universidad politécnica de madrid
Escuela Técnica Superior de
Ingenieros Informáticos

Operational Aspects of Full Reduction in Lambda Calculi

PhD Thesis

Álvaro García Pérez

October 2014

Departamento de Lenguajes,
Sistemas Informáticos e Ingeniería de Software
Escuela Técnica Superior de
Ingenieros Informáticos

Operational Aspects of Full Reduction in Lambda Calculi

Submitted in partial fulfilment
of the requirements for the degree of
Doctor en Software y Sistemas

Author

ÁLVARO GARCÍA PÉREZ
Máster en Tecnologías para el Desarrollo de Sistemas Software Complejos
IMDEA Software Institute and Universidad Politécnica de Madrid

Advisors

JUAN JOSÉ MORENO NAVARRO
Catedrático de Universidad
IMDEA Software Institute and Universidad Politécnica de Madrid

PABLO NOGUEIRA
Profesor Ayudante Doctor
Universidad Politécnica de Madrid

October 2014

Summary

This thesis studies full reduction in lambda calculi. In a nutshell, full reduction consists in evaluating the body of the functions in a functional programming language with binders. The classical (i.e., pure untyped) lambda calculus is set as the formal system that models the functional paradigm. Full reduction is a prominent technique when programs are treated as data objects, for instance when performing optimisations by partial evaluation, or when some attribute of the program is represented by a program itself, like the type in modern proof assistants.

A notable feature of many full-reducing operational semantics is its hybrid nature, which is introduced and which constitutes the guiding theme of the thesis. In the lambda calculus, the hybrid nature amounts to a 'phase distinction' in the treatment of abstractions when considered either from outside or from inside themselves. This distinction entails a layered structure in which a hybrid semantics depends on one or more subsidiary semantics.

From a programming languages standpoint, the thesis shows how to derive implementations of full-reducing operational semantics from their specifications, by using program transformations techniques. The program transformation techniques are syntactical transformations which preserve the semantic equivalence of programs. The existing program transformation techniques are adjusted to work with implementations of hybrid semantics. The thesis also shows how full reduction impacts the implementations that use the environment technique. The environment technique is a key ingredient of real-world implementations of abstract machines which helps to circumvent the issue with binders.

From a formal systems standpoint, the thesis discloses a novel consistent theory for the call-by-value variant of the lambda calculus which accounts for full reduction. This novel theory entails a notion of observational equivalence which distinguishes more points than other existing theories for the call-by-value lambda calculus. This contribution helps to establish a 'standard theory' in that calculus which constitutes the analogous of the 'standard theory' advocated by Barendregt in the classical lambda calculus. Some proof-theoretical results are presented, and insights on the model-theoretical study are given.

Resumen

Esta tesis estudia la *reducción plena* ('full reduction' en inglés) en distintos cálculos lambda.[1] En esencia, la reducción plena consiste en evaluar los cuerpos de las funciones en los lenguajes de programación funcional con ligaduras. Se toma el cálculo lambda clásico (*i.e.*, puro y sin tipos) como el sistema formal que modela el paradigma de programación funcional. La reducción plena es una técnica fundamental cuando se considera a los programas como datos, por ejemplo para la optimización de programas mediante evaluación parcial, o cuando algún atributo del programa se representa a su vez por un programa, como el tipo en los demostradores automáticos de teoremas actuales.

Muchas semánticas operacionales que realizan reducción plena tienen naturaleza híbrida. Se introduce formalmente la noción de naturaleza híbrida, que constituye el hilo conductor de todo el trabajo. En el cálculo lambda la naturaleza híbrida se manifiesta como una 'distinción de fase' en el tratamiento de las abstracciones, ya sean consideradas desde fuera o desde dentro de si mismas. Esta distinción de fase conlleva una estructura en capas en la que una semántica híbrida depende de una o más semánticas subsidiarias.

Desde el punto de vista de los lenguajes de programación, la tesis muestra como derivar, mediante técnicas de transformación de programas, implementaciones de semánticas operacionales que reducen plenamente a partir de sus especificaciones. Las técnicas de transformación de programas consisten en transformaciones sintácticas que preservan la equivalencia semántica de los programas. Se ajustan las técnicas de transformación de programas existentes para trabajar con implementaciones de semánticas híbridas. Además, se muestra el impacto que tiene la reducción plena en las implementaciones que utilizan entornos. Los entornos son un ingrediente fundamental en las implementaciones realistas de una máquina abstracta.

Desde el punto de vista de los sistemas formales, la tesis desvela una teoría novedosa para el cálculo lambda con paso por valor ('call-by-value lambda calculus' en inglés) que es consistente con la reducción plena. Dicha teoría induce una noción de equivalencia observacional que distingue más puntos que las teorías existentes para dicho cálculo. Esta contribución ayuda a establecer una 'teoría estándar' en el cálculo lambda con paso por valor que es análoga a la 'teoría estándar' del cálculo lambda clásico propugnada por Barendregt. Se presentan resultados de teoría de la demostración, y se sugiere como abordar el estudio de teoría de modelos.

[1] Traducimos el adjetivo inglés 'full' como 'pleno' (con el significado de 'exhaustivo') en vez de como el más habitual 'completo', para evitar confusión con el inglés 'complete' que significa algo diferente en este contexto.

Acknowledgements

I would like first to thank the people that enabled me to do a PhD. My PhD advisors, Juan José Moreno Navarro and Pablo Nogueira, and my advisor of the final project of my grade in Ingeniería Informática, Nelson Medinilla. Juanjo, I could never be more grateful for the opportunity you gave to me. Pablo, you did not only take the responsibility to guide me through my doctoral studies, but the determination to help me in achieving the key skills and manners that science requires. Nelson, thanks for telling me which doors I should knock when I was still dubious about pursuing a research career.

Many people helped me by teaching how to do science during my graduate courses at Universidad Politécnica de Madrid. Thanks are specially due to Pablo Nogueira, Julio Mariño, Germán Puebla, Lars-Åke Fredlund, Samir Genaim, and Manuel Carro. I am only fortunate to have attended some graduate courses abroad and on line. My gratitude to Andreas Döring, Olivier Danvy, and Frank Pfenning. I complemented my education in the Oxford Spring School 2010 and in the Oregon Summer School 2013. I had the pleasure to learn from Ralf Hinze, Oleg Kiselyov, Simon Peyton-Jones, Jeremy Siek, and Stephanie Weirich while in Oxford. The courses by Robert Harper, Frank Pfenning, Simon Peyton-Jones, Stephanie Weirich, Amal Ahmed, and Andrew Tolmach at Oregon were specially illuminating.

I also had the opportunity to enjoy academic stays at Oxford University Computing Laboratory and at Aarhus University. Thanks are due to Bruno Oliveira, Jeremy Gibbons, Meng Wang, Ralf Hinze, Nicolas Wu, Richard Bird, and Geraint Jones from Oxford University Computing Laboratory, and to Olivier Danvy, Jan Midtgaard, Ian Zerny, and Erik Ernst from Aarhus University. I am specially grateful for the interest that Peter Sestoft and Lars Birkedal showed during my one-day visit to ITU Copenhagen.

My sincere gratitude to César Muñoz, Hélène Kirchner, Elvira Albert, Shin-Cheng Mu, Tom Schrijvers, Ricardo Peña, Wei-Ngan Chin, and Jurriaan Hage for chairing the venues in which my work has been published. And thanks also to Elvira Albert, Shin-Cheng Mu, Ricardo Peña, and Tom Schrijvers for their role as guests editors in the Journal of Science of Computer Programming. I am in debt to the anonymous reviewers of all these venues, in particular the external reviewers for the special issue of my PEPM'13 paper in the Journal of Science of Computer Programming. They genuinely helped me to disentangle the formal principles of my contribution from the presentational aspects. My gratitude to them is not rhetorical.

Thanks to the external examiners of my dissertation, Olivier Danvy and Herbert Kuchen. I really appreciate your valuable time. Thanks to the two experts from DLSIIS,

Julio Mariño and Germán Puebla, you put me in the track of a good presentation and of a smooth and fluent defence. And thanks to the committee members: Julio Mariño, Santiago Escobar, Małgorzata Biernacka, Jan Midtgaard and Pierre-Yves Strub. Małgorzata and Jan, your comments yielded a thorough reflection on the work and, I hope, they will also yield a fruitful discussion in the future. You were the perfect guardians of rigour. My work benefited in great extent from your comments and points of view.

I always counted with the support and understanding of my colleagues at Babel Group: Emilio Gallego, Ángel Herranz, Julio Mariño, Guillem Marpons, Jamie Gabbay, Lars-Åke Fredlund, Clara Benac, Álvaro Fernández, Susana Muñoz, Ana María Fernández, James Lipton, Elena Gómez-Martínez, Iván Pérez, Victor Pablos Ceruelo, and Raul Alborodo.

No less important was the stimulating atmosphere at the IMDEA Software Institute. Thanks to the three Manueles (Hermenegildo, Clavel, and Carro) for steering the ship. Thanks to Gilles Barthe for his friendly and always constructive criticism, to John Gallagher for his advice, and to Aleksandar Nanevski, Anindya Banerjee, Ilya Sergey, Germán Delbianco, Mark Marron, Ruy Ley-Wild, Noam Zeilberger, and César Sánchez for their feedback and comradeship. Juan Manuel, Julián, César, Jürgen, Santiago, Federico, Artem, Carol, thanks for sharing with me the everyday space. Miguel Ángel, Remy, J. Fran, Teresa, Alex, Javier, José Miguel, Eugenia, Goran, Antonio, Umer, Alejandro, Zoé, Guido, Lucio, Martín, Pedro, Dragan, Zorana, Boris, Juan, Pierre, Pavithra, Michael, Alexey, Salvador, Miriam, Joaquín, Natalia, Platon, Andrea, Giovanni, thanks for keeping IMDEA as the friendly place that it is and always was. And thanks to the numerous colleagues and visitors that have spent time at IMDEA during these last years.

Nothing would have been possible without the technical support and management: Ana María Fernández, María Alcaraz, Paola Huerta, Tania Rodríguez, Carlota Gil, Andrea Ianetta, Juan Céspedes, and Roberto Lumbreras, thanks for that.

My thesis is made up of articles. I could not have written these articles without the help of my collaborators. Pablo, Emilio, Juanjo, Ilya, Pierre-Yves, you were excellent mirrors in which to project my ideas. Thanks for this thrilling interchange.

Among the numerous encounters in scientific venues, I would like to emphasise those with Kenichi Asai, Beniamino Accattoli, Oleg Kiselyov, Flavien Breuvart, Alberto Carraro, Jeremy Siek, Ronald Garcia, Jacob Johannsen, Thomas Ehrhard, Frank Pfenning, Amal Ahmed, Stephanie Weirich, and Simon Peyton-Jones.

This research has been partially funded by the Spanish Ministerio de Ciencia e Innovación through project DESAFIOS10 TIN2009-14599, and by Comunidad de Madrid through programme PROMETIDOS P2009/TIC-1465. I have been supported by Comunidad de Madrid grant CPI/0622/2008 and by IMDEA Software Institute. I am deeply grateful for their essential financial support.

I am also grateful to Luca Aceto and Anna Ingólfsdóttir for providing me with the opportunity to keep doing research in the NoSOS project at Reykjavík University.

Finally, I am grateful to my family and friends, who were always patience and helpful. Eugenia, I would have never started a PhD without your encouragement. I am in debt to you for your cheer and sympathy in the difficult moments. Tina, you provided decisive

support in the second half of my journey, sometimes shaky, but always lively. I am only too grateful for the help you offered while polishing my defence and for your warm presence.

Disclaimer

This thesis builds on several published and submitted works that I have co-authored.

Journal publications:

- On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. With Pablo Nogueira. *Science of Computer Programming* **95**(2), 176-199, Elsevier (2014).

Conference publications (peer-reviewed):

- A syntactic and functional correspondence between reduction semantics and reduction-free full normalisers. With Pablo Nogueira. In: *ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation* (2013).
- Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order. With Pablo Nogueira and Juan José Moreno Navarro. In: *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (2013).
- Deriving interpretations of the gradually-typed lambda calculus. With Pablo Nogueira and Ilya Sergey. In: *ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation* (2014).

Extended abstracts (peer-reviewed):

- The Beta-Cube. With Pablo Nogueira and Emilio Jesús Gallego Arias. In: *1st International Workshop on Strategies in Rewriting, Proving, and Programming* (2010).
- A standard theory for the pure lambda-value calculus. With Pablo Nogueira. In: *11th International Workshop on Domain Theory and Application* (2014).

I contributed in the elaboration of all of them as the first author.

Contents

List of Figures

List of Tables

Preface

> En los [lenguages] del hemisferio boreal [...] el sustantivo se forma por acumulación de adjetivos. [...] En el caso elegido la masa de adjetivos corresponde a un objeto real; el hecho es puramente fortuito. En la literatura de este hemisferio [...] abundan los objetos ideales, convocados y disueltos en un momento, según las necesidades poéticas. Hay objetos compuestos de dos términos. [...] Los hay de muchos. [...] Esos objetos de segundo grado pueden combinarse con otros; el proceso, mediante ciertas abreviaturas, es practicamente infinito. El hecho de que nadie crea en la realidad de los sustantivos hace, paradójicamente, que sea interminable su número.[2]
>
> (*Tlön, Uqbar, Orbis Tertius*, Jorge Luis Borges)

The lambda calculus is a formal system whose terms denote functions. Similar to the nouns in the imaginary world of Tlön, the meaning of a function in the lambda calculus is only conveyed *intensionally*. This is, a function is specified as a particular combination of (the invocations of) other functions, which contrasts with the *extensional* specification of a function consisting of a (possibly infinite) collection of pairs relating inputs with return values. The lambda calculus is the outcome of the early investigations on the intensional aspect of mathematics carried out by a generation of logicians in the late nineteenth and early twentieth century.

