète du λ-calcul. Brouillon. Available online at `<http://www.pps.jussieu.fr/~krivine/>`, 1985.

[49] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3), 2007. To appear. Available online at `<http://www.pps.jussieu.fr/~krivine/>`.

[50] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[51] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[52] Peter J. Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In Olivier Danvy, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW'97)*, Technical report BRICS NS-96-13, University of Aarhus, pages 1:1–9, Paris, France, January 1997.

[53] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\nu$ a journey through calculi of explicit substitutions. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 60–69, Portland, Oregon, January 1994. ACM Press.

[54] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

[55] Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.

[56] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[57] Peter D. Mosses. A foreword to 'Fundamental concepts in programming languages'. *Higher-Order and Symbolic Computation*, 13(1/2):7–9, 2000.

[58] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 143–154, New York, NY, USA, January 2007. ACM Press.

[59] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[60] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[61] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [63].

[62] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[63] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[64] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[65] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[66] Dana S. Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, 1971.

[67] Christopher Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming, Copenhagen, Denmark, August 1967. Reprinted in Higher-Order and Symbolic Computation 13(1/2):11–49, 2000, with a foreword [57].

[68] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

[69] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[70] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

Recent BRICS Report Series Publications

RS-08-3 **Johan Munk.** ***A Study of Syntactic and Semantic Artifacts and its Application to Lambda Definability, Strong Normalization, and Weak Normalization in the Presence of State*****. April 2008. xi+144 pp.**

RS-08-2 **Gudmund Skovbjerg Frandsen and Piotr Sankowski.** ***Dynamic Normal Forms and Dynamic Characteristic Polynomial*****. April 2008. 21 pp. To appear in ICALP '08.**

RS-08-1 **Anders Møller.** ***Static Analysis for Event-Based XML Processing*****. jan 2008. 23 pp. Appears in PLAN-X '08.**

RS-07-18 **Jan Midtgaard.** ***Control-Flow Analysis of Functional Programs*****. December 2007. iii+38 pp.**

RS-07-17 **Luca Aceto, Willem Jan Fokkink, and Anna Ingólfsdóttir.** ***A Cancellation Theorem for 7BCCSP*****. December 2007. 30 pp.**

RS-07-16 **Olivier Danvy and Kevin Millikin.** ***On the Equivalence between Small-Step and Big-Step Abstract Machines: A Simple Application of Lightweight Fusion*****. November 2007. ii+11 pp. To appear in Information Processing Letters (extended version). Supersedes BRICS RS-07-8.**

RS-07-15 **Jooyong Lee.** ***A Case for Dynamic Reverse-code Generation*****. August 2007. ii+10 pp.**

RS-07-14 **Olivier Danvy and Michael Spivey.** ***On Barron and Strachey's Cartesian Product Function*****. July 2007. ii+14 pp.**

RS-07-13 **Martin Lange.** ***Temporal Logics Beyond Regularity*****. July 2007. 82 pp.**

RS-07-12 **Gerth Stølting Brodal, Rolf Fagerberg, Allan Grønlund Jørgensen, Gabriel Moruz, and Thomas Mølhave.** ***Optimal Resilient Dynamic Dictionaries*****. July 2007.**

RS-07-11 **Luca Aceto and Anna Ingólfsdóttir.** ***The Saga of the Axiomatization of Parallel Composition*****. June 2007. 15 pp. To appear in the Proceedings of CONCUR 2007, the 18th International Conference on Concurrency Theory (Lisbon, Portugal, September 4–7, 2007), Lecture Notes in Computer Science, Springer-Verlag, 2007.**

RS-07-10 **Claus Brabrand, Robert Giegerich, and Anders Møller.** ***Analyzing Ambiguity of Context-Free Grammars*****. May 2007. 17 pp. Full version of paper presented at CIAA '07.**