In elementary mathematics a function is denoted by $F(x) =$ 'some expression of x', where the variable x is called the formal parameter and the 'some expression of x' is the body of the function. In the lambda calculus this is written as the term $\lambda x.B$, where the term B encodes the body 'some expression of x' which calculates the result returned by the function. The term $\lambda x.B$ is called an 'abstraction', where the binding symbol λx signals that the formal parameter x is abstracted in the body B.[3] Formally, this means that the free occurrences of variable x in term B are to be replaced by the argument to which the function is applied. (The free occurrences of a variable in a term are those which

[2] In [the languages of] the boreal hemisphere [...] the nouns are build up by aggregating adjectives. [...] For the case in point, the mass of adjectives corresponds to a real object; however this is solely fortuitous. In the literature of this hemisphere [...] the ideal objects abound, conveyed and dissolved at once according only to poetical needs. Some objects are made up by two terms. [...] Others are made up by many terms. [...] These second-degree objects can be combined with each other; the process, by means of some abbreviations, is virtually infinite. Paradoxically, the lack of belief in the reality of nouns makes them innumerable. (Translation by the author.)

[3] For illustration, other symbols with binding character are the quantifiers $\forall x, \exists x$ in logic, and the integral $\int dx$ in mathematics.

are not bound by any λ.) The abstractions in the lambda calculus denote mathematical functions. These functions are anonymous, *i.e.*, they are defined *in situ* and applied to their arguments if any, which explains the absence of the function name 'F' in the term representation $\lambda x.B$.

For example, consider the identity function, $ID(x) = x$, which is written as the lambda term $\lambda x.x$. Intensionally, the identity function is encoded as an abstraction that only returns its formal parameter. Thus, given any argument n (and a term N encoding it) the application $ID(n)$ is written $(\lambda x.x)(N)$. (Customarily, parenthesis are dropped from applications and the latter is written $(\lambda x.x)N$.) By the definition of ID, the application $(\lambda x.x)N$ is equivalent to its operand N. This equivalence embodies a contraction which represents an elementary step of computation. The contraction is specified by a rule (called 'β-rule' for historical reasons) which stipulates how a reducible expression is replaced by a more elementary term. The β-rule reads $(\lambda x.B)T \to_\beta [T/x]B$, where $[\_/\_]\_$ is an external substitution function which replaces by T the free occurrences of variable x in B. For instance, $(\lambda x.x)N \to_\beta N$. The term 'reducible expression' is customarily abbreviated to 'redex' (or 'redices' if plural).

Encoding functions by combining lambda terms is a process of programming in this setting of intensional definitions, which resembles the construction of nouns in the boreal hemisphere of Tlön. For instance, consider the mathematical $TWICE(f, y) = f(f(y))$ which defines a rule of computation that takes two arguments—a function f and a datum y—and applies the former twice to the latter. In the lambda calculus this is written $\lambda f.\lambda y.f(f\,y)$. The two nested lambdas '$\lambda f.\lambda y.\ldots$' stand for the two formal parameters of $TWICE$—a binary function is just a unary function that takes the first argument, and returns another unary function that takes the second argument—and the applications of the f to the y are written $f(f\,y)$ according to the parenthesis-dropping convention.[4] The β-rule provides the operational framework for the minimalistic programming language defined by the lambda calculus. Consider the program $TWICE(ID, n)$ and the lambda term $(\lambda f.\lambda y.f(f\,y))(\lambda x.x)N$ encoding it (recall $\lambda x.x$ stands for ID and N for n). The program can be executed (*i.e.*, evaluated or reduced) by successively applying the β-rule to the program redices until reaching a result of computation (*e.g.*, a term without redices, called a normal form). The following reduction sequence, where N is assumed to be in normal form, illustrates:

$$\underline{(\lambda f.\lambda y.f(f\,y))(\lambda x.x)}N \to_\beta \underline{(\lambda y.(\lambda x.x)((\lambda x.x)\,y))n}$$
$$\to_\beta \underline{(\lambda x.x)((\lambda x.x)\,N)} \to_\beta \underline{(\lambda x.x)N} \to_\beta N$$

The above reduction sequence precisely matches the evaluation of the program in point according to elementary mathematics, *i.e.*, $TWICE(ID, n) = ID(ID(n)) = ID(n) = n$. In each step, the redex being contracted is underlined.

[4] In addition to dropping parenthesis from applications, it is standard that application associates to the left and that λ has greater precedence than application. Thus, the term $\lambda x.(\lambda y.B)M\,N$ stands for $(\lambda x.(((\lambda y.B)M)N))$.

In its pure version the lambda calculus only consists of a denumerable set of variables x, y, *etc.*, of abstractions $\lambda x.B$, and of applications $M\,N$. To take the analogy of the languages in Tlön further, the objects that can be represented in the lambda calculus are not different from the 'means to make them up'. Let us think of Tlön adjectives as predicates, *i.e.*, formulae that take an argument and qualify it, like the predicate 'big x' which qualifies object x. Interestingly, the objects themselves can only be represented by predicates, *i.e.*, the nouns do not exist by their own, but as the consequence of combining adjectives. Similarly, the mathematical objects underneath pure lambda terms consist only of functions. There is no primitive data other than functions, and a mathematical domain D for this data would have to be isomorphic to the space of functions within itself, *i.e.*, functions with source D and image D. In the lambda calculus, and in the domains of mathematical objects that it represents, functions are inherently *higher-order functions*, this is, they take and return functions.

The pure lambda calculus can be enlarged with *applied features*, which consider a domain of primitive data and primitive operators over them, and specify additional contraction rules that implement those primitive operators. Although applied versions of the lambda calculus are convenient and widely used, the pure version alone is enough to define computable functions. As stated by the Church-Turing Thesis (Church, 1936; Turing, 1937) any computation that could be mechanised (*i.e.*, any general recursive function over the natural numbers) can be represented as a term of the pure lambda calculus.[5]

Turning the β-rule into a symmetric relation induces an equivalence relation (called β-equivalence or $=_\beta$) which can be used to manipulate programs in an algebraic way and to reason about equivalence of programs. Consider the two programs

$$\begin{array}{lcccc} PROG_1(x) & = & \ldots & TWICE(ID,x) & \ldots \\ PROG_2(x) & = & \ldots & ID(x) & \ldots \end{array}$$

where the ellipsis '$\ldots\ \ldots$' stand for the same program context in which $TWICE(ID,x)$ and $ID(x)$ are respectively plugged. Since $TWICE(ID,x) =_\beta ID(x)$ (here the mathematical notation and the term encoding are used indistinctly) we can conclude that $PROG_1$ and $PROG_2$ are equivalent. This is known as the principle of *indiscernibility of identicals*, or *Leibniz's law*. This principle is supported by the principle of *referential transparency*, which enforces that evaluating a function does not encompass any side-effect, *i.e.*, given a function F and an argument n the application $F(n)$ delivers the same result regardless of when or where such application appears. Referential transparency, together with the higher-order functions that we mentioned above, are the defining features of the functional programming paradigm. The lambda calculus (or some of its variants) lays at the core of nowadays functional programming languages used worldwide.

[5] In order to define the computable functions, the natural numbers and the booleans are represented as functions through what is known as the *Church encoding*. The details of this or other encodings, and of how general recursion is represented on top of them, are not presented here. The interested reader is referred to (Hindley & Seldin, 2008, Chapter 4).

Since a term may have several redices the β-reduction so far does not define a deterministic way to execute our programs. Consider the program example $TWICE(ID, n)$ and its reduction sequence above. One may have chosen to contract the innermost redex $ID(n)$ before contracting the outermost redex $ID(ID(n))$. The alternative reduction sequence would have resulted:

$$\underline{(\lambda f.\lambda y.f(f\,y))(\lambda x.x)N} \rightarrow_\beta \underline{(\lambda y.(\lambda x.x)((\lambda x.x)\,y))N}$$
$$\rightarrow_\beta (\lambda x.x)(\underline{(\lambda x.x)\,N}) \rightarrow_\beta \underline{(\lambda x.x)N} \rightarrow_\beta N$$

(Notice that in the third step the redex being contracted is the inner $(\lambda x.x)N$.)

A reduction strategy fixes the order in which the redices in a term are contracted, and specifies a deterministic operational framework (*i.e.*, an operational semantics) for the lambda calculus as a programming language. Although this seems irrelevant for the program example above (the result is the same no matter which strategy is used) the choice of a reduction strategy has a paramount impact when assuming that some programs may not terminate. Consider the term $\Omega \equiv (\lambda x.x\,x)(\lambda x.x\,x)$. The self-application '$x\,x$' in the body of Ω's operator $\lambda x.x\,x$ makes the term to contract to itself, *i.e.*, $(\lambda x.x\,x)(\lambda x.x\,x) \rightarrow_\beta (\lambda x.x\,x)(\lambda x.x\,x)$. The term Ω lacks any normal form and entails an infinite reduction sequence $\underline{\Omega} \rightarrow_\beta \underline{\Omega} \rightarrow_\beta \ldots$ where in each step there is always a redex candidate to be contracted. A term with an infinite reduction sequence stands for a computation that gets stuck without producing any output. Such a term denotes the mathematical 'undefined', customarily written $\perp$. Now consider the term $\lambda z.N$ where fresh variable z is picked such that z does not appear free in N. Since z does not have any effect in N, the term $\lambda z.N$ denotes a constant function that always returns n, *i.e.*, $CONST_n(z) = n$. (The subindex in $CONST_n$ is just to distinguish this function from other constant functions with return values different from n.) The reduction strategy determines whether a function is strict or non-strict in its argument. (A strict function always returns an undefined result if its argument is undefined, a non-strict function may not.) The program $CONST_n(\perp)$ is encoded as the term $(\lambda z.N)\Omega$. The outermost strategy entails the reduction sequence $\underline{(\lambda z.N)\Omega} \rightarrow_\beta N$, which delivers result n and corresponds to non-strict functional semantics. The innermost strategy entails the reduction sequence $(\lambda z.N)\underline{\Omega} \rightarrow_\beta (\lambda z.N)\underline{\Omega} \rightarrow_\beta \ldots$ which reduces forever (recall this is assimilated to 'undefined') and corresponds to strict functional semantics.

The outermost and innermost strategies are representatives of the two foremost calling policies in programming languages, known respectively as call-by-name and call-by-value. A calling policy determines how the argument of a function is treated in a function application. With call-by-name the formal parameter of a function is a mere signifier (*i.e.*, a name) that refers to the argument program. Hence, the signifier could be discarded inside the body of the function without further ado, and the argument program would only be retrieved if the formal parameter is in turn applied to some other argument. With call-by-value the formal parameter of a function is as the argument program itself, *i.e.*, the argument program is meant to be evaluated to a value before the body of the function is

even considered.[6] There is a tension between the two policies. Call-by-value may diverge for programs that do have a result under call-by-name, but call-by-name may replicate the redices in the argument program along the body of the function, in which case call-by-value is proven more efficient. The controversy cannot be settled since each of the strategies is convenient in particular settings.[7]

Furthermore, what a program and a value are varies among different definitions of operational semantics. Programs can be considered as 'specifications of computation' or as 'inputs to a reduction sequence', and values can be considered as 'results of computation' or as 'irreducible terms'. These two acceptations (respectively to programs and values) may not coincide with each other. The first acceptation is customary in the programming languages community. A program cannot have free variables (*i.e.*, has to be a closed term), since variables are only placeholders which by themselves do not define any computation. A value consists of a function (*i.e.*, an abstraction, whether it contains redices in its body or not) since a function alone is a passive element that only waits for some argument to be applied to, and thus it is a result of computation.[8] According to this, call-by-name and call-by-value never deal with free variables and never reduce inside the body of abstractions. The second accepting is customary in the formal systems community. Only the terms with no redices at all (not even under lambda) are irreducible terms, otherwise the β-rule would still be applicable. When reducing the body of an abstraction, dealing with free variables is compulsory because, by definition, every occurrence of the formal parameter is free in the body's scope. According to this, call-by-name and call-by-value have to deal with free variables and have to reduce inside the body of abstractions.

Although undeveloped, the second acceptation is also useful in the context of programming languages. This is the case when programs are treated as objects, *i.e.*, when the terms are manipulated with purposes different than their evaluation, for instance when optimising a program or when some attribute of a program is represented by a term. In these cases, contracting the redices inside the bodies of abstractions (which is known as full reduction) is at need.

There is no general consensus on how the customary reduction strategies in programming languages are transposed into the 'reduction sequence' acceptation. Whenever the redices 'under lambda' are eligible for contraction the space of strategies explodes and their features become singular. For instance, a call-by-value-alike strategy can be specified in which the results of computation coincide with the normal forms (*i.e.*, irreducible terms), but which only enforces the arguments in redices to be reduced up to abstractions, regardless of whether they are in normal form or not. Besides being full-reducing, such a strategy has other desirable properties, the most remarkable is being complete. Completeness means that the strategy will always deliver a normal form which is equivalent to the

[6] This distinction is reminiscent of the use-mention distinction in analytical philosophy (Quine, 1940, Section 4).

[7] There are strategies that combine, up to some degree, the ability of call-by-name to not diverge in the presence of certain inputs with the efficiency of call-by-value. However, for illustration purposes call-by-name and call-by-value are enough here.

[8] A function is a black box that can only be observed by providing arguments to it.

input program if such a normal form exists, for particular notions of normal form and of equivalence which are peculiar of a setting with strict functional semantics. In comparison, the customary call-by-value strategy fails to deliver a normal form (*i.e.*, values may have redices under lambda) and transposing this strategy naively into the second acceptation (*i.e.*, reducing the arguments in redices up to normal form instead of up to abstraction) yields a strategy which fails to be complete.

The study of such full-reducing strategies, and the repercussion that full reduction has in the lambda calculus meta-theory, is the topic of this thesis.

1
Introduction

> Il fine di una buona introduzione definitiva è che il lettore si accontenti di questa, capisca tutto, e non legga più il resto.[1]
>
> (*Come si fa una tesi di laurea*, Umberto Eco)

A computer program is the specification of a computation written in a programming language. The program is *evaluated* when the operations specified by the program are executed by means of some computing device, typically a computer or an abstract machine.

A reduction system consists of a language of expressions (a.k.a. terms) and one or more contraction rules, each of them specifying an elementary step of computation by prescribing how a reducible expression (a redex) is to be replaced by a more elementary expression. A term is *reduced* when the redices (plural of 'redex') in it are successively contracted. This process induces a reduction relation whose pairs make up the reduction theory of the system.

Reduction systems are crafted to model the features of programming languages and to study their formal semantics. In this setting, 'evaluation' and 'reduction' refer to the process of executing a computation, respectively in the programming languages world and the reduction systems world. Although similar in spirit, 'evaluation' and 'reduction' carry contrasting connotations. The former is subject to practical considerations (whether a program is valid, or whether a result is observable) which emphasise the semantic aspect of computation, whereas the latter mirrors the internals of the reduction system (terms, redices, contraction rules, *etc.*) which emphasise the syntactic aspect of computation.

The lambda calculus (Church, 1941; Barendregt, 1984) is the formal system that lays on the basis of the functional programming languages (Landin, 1964; Scott & Strachey, 1971; Stoy, 1979). The traditional syntax of lambda terms is specified by the pseudo-grammar $\Lambda ::= x \mid (\lambda x.\Lambda) \mid (\Lambda\,\Lambda)$, where x, y, etc, range over the elements of a countably infinite

[1] The aim of a good definitive introduction is that the reader contents him or herself with it, understands everything, and skips the rest of the text. (Translation by the author.)

set of variables. The computational part of the lambda calculus hinges on the β-rule $(\lambda x.B)N \to_\beta [N/x]B$, which specifies how to contract redices in a term. (Section 2.1 of this thesis provides a thorough description of the calculus and its reduction theory.) A minimalistic functional programming language would take closed terms as programs, which would be evaluated by means of some customary abstract machine, *e.g.*, the Krivine Abstract Machine (Crégut, 1990; Krivine, 2007).

With this scenario in mind, the evaluation–reduction connotations concern primarily whether the redices in the bodies of abstractions are contracted or not. In the context of programming languages these redices are never contracted since abstractions themselves are considered as results of evaluation. This is because a function could only be observed by providing arguments to it. It is *customary* to dispense with free variables because a program is a closed term. In the context of reduction systems any redex could be picked for contraction, even the ones which lay 'under lambda'. The results of reduction are the normal forms, *i.e.*, the terms with no redices at all. It is *compulsory* to deal with free variables because in the body of an abstraction the formal parameter may pop up, which by definition is free in the body's scope.

In this thesis, the two connotations above are referred to as 'weak reduction' and 'full reduction' respectively. Weak reduction prevails in the practice of programming languages. For a realistic programming language, evaluating up to abstractions is more than enough when executing a program (Landin, 1964; Felleisen & Friedman, 1986; Felleisen & Flatt, 2002; Krivine, 2007). Furthermore, the weak-reducing theories have good model-theoretical properties, the most notable is the existence of initial models in any of the conventional categories for semantic domains (Abramsky, 1990; Egidi *et al.*, 1991). However, when a program is manipulated with purposes different than its evaluation, full reduction may not only be desirable, but compulsory. This is the case for program optimisation by partial evaluation, and for the conversion rule in type-checkers of proof assistants (Crégut, 1990).

Numerous works on full reduction exist in the literature (McGowan, 1970; Crégut, 1990; Sestoft, 2002; Grégoire & Leroy, 2002; Crégut, 2007; Munk, 2008). However, the full-reducing operational frameworks have remained underdeveloped, in comparison with the conventional weak-reducing ones, which predominate. This thesis helps to cover this gap by studying full-reducing strategies. The contributions are twofold. In the context of reduction systems, the full-reducing strategies are analysed and new meta-theoretic results for full-reduction are provided. Many full-reducing strategies are *hybrid*, in their definition depends on one or more *subsidiary* strategies. In the context of programming languages, it is shown how to implement the aforementioned strategies by deriving efficient abstract machines from them. The hybrid character of the strategies is used to adjust the existing techniques for program transformation to obtain first-order abstract machines with a single control stack.

The study of lambda calculus and related systems has been more than profuse in the past four decades. Devoting a PhD thesis to the operational aspects of the *pure untyped* version might look nowadays as awkward or outmoded, if not entirely obsolete. However, full-reduction stands at the basis of advanced techniques in functional programming, like

program optimisation and type checking in proof assistants (Crégut, 1990), and 'going under lambda' shows up the paramount issue with binders, *i.e.*, reasoning locally in a scope where the binding of a free variable is not available (Aydemir *et al.*, 2008). This thesis aims to show that the operational aspects of full reduction in the untyped lambda calculi still deserve attention.

1.1 Full reduction and hybrid strategies

Working with a reduction theory directly is cumbersome because the reduction relation lacks determinism, *i.e.*, an input term is not uniquely mapped to an equivalent term. A reduction strategy makes the computation deterministic by imposing an order in which the redices in a term are to be contracted (see Section 2.1 for a formal definition of a strategy).

Given an input term T_0, a reduction strategy entails a unique reduction sequence $T_0, T_1, T_2, \ldots$ where T_{n+1} follows from T_n by a single step of reduction. The reduction sequence may be finite if T_0 has an equivalent term which is irreducible according to the strategy, or it may be infinite otherwise (*i.e.*, T_0 diverges). The reduction sequence stands for the trace of execution of T_0 according to the strategy.

Broadly, 'full reduction' refers to reduction strategies that progress inside the body of abstractions. 'Full reduction' is preferred over 'normalisation' because the latter is often used in connection with big-step semantics, for any particular notion of irreducible term. For example, the weak-reducing Krivine abstract machine (Crégut, 1990; Krivine, 2007) normalises terms to weak head normal form (Figure 2.2 defines the customary irreducible terms in the lambda calculus). 'Strong reduction', as used by some authors (Grégoire & Leroy, 2002; Crégut, 2007; Munk, 2008), is also avoided because the latter can be confused with 'strong normalisation' which is a property of a calculus in which any full-reducing strategy is complete (Barendregt, 1984).

There are strategies that contract redices under lambda but only up to some notion of irreducible term which is different from normal form proper. (The normal forms are the terms which do not have any redex at all, see Section 2.1.) Some of these strategies are referred to as 'head'—head reduction (Barendregt, 1984), head spine order (Sestoft, 2002), ahead machine (Paolini & Ronchi Della Rocca, 1999). The 'head' qualifier is connected to the notions of head reduction and head normal form, which are paramount in the study of lambda theories (Barendregt, 1984, Chapter 4 and Part IV) since the syntactic characterisation of *solvability* and of *Böhm trees* rely on them. (Solvability and Böhm trees are, in turn, essential to define the operational relevance of a term (Barendregt, 1971; Wadsworth, 1976; Barendregt, 1984).)

One prime observation is that the behaviour of some existing full-reducing strategies changes when they are invoked over certain subterms of the input term. The strategy may reduce those subterms *less* in order to uphold some property. A salient example is the normal order strategy (Curry & Feys, 1958; Barendregt, 1984; Sestoft, 2002). Normal

order contracts the leftmost-outermost redex first.[2] Intuitively, given an abstraction $\lambda x.B$, normal order 'goes under lambda' and reduces B to nf. However, given an application $M\,N$, if M reduces in an arbitrary number of steps to an arbitrary abstraction $\lambda x.B$, at that point the leftmost-outermost redex is $(\lambda x.B)N$ and normal order must contract that redex and not the redices in B. Since normal order reduces abstractions fully it cannot invoke itself recursively on M. It must rely on a less reducing strategy, one that does not reduce abstractions. In other words, it must rely on the weak-reducing call-by-name (Barendregt, 1984; Plotkin, 1975; Sestoft, 2002). By using the subsidiary call-by-name, normal order upholds *completeness*, *i.e.*, the strategy always finds a normal form if there exists some that is equivalent to the input term.

This layered character of normal order is in accord with the idea of hybrid strategies, a terminology taken from (Sestoft, 2002) where it is used informally. Hybrid strategies are the guiding theme of this thesis. (A thorough description of some prominent hybrid strategies is presented in Chapter 3, and the formal definition of the *hybrid nature* of a strategy is delayed to Section 5.3.)

The hybrid nature of the strategies that contract 'under lambda' has been amply echoed in the literature, although seldom in an explicit way:

- In the folklore, it is known that normal order (Curry & Feys, 1958; Barendregt, 1984) has to rely on the weak-reducing call-by-name to reduce operators in applications.

- The *normalisation by evaluation* (NBE) approach (Berger & Schwichtenberg, 1991; Danvy, 1996) reveals the phase distinction inherent to a full-reducing algorithm which is implemented in terms of evaluation (*i.e.*, weak reduction). With NBE, normalisation is defined as the composition of *two* functions: evaluation proper, and a readback loop which reifies values and propagates normalisation into the bodies of abstractions. In the *typed* original setting of NBE (Berger & Schwichtenberg, 1991; Danvy, 1996) the paramount issue with binders is consigned to the η-expansion of meta-level function bodies. In (Filinski & Rohde, 2004, 2005), the NBE solution is transposed to an untyped setting via a residualising model where—additionally to the η-expansion of function bodies—the domain contains a single type of terms (*i.e.*, injected terms) and functions. These injected terms are akin to the neutral terms and ground terms of Chapters 5 and 6 of this thesis.

- Paolini & Ronchi Della Rocca (1999) introduce the *ahead machine* $\Downarrow_a$ which constitutes the analogous of head reduction in a strict-functional-semantics framework where arguments are reduced up to weak normal form. The ahead machine is a strategy consisting of a *single* function, instead of the composition of *two* functions present in NBE. Yet the ahead machine has a layered character, and distinguishes *two* different reduction modes, $\Downarrow_a^0$ and $\Downarrow_a^1$. The former depends on the latter, and the latter coincides with reduction to weak normal form.

[2] The *leftmost* redex according to the definition in (Curry & Feys, 1958, p. 140).

- Sestoft (2002) introduces the *uniform/hybrid* terminology informally and glimpses the problem underneath full-reducing strategies. His aim was to unify varying definitions for the conventional strategies in programming languages and to connect them to the pure calculus and to the theory of reduction. Sestoft collects the natural semantics of several paramount strategies in the literature, and notices the dependencies between some of the full-reducing ones (which he calls *hybrid*) and the ones which are defined only in terms of themselves (which he calls *uniform*). Chapter 3 develops this intuition further by systematising the space of strategies unveiled by Sestoft, and Section 5.3 provides a definition of the uniform/hybrid nature of a strategy, giving formal grounds to the informal terminology in (Sestoft, 2002).

- Grégoire & Leroy (2002) redeploy the NBE approach and introduce a full-reducing strategy which relies on the optimised Zinc Abstract Machine (Leroy, 1991), a machine that performs weak reduction. Their strategy is the analogous of normal order in the strict-functional-semantics framework where arguments are reduced up to weak normal forms.

- Munk (2008) presents context-based definitions of reduction strategies (Felleisen, 1987) where the grammar of reduction contexts is *stratified* (*i.e.*, layered) meaning that the productions for the contexts rely on the occurrence of an auxiliary non-terminal symbol which does not coincide with the start symbol of the grammar. The formal definition of hybrid nature in Section 5.3 of this thesis states that the grammar for reduction contexts of a strategy with hybrid nature is necessarily layered.[3]

1.2 Call-by-value and the lambda-value calculus

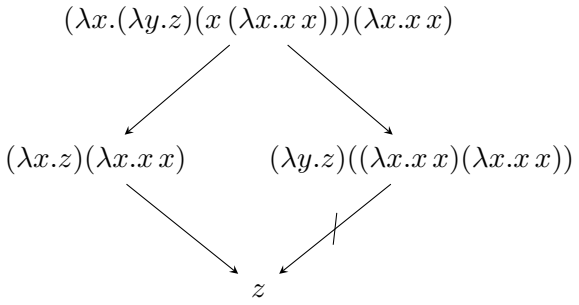
The lambda-value calculus (Plotkin, 1975) is the variant of the lambda calculus that corresponds to the SECD machine introduced by (Landin, 1964). This machine underlies the programming languages that implement a *by-value* calling policy. The calling policy refers to the way in which arguments in a function call are treated according to a given operational semantics. The by-value policy specifies that any argument has to be evaluated before being considered as the input of a function. The by-value policy implements strict functional semantics (a.k.a., *eager* semantics).

In the lambda-value calculus, a redex is contracted only when the redex's operand is a value. Here 'value' has a precise technical meaning as a term which is not an *application*, *i.e.*, an *abstraction* or a *variable*. (A thorough description of the lambda-value calculus and its reduction theory is given in Section 2.1.) Considering abstractions and variables as values is adequate with respect to the conventional call-by-value semantics in the programming languages world. Abstractions stand for functions, which are results of evaluation.

[3]The 'layered contexts' has a technical meaning which is connected to 'iterated CPS' and 'n-level CPS'. This thesis introduces a 'hybrid approach' for semantic artefacts which is an alternative to the 'n-level CPS approach'. The reduction contexts in both the n-level CPS and the hybrid approaches have stratified nature, and thus the epithet 'layered' is used for both approaches.

Variables stand for 'arguments of functions' (notice that programs are closed and every variable is the formal parameter of some abstraction), which range over values according to the by-value policy.

In the context of eager semantics, the requirement 'values as non-applications' transcends the practice of programming languages and pervades the theory of reduction systems. A reduction theory for eager semantics would lack consistency if applications were allowed in values, even if those applications were irreducible (Plotkin, 1975). For illustration, the application $x\,(\lambda x.x\,x)$ is irreducible, but it should not be considered a value because substituting the free x in it by the term $(\lambda x.x\,x)$ delivers $(\lambda x.x\,x)(\lambda x.x\,x)$, which is a divergent term. Consider the example term $M \equiv (\lambda x.(\lambda y.z)(x\,(\lambda x.x\,x)))(\lambda x.x\,x)$ in (Plotkin, 1975). The innermost redex in M has the application $x\,(\lambda x.x\,x)$ as operand. If irreducible applications were allowed in values, the following reduction diagram would be possible:



In the left part of the diagram, the innermost redex is contracted first, which discards the application $x\,(\lambda x.x\,x)$. Then, the residual of the outermost redex is contracted, which delivers z. In the right part of the diagram, the outermost redex is contracted first. This makes up the divergent operand $(\lambda x.x\,x)(\lambda x.x\,x)$ in the residual of the innermost redex, which cannot be contracted. In one path, the application is discarded. In the other path, the application is turned into a divergent term. The two paths cannot be joined together, *i.e.*, the system lacks confluence, which ultimately makes the system inconsistent. Plotkin (1975) shows that in order to recover confluence it is enough to filter out applications from values, which forbids the left path in the diagram above and restores confluence. The intuition behind this requirement could be summarised as 'preserving confluence by preserving potential divergence'.

Although the calculus was crafted to model eager semantics in the programming languages world, the reduction theory entailed by the requirement 'values as non-applications' is richer than the theory induced by conventional call-by-value semantics. Values should not be understood as results of evaluation, but rather as a side condition in the contraction rule which upholds confluence. The reduction system obtained is amenable to full reduction and is endowed with a notion of lambda-value normal forms which are the irreducible terms, *i.e.*, the terms without redices in which the operand is a value. (Such redices

are called β_V-redices, to distinguish them from the redices in the classical lambda calculus.) The lambda-value normal forms contain *stuck terms*, which consist of applications that do not reduce to a β_V-redex. There has been controversy regarding the full-reducing character of the lambda-value calculus and the role of stuck terms. Ronchi Della Rocca & Paolini (2004) argue that some lambda-value normal forms containing stuck terms are observationally equivalent to terms that are not operationally relevant (*i.e.*, meaningless terms), and thus the lambda-value normal forms should be considered uninteresting. This claim is sustained by a notion of operational relevance based on the notion of call-by-value solvability introduced in (Paolini & Ronchi Della Rocca, 1999) and further described in (Accattoli & Paolini, 2012). However, call-by-value solvability only characterises operational relevance according to conventional call-by-value, *i.e.*, in the weak-reducing version of the lambda-value calculus.[4]

An alternative calling policy for eager semantics appears recurrently in the literature where, instead of values, operands are restricted to *weak normal forms*. (Section 2.1 gives the definition of weak normal forms and of some other irreducible terms in the lambda calculus.) In this thesis, this calling policy is referred to as *by-weak-normal-form*, or just *by-wnf*. Similar to conventional call-by-value, the by-wnf policy requires the operands to be reduced weakly. Differently from conventional call-by-value, the by-wnf policy adds free variables and allows applications in operands, thus restricting the operands to arbitrary weak-irreducible terms instead of to values. Both the by-value and by-wnf policies coincide in the context of programming languages, where the input program is closed and the bodies of abstractions are never reduced. However, the reduction theory induced by the by-wnf policy presents the problems with confluence described in (Plotkin, 1975).

The following are examples of strategies implementing the by-wnf policy. Paulson (1996) introduces `eval` and `byValue`, which are weak- and full-reducing by-wnf strategies respectively. Paolini & Ronchi Della Rocca (1999) distinguishes the *inner machine* and the *ahead machine*, where the former coincides with Paulson's `eval`, and the latter is the analogous of head reduction (Barendregt, 1984) in the by-wnf setting. Grégoire & Leroy (2002) introduce *strong normalisation*, which is a by-wnf normalisation-by-evaluation approach that uses the Zinc Abstract Machine (Leroy, 1991) as the eval weak-reducing stage. Strong normalisation is equivalent to Paulson's `byValue` strategy.

1.3 Formal semantics and inter-derivation of semantic artefacts

The meaning of a program can be given by a mathematical device (an interpreter, an inference system, a model, a logic...) which underlies the programming language. Formal semantics of programming languages, which concerns the study of such mathematical

[4]This version of the calculus is known in the literature as the *lazy lambda-value calculus* (Egidi *et al.*, 1991, 1992), where 'lazy' is used only in the sense of 'weak-reducing', losing the sense of 'non-strict' that is commonly associated with the word 'lazy'.

devices, usually comes in one of the following three styles.

An *operational semantics* provides the meaning of a program by specifying how the program is to be evaluated or executed. An operational semantics is founded on a reduction strategy that specifies the order in which redices must be contracted. Section 2.2 provides a thorough description of the different formalisms that define an operational semantics.

A *denotational semantics* provides meaning by identifying a mathematical structure (*i.e.*, a *model*) and assigning each program to an object (*i.e.*, a *denotation*) in that model. The denotation of a program is constructed after the denotations of its sub-programs by a *semantic function*. Section 2.3 comments on the properties of Scott's domain D_∞ (Scott, 1970) which is the first known lattice model for the lambda calculus.

An *axiomatic semantics* provides logical assertions for each of the language constructs, such that the combination of the assertions corresponding to a particular program characterises the meaning of that program in any of the possible models for the programming language. This thesis does not consider axiomatic semantics.

Different formalisms for operational semantics exist that can be implemented as programs. Such programs are called 'semantic artefacts', a terminology perhaps coined for mathematical descriptions of semantics but often used by extension to their implementations (Danvy, 2006a). The semantic artefacts can be derived by means of program transformation. The derivation techniques were pioneered by (Reynolds, 1998), where a 'definitional interpreter' for a 'simple applicative language' is transformed into continuation passing style (CPS) in order to make the semantics of the target language independent from the semantics of the implementation language, and the interpreter is later defunctionalised in order to remove the higher-order functions from it. These techniques were later collected and extended by Olivier Danvy and his collaborators (Danvy, 2006b, 2008a; Danvy *et al.*, 2011). The CPS transformation and the defunctionalisation constitute the core of the *functional correspondence* (Ager *et al.*, 2003b; Danvy, 2006b) which connects a natural semantics (Kahn, 1987) with an abstract machine (Landin, 1964). On a related note, refocusing (Danvy & Nielsen, 2004) and lightweight fusion by fixed-point promotion (Ohori & Sasano, 2007; Danvy & Millikin, 2008) allow to connect a reduction semantics (Felleisen, 1987) with an abstract machine through a *syntactic correspondence* (Danvy, 2005; Biernacka & Danvy, 2007; Danvy, 2008b). The functional and syntactic correspondences together are referred to as 'inter-derivation techniques', with the particular meaning that a reduction semantics and a natural semantics inter-derive to each other when the abstract machine derived from both is the same. The CPS transformation and defunctionalisation (Danvy & Millikin, 2009; Danvy *et al.*, 2011) are also used to derive a reduction semantics from the search function that characterises a structural operational semantics (SOS) (Plotkin, 1981). Finally, the thunks (Danvy & Hatcliff, 1992; Hatcliff & Danvy, 1997) that appear in a semantic function characteristic of a denotational semantics can be closure-converted (*i.e.*, defunctionalised) to obtain an environment-based natural semantics (Ager *et al.*, 2003b; Danvy, 2008a). (Section 2.2 provides a thorough description of the different formalisms for operational semantics and of the inter-derivation techniques, and Section 1.5 comments about the closures and the environment technique.)

In (Munk, 2008; Danvy *et al.*, 2013) the 2-layer CPS (2CPS) and the concomitant 2CPS transformation (Danvy & Filinsky, 1990; Biernacka *et al.*, 2005; Danvy, 2006a) are used to inter-derive semantic artefacts for full-reducing strategies in the lambda calculus. Munk (2008) connects the environment-based layered reduction semantics of full-reducing strategies to the different full-reducing abstract machines in (Curien, 1993; Crégut, 2007; Kluge, 2010). However, the inter-derivations in (Munk, 2008) are not detailed (*i.e.*, the code with all the intermediate artefacts is not available) and the 2CPS transformation increases the complexity when reasoning about the different artefacts obtained.

1.4 Nameful and name-free representations of the lambda calculus

A binding occurrence of a variable λx specifies that the applied occurrences of x in the subsequent body are abstracted. This nameful representation entails an equivalence relation that equates all the terms differing only on the names of their bound variables. The equivalence is induced by the α-conversion rule, $M \to_\alpha M'$, which rewrites a subterm $\lambda x.N$ of M into $\lambda y.[x/y]N$ of M', where the y does not appear at all in N (Barendregt, 1984). To avoid issues with the capture of free variables when placing a term into some context, the *Barendregt convention* (Barendregt, 1984) is usually assumed. The Barendregt convention specifies that the free variables of the set of terms $M_1, M_2, \ldots$ in a mathematical context (a definition, a proof, *etc.*) is always different from the bound variables of that set of terms, *i.e.*, the required α-conversion is always performed in order to avoid capture of free variables. This is *as is* the terms where considered up to α-equivalence.

An alternative solution is to use a name-free representation which maps the linear order of the nested binding occurrences to the natural numbers, and encodes each applied occurrence as a natural number. This name-free representation, which is due to (De Bruijn, 1978), comes in one of the following two styles.

The *de Bruijn indices* represent applied occurrences as the relative distance to the binding occurrence, with 0 the closest binding occurrence (*e.g.*, nameful $\lambda x.(\lambda y.y\,x)x$ is written as the name-free $\lambda.(\lambda.0\,1)0$).

The *de Bruijn levels* represent applied occurrences as the absolute nesting level of the binding occurrence, with 1 the outermost binding occurrence (*e.g.*, nameful $\lambda x.(\lambda y.y\,x)x$ is written as the name-free $\lambda.(\lambda.2\,1)1$).

The index (resp. level) for the closest (resp. outermost) binding occurrence is chosen by convention. The convention here will simplify the calculations when using both indices and levels together in Chapter 6. Index 0 corresponds to the closer binding occurrence, whose binding is stored in the first position in the environment (*i.e.*, zero is conventionally the index for the first position when accessing arrays and other data structures, which simplifies index arithmetic). Level 0 corresponds to the nesting at the root level (*i.e.*, no nesting at all), and thus level 1 corresponds to the level when crossing the first lambda (*i.e.*, the outermost binding occurrence).

1.5 Explicit substitutions and the environment technique

Explicit substitutions provide an intermediate step between the formal specifications of lambda calculi and its concrete implementations (Abadi *et al.*, 1991; Curien, 1991; Lescanne, 1994; Hardin *et al.*, 1998; Kesner, 2007; Biernacka & Danvy, 2007). Conventionally, the contraction rules of a reduction system are specified by means of the external substitution function $[\_/\_]\_$ which replaces a term for the free occurrences of a variable in another term. Explicit substitutions dispose of the external substitution function by incorporating the notion of substitution as a syntactical construct, and by providing contraction rules that distribute this syntactical construct into the term structure and progressively carry out the substitution. The implementations of explicit substitutions are akin to the *environment technique*, which was introduced by (Landin, 1964) in computer science and by (Scholz & Hasenjaeger, 1961, §54) in logic. A term with explicit substitutions makes up a *closure*, which consists of a term and an environment that carries the bindings of the free variables in the term.

Curien (1991) implements explicit substitutions by using the de Bruijn indices representation for variables (De Bruijn, 1978) such that an index n points to its binding in an environment ρ consisting of a list of terms. Curien turns the β-rule into the contraction rules:

$$\frac{M[\rho] \to^* (\lambda.B)[\rho']}{(M\,N)[\rho] \to B[N[\rho] : \rho']}\ (\textsc{Eval}) \qquad \frac{}{n[\rho] \to n^{\text{th}}(\rho)}\ (\textsc{Var})$$

Contraction is split into the creation of a new explicit substitution (*i.e.*, pushing a binding on the environment) and the deferred replacement of the new binding for the formal parameter in the body of the abstraction (*i.e.*, looking up the formal parameter on the environment). The following reduction sequence illustrates:

$$((\lambda.0)N)[\rho] \to 0[N[\rho] : \rho] \to N[\rho] \to \ldots$$

Curien's calculus simulates weak reduction in lambda calculus. However, the rules above do not constitute a truly reduction semantics since rule Eval is not a contraction rule proper. Although Eval looks superficially like an inference rule, the premiss $M[\rho] \to^*_\rho (\lambda x.B)[\rho']$ is not a judgement of the reduction theory, and hence it does not correspond to a derivation of the inference system. In other words, rules Eval and Var do not constitute a structural operational semantics proper (Plotkin, 1981) (recall the structural operational semantics (SOS) formalism of Section 1.3, which is further described in Section 2.2.) The premiss of Eval consists of the concatenation of multiple reduction steps. These steps have to be disentangled from the contraction of the outermost redex $(\lambda.B)\,N$, which is the elementary step that rule Eval should stand for. Biernacka & Danvy (2007) fix this by adding a syntactical construct for *closure application*, in the tradition of eval/apply interpreters for programming languages (Abelson *et al.*, 1985). The new construct allows the definition of a SOS proper, since the context of both the operator

and operand in a closure can be navigated. The contraction rules in (Biernacka & Danvy, 2007) read

$$\frac{}{(\lambda.B)[\rho] \cdot \mathsf{N} \to B[\mathsf{N} : \rho]}\ (\beta) \qquad \frac{}{n[\rho] \to n^{\text{th}}(\rho)}\ (\textsc{Var})$$

$$\frac{}{(M\,N)[\rho] \to M[\rho] \cdot N[\rho]}\ (\textsc{App})$$

where sans-serif fonts M stand for closures, and closure application is written $\mathsf{M} \cdot \mathsf{N}$. Rule (β) is the conventional β-rule lifted to closure applications, and Var is the look up rule as before. The new App rule expands an application closure into a closure application. A SOS for the call-by-name strategy could be easily defined by adding the compatibility rule

$$\frac{\mathsf{M} \to \mathsf{M}'}{\mathsf{M} \cdot \mathsf{N} \to \mathsf{M}' \cdot \mathsf{N}}\ (\mu)$$

which specifies that closure operators have to be navigated when looking for redices. This SOS behaves in the same way than rules Eval and Var before, except for the application expansion step of rule App. The following reduction sequence illustrates:

$$((\lambda.0)N)[\rho] \to (\lambda.0)[\rho] \cdot N[\rho] \to 0[N[\rho] : \rho] \to N[\rho] \to \dots$$

The calculus in (Biernacka & Danvy, 2007) simulates the one in (Curien, 1991), and is therefore suitable for weak reduction in a setting with explicit substitutions.

1.6 The problem

One of the motivations of (Sestoft, 2002) was that varying definitions for the conventional call-by-name and call-by-value strategies existed in the literature (Plotkin, 1975; Felleisen & Hieb, 1992; Paulson, 1996), and the relation between programming languages and reduction systems was unclear. Sestoft did clarify the aforementioned relation to some extent, but the situation for the full-reducing strategies deserves more attention. Different formalisms exist (SOS, reduction semantics, abstract machines, natural semantics) which are presented using different styles (hybrid single functions, NBE). Some strategies *are altered unwittingly* when altering their presentation style. For instance, Sestoft (2002) claims that his hybrid applicative order is equivalent to function `byValue` in NBE style (Paulson, 1996), but actually it is not. Other strategies are defined combining different formalisms together, and it is unclear how to connect them to homogeneous definitions. For instance, strong reduction (Grégoire & Leroy, 2002) uses a NBE approach where the eval stage is presented as a context-based reduction semantics, and the readback stage is defined in an equational way which is akin to a natural semantics. Function `byValue` (Paulson, 1996) also uses the NBE approach, where both the eval and the readback stages are a straightforward translation of natural semantics into ML code. Strong reduction and `byValue` define the same strategy, but the connection between them is far from being obvious.

A significant work has been done to connect reduction semantics with full-reducing abstract machines (Ager *et al.*, 2003a; Munk, 2008; Danvy *et al.*, 2013). However, the latter are usually defined in environment-based style, and the correspondence of these to the full-reducing strategies in the traditional substitution-based style is not formally shown.

There is even more confusion regarding eager semantics. The by-value policy is usually mixed up with the by-wnf policy (see Section 1.2). Both policies are equivalent in a weak-reducing setting (*i.e.*, closed terms, not going 'under lambda'), but they differ in a full-reducing setting. The semantics which coincide with the by-wnf policy have received more attention (Paulson, 1996; Paolini & Ronchi Della Rocca, 1999; Sestoft, 2002; Grégoire & Leroy, 2002; Accattoli & Paolini, 2012), with the notable exception of the principal reduction machine of (Ronchi Della Rocca & Paolini, 2004) which truly implements[5] the original by-value policy in (Plotkin, 1975). However, (Ronchi Della Rocca & Paolini, 2004) brings up a bigger problem. Although the meta-theory for classical lambda calculus is well understood (Scott, 1970; Scott & Strachey, 1971; Wadsworth, 1976; Barendregt, 1984; Barendregt *et al.*, 1987), the analogous for the lambda-value calculus is controverted. According to the tradition in (Egidi *et al.*, 1991, 1992; Paolini & Ronchi Della Rocca, 1999; Ronchi Della Rocca & Paolini, 2004; Accattoli & Paolini, 2012), the full normal forms in lambda-value, and hence the by-value full-reducing operational semantics, are considered uninteresting upfront. This is because of an emphasis in weak-reducing machines in the context of programming languages, and because of the confusion between the by-value and by-wnf policies.

The recurrent hybrid (or layered) character which is present in the full-reducing (and some other) strategies has been acknowledged in the folklore (see Section 1.1). However, the hybrid character is often considered as a matter of *style* (*i.e.*, connected to a particular formalism or representation). Whether the hybrid character is a matter of *nature* (*i.e.*, intrinsic to the strategy itself) is still unclear. No formal characterisation of either style or nature exists.

Some full-reducing environment-based operational semantics have been inter-derived (Ager *et al.*, 2003a; Danvy *et al.*, 2013; Munk, 2008). The hybrid character of the underlying strategies has an impact on the complexity of the inter-derivation. The NBE approach is used and the machines obtained present either *two* control stacks or a *single* control stack which has a *layered* structure (*i.e.*, implemented by two datatypes, one depending on the other). In both cases there are different configurations (*i.e.*, states) dispatching on each of the control stacks (resp. datatypes for control stacks). In (Ager *et al.*, 2003a) a virtual machine[6] with a single layered control stack is derived, but only the functional correspondence is shown and the detailed inter-derivation (the code implementing the intermediate artefacts) is omitted. In (Munk, 2008; Danvy *et al.*, 2013) an abstract machine with two

[5]The machine in (Ronchi Della Rocca & Paolini, 2004) is parametric in the set Δ and in the Δ-normal forms. To implement the by-value policy Δ has to be instantiated to *values*, and the Δ-normal forms to *lambda-value normal forms*.

[6]A *virtual machine* has an instruction set and requires a compilation stage. An *abstract machine* operates directly on terms and does not need a compiler.

control stacks is inter-derived, and both the functional and the syntactic correspondences are shown. (Yet the code implementing the intermediate artefacts is also omitted.) Munk (2008) considers up to three derivation paths for the syntactic correspondence. In the first path, the full-reducing machine of (Curien, 1993) is connected to a 2CPS environment-based reduction semantics. In the second path, this very machine is connected to a plain CPS (*i.e.*, 1-layer CPS) environment-based reduction semantics with control delimiters. In the third path, the full-reducing Krivine machine of (Crégut, 2007) (this machine is almost equivalent to the HOR machine of (Kluge, 2010)) is disentangled to match the machine in the other two paths, which can be subsequently connected to either of the 2CPS reduction semantics or the plain CPS reduction semantics with control delimiters. (The work of (Danvy *et al.*, 2013) only collects the two first paths in (Munk, 2008).) The eval/readback phase distinction in NBE entails, alternatively, intermediate artefacts in 2CPS or the use of control delimiters. None of the paths shows how to obtain artefacts with plain CPS and without control delimiters, since those two features seem to clash. Both features are desirable for the endeavour of 'mechanising' the inter-derivation techniques.

There is an outgrowing number of operational semantics with a layered character, where a guest semantics is plugged into a host semantics. A recent example is the gradually-typed lambda calculus (Henglein, 1994; Siek & Taha, 2006; Siek *et al.*, 2009; Siek & Garcia, 2012; Garcia, 2013), where a family of coercion calculi can be plugged into a simply-typed lambda calculus. Interpretations of gradually-typed lambda calculus exist (Siek & Garcia, 2012), but the correctness of the interpreter—for each choice of coercion semantics—has been only conjectured. The inter-derivation techniques would provide a constructive proof of correctness, but there is the additional issue of making the inter-derivation modular on the coercion semantics (*i.e.*, the different inter-derivations for the choices of coercion semantics could be plugged in the inter-derivation of the host calculus, which would be reused).

1.7 Contributions

The contributions of this thesis can be classified according to their theoretical or practical character. Chapter 3 and 4 deals with (most of) the theoretical contributions, and Chapters 5, 6, and 7 with the practical contributions. All they are detailed in what follows.

1.7.1 Theoretical contributions

Hybrid nature and the beta cube

Chapter 3 provides a rationale for the space of strategies unveiled by (Sestoft, 2002). A definition of the hybrid/uniform nature of a strategy, which gives formal grounds to the informal terminology in (Sestoft, 2002), is deferred to Section 5.3:

- Section 3.2 introduces a big-step-style template for reduction strategies that can be instantiated to the foremost strategies in the literature and more. The template is implemented in Haskell as a higher-order monadic function whose fixed points are

reduction strategies. The resulting code is readable, clean, and abstracts away from the machinery required to guarantee semantics preservation for all strategies in lazy Haskell.

- Section 3.3 introduces a lattice of reduction strategies which is referred to as the β-cube. The lattice is obtained by interpreting some parameters of the big-step template as boolean switches. The β-cube captures neatly and systematically many reduction strategies of the pure lambda calculus. It contains eight uniform strategies: applicative order, call-by-value, call-by-name, head spine order, and four new ones.

- Section 3.4 defines a hybridisation function that generates up to ten hybrid strategies by appropriately composing a base and a subsidiary strategy taken from the cube. The hybrid strategy is a single function that depends on the subsidiary, and the subsidiary is a uniform strategy. Hybrids are paramount for full reduction. In particular, the full-reducing strategies that uphold completeness under the calling policy that they implement, are hybrid.

- Section 3.8 proves that for each hybrid strategy generated from base and subsidiary from the cube, there exist a pair eval/readback functions in the NBE approach corresponding to the hybrid. The subsidiary coincides with the eval stage. This correspondence helps to prove an absorption theorem which states that a subsidiary strategy is a right identity of any hybrid that depends on the subsidiary.

Towards a standard theory for the lambda-value calculus

Chapter 4 develops the meta-theory of lambda-value calculus for the endeavour of establishing a 'standard theory' which is the natural analogous of the 'standard theory' of the classical lambda calculus (Scott, 1970; Scott & Strachey, 1971; Wadsworth, 1976; Barendregt, 1984; Barendregt *et al.*, 1987):

- Section 4.3 introduces the syntactic notion of quasi-v-solvability, which aims at reinstating the validity of lambda-value normal forms (Egidi *et al.*, 1991, 1992; Paolini & Ronchi Della Rocca, 1999; Ronchi Della Rocca & Paolini, 2004; Accattoli & Paolini, 2012) and to develop on the meta-theory of lambda-value. The new definition, which differs from that in (Paolini & Ronchi Della Rocca, 1999), captures 'preservation of unsolvables' and 'genericity lemma' (Wadsworth, 1976). Similar to (Abramsky, 1990; Paolini & Ronchi Della Rocca, 1999), an unsolvable is qualified with an order which indicates the maximum number of trailing lambdas of any term which is β_V-equivalent to the unsolvable. Quasi-v-solvability rests on the notion of lambda-value needed reduction, which is a generalisation of the needed reduction in classical lambda calculus (Barendregt *et al.*, 1987). Lambda-value needed reduction captures effective use in lambda-value.

- Section 4.5 introduces the theory $\mathcal{H}_V$ which equates all the unsolvables of equal order and proves that $\mathcal{H}_V$ is a consistent extension of the lambda-value theory. The

concomitant notion of ω-sensibility (*i.e.*, satisfying $\mathcal{H}_V$) aims at characterising the theories and models in lambda-value with good operational properties.

- Section 4.7 characterises all complete strategies in lambda-value. Theorem 4.8.7 broadens the Standardisation Theorem in (Plotkin, 1975) showing that *there are complete strategies in lambda-value which fail to be standard.* The novel value spine order strategy is introduced, which traverses the spine of the term in an way analogous to head spine and hybrid normal order (Barendregt *et al.*, 1987; Sestoft, 2002). Value spine order is the most eager strategy of lambda-value that is complete. Value spine order is a hybrid strategy that depends on two subsidiaries, of which, one is in turn a hybrid that also depends on the other subsidiary.

1.7.2 Practical contributions

Inter-deriving hybrid strategies

Chapter 5 shows how to apply the techniques for inter-derivation of semantic artefacts (Ager *et al.*, 2003b; Danvy, 2005, 2006b, 2008a; Danvy & Millikin, 2008; Danvy *et al.*, 2011) to the case of hybrid strategies. The solution is showcased by inter-deriving semantics for normal order, the standard, full-reducing, and complete strategy of the classical lambda calculus. This solution, which is an alternative to the use of 2CPS and control delimiters in (Munk, 2008; Danvy *et al.*, 2013), relies on the shape invariant of the strategy's reduction contexts to obtain a first-order abstract machine with shallow inspection (*i.e.*, in defunctionalised form) and with only one control stack. The intermediate artefacts in the inter-derivation are in plain CPS and do not require the use of control delimiters:

- Section 5.2 presents SOS, reduction semantics, and natural semantics of normal order defined by a *single* hybrid function, as opposed to the composition of *two* functions in the NBE approach.

- Section 5.3 introduces a formal definition of the hybrid/uniform nature of a strategy. The hybrid/uniform character is not an accidental property (*i.e.*, akin to a *style* of presentation) but intrinsic to the strategy (*i.e.*, determined by the strategy's *nature*). A strategy is uniform if inclusion of contexts is necessary and sufficient to characterise all the reduction contexts of the strategy. Otherwise, the strategy is hybrid.

- Section 5.5 obtains the grammar of well-formed continuation stacks and proves its shape invariant, which will be used latter to enforce the shallow inspection of the abstract machine.

- Section 5.6.2 shows how the intensional refocusing function for the strategy is context-dependent, *i.e.*, the function inspects the current continuation in order to decide which normalising function (subsidiary or hybrid) is to resume.

- Section 5.6.6 uses the shape invariant of the reduction contexts to recover shallow inspection of the abstract machine that is arrived to by applying the standard inter-derivation steps to the semantics with context-dependent refocusing function. The obtained abstract machine is in defunctionalised form.

- As further evidence of the applicability of this solution, Section 5.9 discusses the inter-derivation of many hybrid strategies in the literature, including head reduction (Barendregt, 1984), strong reduction (Grégoire & Leroy, 2002) and `byValue` (Paulson, 1996), hybrid applicative order (Sestoft, 2002), the ahead machine (Paolini & Ronchi Della Rocca, 1999), and the outermost strategy for arithmetic expressions (Danvy & Johannsen, 2013). All the strategies posses layered reduction contexts where the appropriate shape invariant can be identified.

It is the experience of the author and his collaborators that plain CPS without control delimiters makes the inter-derivation more amenable to automation, and could help for the endeavour of 'mechanising' the inter-derivation.

Closure calculus for full reduction

Chapter 6 derives by program transformation the full-reducing Krivine machine of (Crégut, 2007) from the SOS of normal order. This derivation, which is an alternative to the derivation in (Munk, 2008), unveils the calculus of closures underneath the full-reducing Krivine machine (Crégut, 2007). The SOS definition of normal order in this calculus enjoys index alignment and balanced derivations, which are key properties to reason locally in a scope where the bindings of free variables are not available. The full-reducing Krivine machine is arrived from a single-stage hybrid SOS by constructing the grammar of well-formed continuation stacks and observing its shape invariant:

- Section 6.7 introduces the calculus of closures $\lambda_{\widetilde{\rho}}$ which naturally extends the λ_{ρ} calculus of (Curien, 1991) and the $\lambda_{\widehat{\rho}}$ calculus of (Biernacka & Danvy, 2007). The $\lambda_{\widetilde{\rho}}$ calculus adds de Bruijn levels, closure abstractions, and absolute indices, which are required for full-reduction. The de Bruijn indices represent the formal parameter of an unapplied abstraction when its body is being evaluated, a technique dubbed 'parameter as levels' which is akin to the full-reducing Krivine Machine of (Crégut, 2007). Closure abstractions are required to represent closures where the redex may occur under lambda, and absolute indices are required to represent 'neutral closures', *i.e.*, non-redex closure applications (see Section 6.7.1.

- Section 6.7.1 defines the SOS of normal order in $\lambda_{\widetilde{\rho}}$, and proves that normal order reduction in $\lambda_{\widetilde{\rho}}$ mirrors *stepwise* normal order reduction in the pure lambda calculus. The SOS enjoy *index-alignment* and *balanced derivations*. This features allows to reason over the SOS, solving the paramount issue with binders of reasoning locally in an scope where the binding of a free variable is not available (Aydemir *et al.*, 2008).

- Section 6.9.2 adds a non-standard but straightforward 'preponing' step to the inter-derivation, which is needed for shortcut optimisation. The preponing step is proved to be equivalence-preserving. The preponed semantics obtained after this step resembles the environment-based semantics in (Munk, 2008). In this thesis, the preponed semantics are arrived to by program transformation, instead of contrived, as is the case in (Munk, 2008).

- Section 6.10.1 and 6.10.3 show how to use the correlation of certain constructors on the top of the well-formed continuation stack with some explicit control that has to be introduced in one of the intermediate artefacts. This is another application of constructing the grammar of well-formed continuation stacks.

- Section 6.10.4 arrives to a machine which is a slightly optimised version of (Crégut, 2007) that can also work with *open* terms and that does not need to carry lambda levels in ground terms. These features are also present in the HOR machine of Kluge (2010).

Deriving interpretations for the gradually-typed lambda calculus

Chapter 7 shows how to combine hybrid semantics and 2CPS to obtain an inter-derivation which is modular in the coercion semantics. 'Modular' means that different inter-derivations for different choices of coercion semantics can be plugged in the overall inter-derivation of the host semantics:

- Section 7.2 introduces a core calculus for gradual-typing which unify and slightly amends and emends the calculi in (Siek & Garcia, 2012; Siek *et al.*, 2009) so as to have an *implementable* reduction semantics satisfying unique-decomposition (Felleisen, 1987).

- Section 7.4 translates the definitional interpreter in (Siek & Garcia, 2012) to ML and derive an instantiation for a coercion-based eager-downcast dynamic semantics (Siek *et al.*, 2009).

- Section 7.8 closure-converts the core calculus and introduces a simply-typed lambda calculus of closures with explicit casts. The reduction semantics of the calculus of closures is transformed into the instantiation of the definitional interpreter.

The small-step and big-step artefacts for expressions with coercion casts are parametric on the artefacts for coercions. Thanks to layering and 2CPS the artefacts for coercions can be replaced by other artefacts implementing different dynamic semantics. This technique provides the basis for modular derivations of any hybrid semantics, not limited to the definitional interpreters of the gradually-typed lambda calculus.

1.8 Overview

Chapter 2 introduces some technical preliminaries. The rest of the thesis builds on a collection of five papers[7] that correspond roughly to Chapters 3 to 7. The collection is structured in three parts. The first part consists of Chapter 3 (The Beta Cube) and Chapter 4 (Towards a Standard Theory for The Lambda-Value Calculus), which deal with the hybrid and full-reducing strategies in the classical lambda calculus, and with the meta-theory for full reduction in the lambda-value calculus. The second part consists of Chapter 5 (Interderiving Hybrid Normalisers) and Chapter 6 (Deriving the Full-Reducing Krivine Machine), which concern efficient implementations of full-reducing machines in substitution-based and closure-based styles respectively. The third part consists of Chapter 7 (Deriving Interpretations of the Gradually-Typed Lambda Calculus), where the lessons about hybrid strategies that were learnt in Parts I and II are applied to the interpretations of the gradually-typed lambda calculus. Chapter 8 concludes.

The collection of papers contains the published versions or, for those papers whose extended versions are still unpublished, a stable version which has been considered for submission. However, these stable versions may not contain some of the contents which are still in progress. An addendum may be included at the end of the chapters which drafts and outlines the work in progress. The reader is warned to read these addenda for the most updated material.

Each chapter in the collection of papers is self-contained, in that a section with conclusions and future work is included when appropriate. For the convenience of the reader, Chapter 8 provides general conclusions and summarises the most relevant discussions.

[7] Contrary to this chapter and to Chapter 2, where the impersonal voice is used, the collection of papers uses the active voice which was originally present in the papers.

2
Technical Preliminaries

2.1 Lambda calculus

This thesis focuses on the pure untyped lambda calculus and in its strict-functional-semantics counterpart, *i.e.*, the pure versions of the call-by-name and call-by-value calculi in (Plotkin, 1975) respectively. The pure version of the call-by-name calculus coincides with the classical lambda calculus in (Church, 1936; Barendregt, 1984). The pure version of the call-by-value calculus is described in (Egidi *et al.*, 1991, 1992; Paolini & Ronchi Della Rocca, 1999; Ronchi Della Rocca & Paolini, 2004). Both calculi shall be referred to generically as lambda calculus or λ-calculus. When referring to the call-by-name version, the epithet 'classical', or the abbreviation λK, may be used.[1] When referring to the call-by-value version, the qualifier '-value', or the abbreviation λ_V, may be used.

The traditional syntax of lambda terms is specified by the pseudo-grammar $\Lambda ::= x \mid (\lambda x.\Lambda) \mid (\Lambda\,\Lambda)$, where x, y, etc, range over the elements of a countably infinite set of variables. In words (to refresh terminology), lambda terms consist of *variables*, of *abstractions* (consisting of a bound variable and the abstraction's *body*), and of *applications* of an *operator* to an *operand*. For example, $((\lambda x.x)y)$ is the identity abstraction applied to variable y. The abstraction $(\lambda x.x)$ is the operator and the variable y is the operand. Symbol Λ is overloaded to stand for a grammatical non-terminal and for the set of lambda terms. (The subset containing the *closed* lambda terms is written Λ^0.) Uppercase, sometimes primed, letters M, N, B, M', etc, range over elements of Λ. Parenthesis are dropped according to the standard precedence and association conventions: applications associate to the left and application binds tighter than abstraction. Hence, the term above is written $(\lambda x.x)y$.

The reader must be familiar with the usual definitions of bound and free variables, of syntactic equality of terms modulo renaming of bound variables (written $\equiv$), of capture-

[1] The classical lambda calculus is sometimes referred to as λK to distinguish it from the λI version, see (Barendregt, 1984, Chapters 2 and 9).

$$\frac{}{(\lambda x.B)N \to_\beta [N/x]B}\ (\beta) \qquad \frac{N \in \mathsf{Val}}{(\lambda x.B)N \to_{\beta_V} [N/x]B}\ (\beta_V)$$

$$\frac{M \to_\beta M'}{M\,N \to_\beta M'\,N}\ (\mu) \qquad \frac{M \to_{\beta_V} M'}{M\,N \to_{\beta_V} M'\,N}\ (\mu_V)$$

$$\frac{N \to_\beta N'}{M\,N \to_\beta M\,N'}\ (\nu) \qquad \frac{N \to_{\beta_V} N'}{M\,N \to_{\beta_V} M\,N'}\ (\nu_V)$$

$$\frac{B \to_\beta B'}{\lambda x.B \to_\beta \lambda x.B'}\ (\xi) \qquad \frac{B \to_{\beta_V} B'}{\lambda x.B \to_{\beta_V} \lambda x.B'}\ (\xi_V)$$

$$\mathsf{Val} \ ::= \ x \mid (\lambda x.B)$$

Figure 2.1: Single-step reduction relations $\to_\beta$ and $\to_{\beta_V}$

avoiding substitution $[N/x]B$ (which reads substitute N for the free occurrences of x in B), and of contexts $\mathbf{C}[\ ]$ (terms with holes).

Relations are defined as Hilbert-style inference systems (*i.e.*, a set of inference rules with antecedent and consequent separated by a bar, where the axioms are the rules with an empty antecedent). Figure 2.1 defines the single-step reduction relations in λK and in λ_V (denoted $\to_\beta$ and $\to_{\beta_V}$ respectively) which are the compatible closure of contraction rules (β) and (β_V). The β-redices are of the form $(\lambda x.B)N$, and the β_V-redices are similar but with N a *value* (*i.e.*, N in the set Val defined in the bottom of Figure 2.1). Multiple-step reduction relations ($\to^*_\beta$ and $\to^*_{\beta_V}$) and equivalence relations ($=_\beta$ and $=_{\beta_V}$) are respectively the reflexive and transitive closure, and the reflexive, symmetric and transitive closure of the single-step reduction relations.

Given a reduction system, a *reduction strategy* (or just strategy) of the system is a (partial) function which is a sub-relation of the system's multiple-step reduction. In the classical lambda calculus, a strategy is a sub-relation of $\to^*_\beta$, and in the lambda-value calculus, a strategy is a sub-relation of $\to^*_{\beta_V}$. Strategies can be presented in small-step or big-step fashion.[2] A small-step strategy is concerned with the next step of computation, *i.e.*, it maps a term to a subsequent term in the reduction sequence. A big-step strategy is concerned with final results, *i.e.*, it maps an input term to the last term in its reduction

[2]Do not mistake small-step with single-step, nor big-step with multiple-step. Both single-step and multiple-step are small-step, since the multiple-step relation does not exhaust the reduction relation, *i.e.*, reduction is not necessarily carried out until reaching an irreducible form. Through the text, 'single-' and 'multiple-' are used for relations, and 'small-' and 'big-' are used for strategies or semantics.

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

Figure 2.2: Irreducible forms in classical lambda calculus

sequence, or it is undefined if such a term does not exist. Both small- and big-step strategies are proper sub-relations of the system's multiple step reduction relation. Each small-step strategy corresponds to a unique big-step strategy, but not reciprocally. However, each *intensionally defined* big-step strategy in this thesis (*i.e.*, any strategy defined by the big-step operational semantics formalisms described in Section 2.2) does correspond to a unique small-step strategy. Therefore, in this thesis 'reduction strategy' is used to refer to the deterministic order of contraction for redices that lays underneath any of the strategy's small-step or big-step presentations. For a strategy s, its small-step and big-step presentations are written $M \to_s N$ and $M \Downarrow_s N$ respectively. Relational $M \Downarrow_s N$ and functional $\Downarrow_s (M) = N$ notation is used interchangeably, and function composition is used when appropriate, *e.g.*, $(\Downarrow_t \circ \Downarrow_s)(M) = \Downarrow_t (\Downarrow_s (M))$.

Figure 2.2 shows the grammar of normal forms (nf), weak normal forms (wnf), head normal forms (hnf), and weak head normal forms (whnf) respectively, which are conventional notions of irreducible terms in the classical lambda calculus. In the text, grammars are defined in Extended Backus-Naur Form. Sets of terms (or closures, *etc.*) are written as non-terminal identifiers in uppercase sans-serif (*i.e.*, NF).[3] Sets of contexts (terms with a hole) are written as non-terminal identifiers in uppercase bold-roman, followed by the hole symbol (*i.e.*, $\mathbf{C}[\,]$). The regular expression $\{\mathsf{NF}\}^*$ in the first line of Figure 2.2 stands for zero or more occurrences of NF. The sentential forms x, $x\,\mathsf{NF}$, $x\,\mathsf{NF}\,\mathsf{NF}$, etc, are derivable from production $x\,\{\mathsf{NF}\}^*$ and respectively associate (according to the convention) as x, $(x\,\mathsf{NF})$, $((x\,\mathsf{NF})\,\mathsf{NF})$, etc.

Through the text, the following abbreviations for terms are used. The term $I \equiv \lambda x.x$ denotes the identity function. The term $K \equiv \lambda x.\lambda y.x$ denotes the constant function (*i.e.*, a function that takes two arguments, and returns always the first argument, regardless of the second argument). The term $\Delta \equiv \lambda x.x\,x$ denotes the self-application operator (*i.e.*, a function that takes an argument and applies it to itself). The application of Δ to itself gives $\Omega \equiv (\lambda x.x\,x)(\lambda x.x\,x)$, which is a divergent term (it reduces forever) that denotes undefined.

A standard reduction sequence (Curry & Feys, 1958) is a reduction sequence where in each step the leftmost-outermost redex is contracted ('leftmost' in their terminology). The standardisation theorem states that if a term has a normal form, there is a standard reduction sequence ending in it. 'Standard reduction up to normal form' is akin to 'normal

[3] In Chapter 6 we contravene this convention and use uppercase roman for stacks (*i.e.*, S).

order'. The standardisation theorem states that normal order is a complete strategy, *i.e.*, it always delivers a normal form if the input term has some. However, 'standard' does not imply 'full-reducing'. For instance, call-by-name (Plotkin, 1975; Barendregt, 1984) is the standard strategy that reduce up to weak head normal form, and head reduction (Barendregt, 1984) is the one that reduces up to head normal form.

The 'spine' qualifier takes the anatomic metaphor of the 'head' qualifier further and refers to strategies that contract the redices in the left front of the abstract syntax tree of a term (Barendregt *et al.*, 1987). The spine strategies give decidable approximations to the set of *needed redices* of a term (Barendregt *et al.*, 1987) (See Section 4.11. Head spine order and hybrid normal order (Sestoft, 2002) are the spine counterparts of the standard strategies head reduction and normal order. Some head and spine strategies share many features with the full-reducing strategies, in particular they have hybrid nature, and hence they are also studied in this thesis. The spine strategies of the lambda-value calculus are introduced in Chapter 4.

2.2 Operational semantics and inter-derivation

The formalisms for operational semantics can be classified according to their small-step or big-step character. Traditional approaches to small-step operational semantics are structural (Plotkin, 1981), context-based (or reduction semantics) (Felleisen, 1987), and abstract machines (Landin, 1964). The latter are state transition functions that, unlike virtual machines, operate directly on terms, have no instruction set, and no need for a compiler. Traditional approaches to big-step operational semantics are compositional evaluators (Danvy, 2008a),[4] natural semantics (Kahn, 1987), and big-step abstract machines. The latter are first-order tail-recursive presentations of state transition functions.

All the different formalisms for operational semantics can be implemented as programs (*i.e.*, semantic artefacts) that can be *inter-derived by means of program transformation.* 'Inter-derivation' of semantic artefacts is used in the literature in the specific sense that the artefacts derive by program transformation to the same (implementation of an) abstract machine. The following literature is assumed (Ager *et al.*, 2003b,a; Danvy & Nielsen, 2004; Danvy, 2005, 2008a; Danvy & Millikin, 2008; Biernacka & Danvy, 2007; Danvy *et al.*, 2011), the last reference an excellent tutorial introduction on which part of the presentation in this thesis is based.

[4]Some authors refer to this formalism as 'denotational semantics' (Danvy, 2008a), but it shall not be confused with a denotational semantics proper (Section 1.3). A compositional evaluator comprises only the operational part of a denotational semantics, namely the semantic function, and lacks the mathematical structure in which the denotations live. The compositional evaluator, which runs in an *implementation language*, translates and executes a program in the *target language* on the fly. Typically, abstractions in the target language are shallowly embedded as functions in the implementing language. The implementation language is assumed to have a well-behaved formal semantics whose details are hidden, which allows the designer to focus on the semantics of the target language.

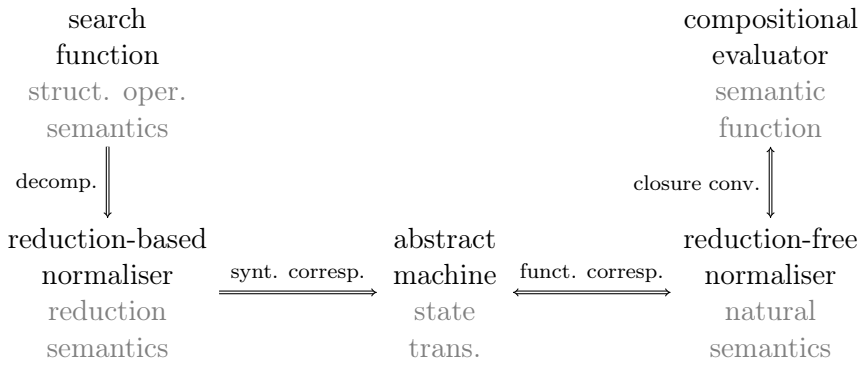


Figure 2.3: Semantic artefacts and derivation paths

Figure 2.3 illustrates the derivation paths among semantic artefacts. From left to right, a *search function* is a simple artefact that mirrors the compatibility rules of a structural operational semantics (the rules that express how to navigate a term to locate a redex). A search function delivers for an input term the redex subterm to be contracted or the input term back if the input term is irreducible. A search function derives to a *reduction-based normaliser* by applying the following program-transformation steps: CPS transformation, simplification, defunctionalisation, and turning the search into a decomposition function which, additionally to the next redex, delivers the context where the redex appears. A reduction-based normaliser is a program that implements a reduction semantics by iterating (i) the unique decomposition of a term into a context and a redex within the context hole, (ii) the contraction of the redex and, (iii) the recomposition of the resulting term. The additional recomposition function consists of a left fold over the contexts. A reduction-based normaliser derives to (a big-step implementation of) an *abstract machine* by applying the following steps: refocusing (which optimises the iteration loop), lightweight fusion by fixed-point promotion, and inlining-of-iterate-function steps. This latter derivation is called a *syntactic correspondence* and its steps are in general not reversible. The abstract machine derives to a *reduction-free normaliser* by applying refunctionalisation and direct-style transformation. This derivation is reversible by CPS transformation and defunctionalisation and is called a *functional correspondence*. A reduction-free normaliser is a program implementing a natural semantics, typically a recursive normaliser for deeply-embedded terms. A reduction-free normaliser derives (in a reversible way) to a *compositional evaluator* by applying closure conversion (closure unconversion respectively). A compositional evaluator is a *semantic function* which sends terms to *semantic values*. A semantic function uses the *environment technique* to store the semantic values that correspond to the free variables in the term. The input of the semantic function (*i.e.*, the term and the environment) constitutes an unfolded representation of closures, where the bindings are semantic values rather than closures themselves.

Reduction-based and reduction-free normalisers (and intermediate abstract machines) are equivalent because the transformation steps are equivalence-preserving. Consequently, the artefacts implement the same reduction strategy. A search function is a simpler artefact which, although not strictly equivalent, is sufficient to characterise the structural operational semantics (Danvy *et al.*, 2011). It connects the structural and context-based semantics by adding a recomposition function. A compositional evaluator is neither strictly equivalent to a reduction-free normaliser, since the former uses a domain of results with shallow-embedded abstractions (*i.e.*, meta-level functions) and the latter uses deep-embedded terms, of which the results (*i.e.*, irreducible forms) are just a subset. The compositional evaluator characterises the reduction strategy underneath the target language semantics by wiring it to the implementation language semantics.

Translating the target language into an implementation language with definite semantics raises the paramount issue, first noted in (Reynolds, 1998), of independence from the implementation language semantics, *i.e.*, implementing a strict semantics in a non-strict language, or the other way around. This issue pervades every big-step presentation of an operational semantics. To tackle it, this thesis either uses a monadic normaliser (Chapter 3) or fixes the implementation semantics to be strict (Chapters 5, 6, and 7) and implements non-strictness manually by using thunks (Danvy & Hatcliff, 1992).

2.3 Lambda theories and models

The operational and denotational approaches to semantics are akin, respectively, to the disciplines of *proof theory* and of *model theory* in logic. Proof theory is concerned with the syntactic manipulation of symbols in a formula, whereas model theory is concerned with the satisfiability of the formula in a particular mathematical structure. Broadly, a theory is the set of formulae that can be obtained by applying the rules of a formal system, and a model satisfies the formulae in the theory.

The symbol λ denotes the set of closed β-equations among lambda terms, which characterises the equivalence classes of programs induced by $=_\beta$. A *λ-theory* is a consistent extension of λ which considers an additional set of equations $M = N$ and its closure under β-equivalence. Extending λ is a salient procedure to give meaning to the calculus in a less-intensional way (Barendregt, 1984). For instance, it could be reasonable to identify all the terms that lack a normal form. However, such extension happens to identify any term with each other, rendering the resulting theory inconsistent (Barendregt, 1984). Consider $M_1 \equiv (\lambda x.x\, N_1\, \Omega)$ and $M_2 \equiv (\lambda x.x\, N_2\, \Omega)$, with arbitrary $N_1, N_2 \in \Lambda^0$. According to the extension, $M_1 = M_2$ because both lack a normal form, and hence $M_1\, K = M_2\, K$. However,

$$\begin{aligned} N_1 \;&=_\beta\; K\, N_1\, \Omega \;=_\beta\; (\lambda x.x\, N_1\, \Omega)K \;\equiv\; M_1\, K \\ &=\; M_2\, K \;\equiv\; (\lambda x.x\, N_2\, \Omega)K \;=_\beta\; K\, N_2\, \Omega \;=_\beta\; N_2 \end{aligned}$$

which violates consistency since it allows to equate any two arbitrary terms N_1 and N_2.

A better-behaved extension of λ is the one that identifies all the terms that lack a head normal form. The resulting theory, $\mathcal{H}$, is consistent and hence it is a λ-theory. Identifying

terms without head normal form follows from the *operational relevance* of terms, which is captured by the notion of *solvability*. A term T is *solvable* iff it could be *effectively used*[5] in a computation $\mathbf{C}[T]$ which delivers a normal form, *i.e.*, $\mathbf{C}[T] \twoheadrightarrow_\beta^* N \in \mathsf{NF}$ (Barendregt, 1972; Wadsworth, 1976; Barendregt, 1984; Paolini & Ronchi Della Rocca, 1999). Solvability is characterised syntactically as having a head normal form and thus all the terms without head normal form (*i.e.*, unsolvable terms) are operationally irrelevant, or meaningless. The Genericity Lemma states that if an unsolvable U takes part in a computation delivering a normal form, $\mathbf{C}[U] \twoheadrightarrow_\beta^* N \in \mathsf{NF}$, then the U could be replaced by any other term without affecting the result of the computation, *i.e.*, $\forall T.\mathbf{C}[T] \twoheadrightarrow_\beta^* N$ (Barendregt, 1984).

In the λK-calculus, any model has to meet the domain equation $D \cong [D \to D]$. The cardinality of the space of functions over a domain D is always greater than the cardinality of the domain D itself. Scott (1970) showed that the space of *continuous functions* over a *complete lattice* can be embedded in the lattice itself. He constructed the first lattice model for the λK-calculus, D_∞, which is defined as the inverse limit of a complete lattice D_0 where, inductively, D_{n+1} is taken as the space of continuous functions over D_n, *i.e.*, $D_{n+1} \cong [D_n \to D_n]$ (Scott, 1970; Scott & Strachey, 1971; Wadsworth, 1976; Stoy, 1979).

The study of λ-theories bridges proof theory with model theory in λK. The notion of *sensibility* characterises theories and models with good operational behaviour. A theory (or a model) is sensible iff it satisfies $\mathcal{H}$. There is a unique *Hilbert-Post completion*[6] of $\mathcal{H}$, namely $\mathcal{H}^*$, such that the local structure of D_∞ is *fully abstract* (Curien, 2007) with respect to it, *i.e.*, for any choice of a complete lattice D_0, $D_\infty \models M = N$ iff $\mathcal{H}^* \vdash M = N$ (Barendregt, 1984, p. 505).

The lambda-value calculus is also amenable to proof theory. The symbol λ_V denotes the set of closed β_V-equations among terms, which characterises the equivalence classes of programs induced by $=_{\beta_V}$. In Chapter 4 the theory $\mathcal{H}_V$ is identified, which is an extension of λ_V induced by the novel notion of quasi-v-solvability. Quasi-v-solvability is founded on a generalisation of needed reduction to the λ_V-calculus. The theory $\mathcal{H}_V$ is proven to be consistent. Full reduction is intrinsic to the definition of quasi-v-solvability and to the proof theory of the λ_V-calculus.

[5] A term T is *effectively used* in a computation $\mathbf{C}[T]$ iff is not the case that $\mathbf{C}[T] \twoheadrightarrow_\beta^* N \in \mathsf{NF}$ implies $\forall T'.\mathbf{C}[T'] \twoheadrightarrow_\beta^* N$ (Paolini & Ronchi Della Rocca, 1999).

[6] A λ-theory $\mathcal{T}$ is *Hilbert-Post complete* iff for every equation $M = N$, either $\mathcal{T} \vdash M = N$ or $\mathcal{T} + (M = N)$ is inconsistent (Barendregt, 1984).

Part I

Meta-Theory

3
The Beta Cube

Mr Reynolds, this is the acorn that will grow a great oak.

(*Ed Wood*, Tim Burton)

We define a big-step-style template for reduction strategies that can be instantiated to the foremost (and more) reduction strategies of the pure lambda calculus. We implement the template in Haskell as a parametric monadic reducer whose fixed points are reduction strategies. The resulting code is clean and abstracts away from the machinery required to guarantee semantics preservation for all strategies in lazy Haskell. By interpreting some parameters as boolean switches we obtain a reduction strategy lattice or beta cube which captures the strategy space neatly and systematically. We define a hybridisation function that generates hybrid strategies by composing a base and a subsidiary strategy from the cube. We prove an absorption theorem which states that subsidiaries are left-identities of their hybrids. More properties from the cube remain to be explored.

3.1 Introduction

Sestoft (2002) defines the big-step operational semantics of various reduction strategies for the pure lambda calculus, including call-by-value (bv), call-by-name (bn), applicative order (ao), normal order (no), hybrid applicative order (ha), hybrid normal order (hn), and head spine (he). (Strategy ha is not identical to `byValue` in (Paulson, 1996, p.390), and he is identical to `headNF` in (Paulson, 1996, p.390) but different from head reduction (h) in (Barendregt, 1984).) One of his motivations is to clarify the meaning in the pure lambda calculus of strategies used in programming languages, where there are no free variables nor evaluation under lambda. He finds, for example, varying and inaccurate definitions of bn by several authors, including (Plotkin, 1975). He implements each strategy as a

$$\frac{}{x \Downarrow_p x}\ (\textsc{Var}) \qquad \frac{B \Downarrow_{la} B'}{\lambda x.B \Downarrow_p \lambda x.B'}\ (\textsc{Abs})$$

$$\frac{M \Downarrow_{op_1} M' \quad M' \equiv \lambda x.B \quad N \Downarrow_{ar_1} N' \quad [N'/x]B \Downarrow_{su} E}{M\,N \Downarrow_p E}\ (\textsc{Red})$$

$$\frac{M \Downarrow_{op_1} M' \quad M' \not\equiv \lambda x.B \quad M \Downarrow_{op_2} M'' \quad N \Downarrow_{ar_2} N'}{M\,N \Downarrow_p M''\,N'}\ (\textsc{App})$$

Figure 3.1: Template for reduction strategies

reducer function in ML using a deep embedding of lambda terms. He does not discuss the paramount implementation issue, first noted by Reynolds (Reynolds, 1998) in the context of interpretation, of semantics preservation or independence from the evaluation strategy of the implementation language: implementing *no* in an eager language, *ao* in a lazy language, etc. Reynolds showed that continuation-passing style is enough for semantics preservation, but it has a cost in code readability.

We take Sestoft's programme much further.

First, we define a big-step-style template for reduction strategies that can be instantiated to all the aforementioned strategies and more. We implement the template in Haskell as a higher-order monadic function whose fixed points are reduction strategies. We like to think of this function as a generic reducer although technically it is a functional. The resulting code is readable, clean, and abstracts away from the machinery required to guarantee semantics preservation for all strategies in lazy Haskell.

Second, by interpreting some parameters of the generic reducer as boolean switches, we obtain a reduction strategy lattice we call the β-cube (after Barendregt's λ-cube). The β-cube captures neatly and systematically many reduction strategies of the pure lambda calculus. It contains eight uniform strategies: *ao*, *bv*, *bn*, *he*, and four new ones. We define a hybridisation function that generates up to ten hybrid strategies (including *no*, `byValue`, *hn*, and *h*) by appropriately composing a base and a subsidiary strategy taken from the cube. We prove an absorption theorem which states that subsidiaries are left-identities of their hybrids. More properties from the cube remain to be explored.

3.2 Rule template and generic reducer

Figure 3.1 defines a big-step-style template for reduction strategies p that is parametric on strategies la, op_1, ar_1, su, op_2, and ar_2:[1]

[1]We consistently use the red, green, and blue colours respectively for the la, ar_1, and ar_2 parameters of the generic reducer and for the axis in the β-cube (Figure 3.3).

```
type Red = Monad m => Term -> m Term
genred :: Red -> Red -> Red -> Red -> Red -> Red -> Red
genred la op1 ar1 su op2 ar2 t =
  case t
    of v@(Var _) -> return v
       (Lam v b) -> do b' <- la b
                       return (Lam v b')
       (App m n) -> do m' <- op1 m
                       case m'
                         of (Lam v b) -> do n' <- ar1 n
                                            su (subst b n' v)
                            _         -> do m'' <- op2 m
                                            n'' <- ar2 n
                                            return (App m'' n'')

ao = genred ao     ao ao     ao ao ao
bv = genred return bv bv     bv bv bv
bn = genred return bn return bn bn return
he = genred he     he return he he return
...
```

Figure 3.2: Generic reducer in Haskell

Rule ABS leaves to la reduction under lambda. Rule RED relies on op_1 to find the redex's abstraction, on ar_1 to reduce the operand, and on su to reduce after substitution. Rule APP describes what to do when op_1 delivers a variable or a non-op_1-reducible application. The result is the application with subterms reduced by op_2 and ar_2. The reason of the op_1 and *optwo* parameters in rule RED is to capture the hybrid strategies in (Sestoft, 2002) in a straightforward way. The parameter op_2 reduces the M instead of the M' to preserve orthogonality of the parameters (*i.e.*, the result of op_2 does not depend on op_1). The shape of terms M', N', E, etc, depends on what sort of normal form the parameter strategies deliver (if they terminate). For example, nor is p with $la, su, op_2 = p$, $op_1 = bn$, and $ar_1, op_2 = \mathrm{id}$, whereas bv is p with $op_1, ar_1, su, op_2, ar_2 = p$ and $la = \mathrm{id}$.

If the left-hand-sides of conclusions are non-overlapping then the rules are deterministic (Baader & Nipkow, 1998). This is the case after the computation of the leftmost premise in the APP and RED rules. The template can therefore be interpreted as a syntax-directed partial function in which a term matching the left-hand-side of the conclusion is recursively reduced by strategies in the premises from left to right. Infinite derivation accounts for non-termination. The code in Figure 3.2 implement this function in Haskell as a monadic higher-order function shown in Figure 3.2 (colours explained in Section 3.3). The monad constraint `m` must be instantiated to a monad that guarantees semantics preservation (*e.g.*, CPS or strict monad). Specific reducers are fixed points of the function. In the monadic

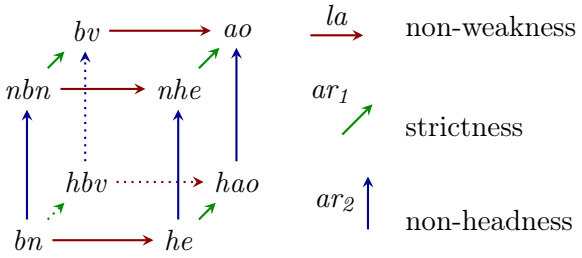


Figure 3.3: The β-cube

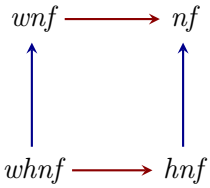


Figure 3.4: Normal forms

code `return` corresponds to the identity strategy.

3.3 The β-cube

The generic reducer `genred` has six parameters. We decrease the number of parameters by focusing on uniform strategies, which are those where `op1`, `op2`, and `su` are recursive calls. So-called hybrid strategies rely on other strategies for `op1`. For example, nor and h rely on bn, hn relies on he, and `byValue` and ha rely on bv (Sestoft, 2002). Uniform strategies differ on whether `la`, `ar1` and `ar2` are either recursive calls or `return`. We can encode this variability using the Cartesian product of three booleans. The obvious partial order relation on them induces a lattice we call the β-cube (Figure 3.3). Function `cube2red` in Figure 3.5 delivers a uniform reducer from a vertex in the cube. Some vertices correspond to novel strategies we call head applicative order (hao), head call-by-value (hbv), non-head spine (nhe) and non-head call-by-name (nbn). Indeed, the boolean parameters (`la`, `ar1` and `ar2`) respectively specify non-weakness (whether abstraction bodies are reduced), strictness (whether arguments are reduced), and non-headness (whether operands with non-reducible applications as operators are reduced). Unsurprisingly, the front and back faces of the cube describe the informal inclusion relation between normal forms ('less reducible form') along the non-headness and non-weakness axes.

```
data BetaCube = BC Bool Bool Bool

sel :: Bool -> Red -> Red
sel b r = if b then r else return

cube2red :: BetaCube -> Red
cube2red (BC la ar1 ar2) =
  let r = genred (sel la r) r (sel ar1 r) r r (sel ar2 r)
  in r

bn = cube2red (BC False False False)
bv = cube2red (BC False True  True )
...
```

Figure 3.5: Function `cube2red`

3.4 Hybridisation

Recall from Section 3.3 that hybrid strategies rely on a uniform strategy (let us call it *subsidiary*) for the `op1` argument (the op_1 strategy in the template). Interestingly, hybrid strategies can be obtained by composing their subsidiary with another uniform strategy from the cube (let us call it *base*) that specifies the behaviour for `la`, `ar1`, and `ar2`. The subsidiary is in general expected to perform less reduction than the base because the former is only used to locate the redex. The following hybridisation function delivers a hybrid strategy from a subsidiary and a base:

```
hybridise :: (BetaCube, BetaCube) -> Red
hybridise (sub, (BC lab ar1b ar2b)) =
  let s = cube2red sub
      h = genred (sel lab h) s (sel ar1b s) h (s >=> h) (sel ar2b h)
  in h
```

Notice that the Kleisli composition `s >=> h` is the monadic implementation of the relational composition $\Downarrow_h \circ \Downarrow_s$ and therefore `op2` reduces at least as much as `op1`. For illustration, we show nor as a fixed point of `genred`, using Kleisli composition for the `op2` argument, and as a hybrid of *bn* (subsidiary) and *nhe* (base):

```
no = genred no bn return no (bn >=> no) no
no = hybridise (BC False False False) (BC True False True)
```

The other cases are: head reduction (h) is a hybrid of *bn* and *he*, hybrid normal order (hn) is a hybrid of *he* and *nhe*, and `byValue` in (Paulson, 1996, p.390) is a hybrid of *bv* and *ao*. The strategy `byValue` differs from hybrid applicative order (ha) in (Sestoft, 2002) because in the back face of the cube (strictness) the choice is between `return` or the subsidiary (not the hybrid) for `ar1`. This has consequences for the absorption.

3.5 Absorption

Proposition 3.5.1 (Absorption)**.** *Let s and b be respectively a subsidiary and a base strategy that have the same* `ar1` *parameter and that considered as points in the cube satisfy $s \sqsubseteq b$,* i.e.*, the $\sqsubseteq$ is the obvious partial order induced by the Cartesian product of the boolean parameters (see Section 3.3). Let h be the hybrid strategy obtained from them. Then s is a right identity of h, that is, $\Downarrow_h \circ \Downarrow_s = \Downarrow_h$.*

Absorption is important because it manifests that the subsidiary is a subrelation of the hybrid, and this is the key to connect hybrid strategies with the eval/readback approach (*i.e.*, the subsidiary could be split off as a separate eval stage).

3.6 Conclusions and future work

The β-cube captures neatly and systematically the foremost reduction strategies of the pure lambda calculus by means of its uniform strategies and of a hybridisation function that completes the space. The cube helps uncover properties of strategies. Absorption is one example, but more remain to be explored. The reduction-strategy template suggests itself naturally as a generalisation of the rules of all the well- and less-well-known strategies collected by (Sestoft, 2002). The need for op_1 and op_2 in rule APP to accommodate hybrids is perhaps the only subtlety. We are surprised that generic reduction for the pure lambda calculus has, to our knowledge, not been considered before nor its consequences been investigated (*e.g.*, hybrid strategies can be defined in terms of two uniform strategies). The Haskell implementation is deceivingly straightforward. It requires careful attention to semantics preservation and deployment of some advanced Haskell programming. The beta cube is one way of focusing on a subspace of the generic reducer, that of uniform strategies, from which we can obtain more (even new) strategies and state properties.

It is possible to construct versions of our generic reducer for other calculi (simply typed, System F, etc.) and for other representations (de Bruijn indices, nominal terms, explicit substitutions, etc). It is also possible to carry the idea to compositional evaluators (interpreters as in (Reynolds, 1998; Danvy, 2008a) or normalisation by evaluation (Danvy, 1996; Filinski & Rohde, 2004)). We also wish to formalise the cube and prove properties like absorption in terms of reduction strategies as mathematical functions on the set of lambda terms. A first-order inductive representation (which alleviates the pain of α-equivalence) will surely help simplify the number of lemmas and proofs.

Addendum

3.7 Monadic style and semantics preservation

Section 3.1 commented on the semantics preservation issue noted by (Reynolds, 1998) in the context of interpretation. CPS is enough to enforce semantics preservation but, as advocated through the chapter, a monadic style is also suitable for this purpose and improves code readability. The choice of monad is important, only a monad which entails a strict bind operator (>>=) will preserve semantics of the target language. Let us show this fact with an example. Consider the following monadic call-by-value evaluator, where the `m` could be instantiated with any monad:

```
cbv :: Monad m => Term -> m Term
cbv v@(Var _)   = return v
cbv l@(Lam _ _) = return l
cbv (App m n)   = do m' <- cbv m
                     n' <- cbv n
                     case m'
                       of (Lam v b) -> do s <- return $ subst b n' v
                                          cbv s
                          _         -> return $ App m' n'
```

We assume that `Term` is an instance of `Show` (the `show` function is the pretty-printing loop that pulls evaluation in lazy Haskell) and we assume a substitution function `subst` that is non-strict in its argument `n'`. (A direct implementation of freshness for variables would force evaluation of `n'`, but here we allow an implementation which delays this evaluation, which is also possible.) We test the functional semantics of `cbv` by applying the evaluator to term $K\,I\,\Omega$, which is encoded as the `Term`

```
constIdentOmega = App (App const ident) omega
  where const = Lam "x" $ Lam "y" $ Var "x"
        ident = Lam "x" $ Var "x"
        omega = App twice1 twice2
        twice1 = Lam "x" $ App (Var "x") $ Var "x"
        twice2 = Lam "x" $ App (Var "x") $ Var "x"  -- To avoid sharing
```

We load the evaluator on GHCi and run:

```
*Main> cbv constIdentOmega
```

The evaluator diverges (*i.e.*, returns undefined) which corresponds to the expected behaviour $K\,I\,\Omega \Uparrow_{bv}$. By default, GHCi instantiates `m` with the `IO` monad, which is strict. However, we can force a failure of the intended functional semantics by instantiating `m` with the `Identity` monad.

```
*Main> import Control.Monad.Identity
*Main Control.Monad.Identity> runIdentity $ cbv constIdentOmega
\x.x
```

Now GHCi returns `ident` (pretty-printed as `\x.x`) which corresponds to $K\,I\,\Omega \Downarrow_{bv} I$. This violates the intended strict-functional semantics, call-by-value should have reduced operand Ω forever.

Let us look at the implementation of `Identity` in module `Control.Monad.Identity`:

```
newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
     fmap f m = Identity (f (runIdentity m))

instance Monad Identity where
     return a = Identity a
     m >>= k  = k (runIdentity m)
```

By code expansion we have

```
  return undefined >>= (\x -> return $ const () x)
=
  Identity undefined >>= (\x -> return $ const () x)
=
  (\x -> return $ const () x) (runIdentity (Identity undefined))
=
  (\x -> return $ const () x) undefined
=
  return $ const () undefined
=
  return ()
=
  Identity ()
```

which shows that the monadic bind operator `>>=` is here non-strict in its left operand (in the monadic sense). If we consider the Kleisli category (Mac Lane, 1971) and Kleisli composition `>=>`, then the composition of a strict function (which may return `undefined`) with a non-strict function (like the `const ()` above) will not be a strict function, and this breaks the intended strict functional semantics.

However, most of Haskell monads are strict. In particular the `Cont` monad, which implements CPS and adheres to the original solution for semantics preservation in (Reynolds, 1998).

3.8 Absorption and normalisation by evaluation

In this section we prove Absorption (Proposition 3.5.1), which states that a subsidiary is a right identity of the hybrid produced after it by the `hybridise` combinator. To this aim, we consider the normalisation by evaluation (NBE) approach (Berger & Schwichtenberg, 1991; Danvy, 1998; Filinski & Rohde, 2004; Aehlig & Joachimski, 2004; Filinski & Rohde, 2005), where normalisation (*i.e.*, reduction) consists of the composition of an eval stage that delivers terms in some intermediate irreducible form, with a readback stage that distributes reduction over the subterms of the intermediate forms.

We implement the hybrid strategies using a NBE approach. The implementation consists of parametric `sub` and `reb` which stand for the subsidiary (the eval stage in NBE) and for the readback loop respectively. The parameters specify the three β-cube coordinates for the subsidiary (`las`, `ar1`, and `ar2s`) and the two remaining coordinates for the base (`lab` and `ar2b`). (Recall from Section 3.4 that when producing hybrids, the `ar1` parameter is fixed in both subsidiary and base.)

```
sub :: Bool -> Bool -> Bool -> Red
sub las ar1 ar2s v@(Var _) = return v
sub las ar1 ar2s (Lam v b) =
  do b' <- (sel las (sub las ar1 ar2s)) b
     return (Lam v b')
sub las ar1 ar2s (App m n) =
  do m' <- (sub las ar1 ar2s) m
     case m'
       of (Lam v b) ->
            do n' <- (sel ar1 (sub las ar1 ar2s)) n
               (sub las ar1 ar2s) (subst b n' v)
          _         ->
            do n' <- (sel ar2s (sub las ar1 ar2s)) n
               return (App m' n')

reb :: Bool -> Bool -> Bool -> Bool -> Bool -> Red
reb las ar1 ar2s lab ar2b v@(Var _) = return v
reb las ar1 ar2s lab ar2b (Lam v b) =
  do b' <- ((sel (xor las lab) (sub las ar1 ar2s))
            >=> (sel lab (reb las ar1 ar2s lab ar2b))) b
     return (Lam v b')
reb las ar1 ar2s lab ar2b (App m n) =
  do m' <- (reb las ar1 ar2s lab ar2b) m
     n' <- ((sel (xor ar2s ar2b) (sub las ar1 ar2s))
            >=> (sel ar2b (reb las ar1 ar2s lab ar2b))) n
     return (App m' n')
```

Function `sub` delivers a strategy equivalent to that implemented by combinator `cube2red` and function `genred` together. Notice the use of the selection function `sel` and the recursive invocations `(sub las ar1 ar2s)`. Function `reb` implements a readback loop which distributes the strategy over the subterms of the intermediate result. Readback intertwines

nested evaluation stages according to the parameters of the base strategy. For the *la* and ar_2 parameters, if they are recursive calls already in the subsidiary, readback omits the evaluation stage and only distribute itself. The 'exclusive or' operator `xor` in the selection clauses `(sel (xor ... ...) ...)` takes care of this. The normal order strategy, which is the hybrid with subsidiary *bn* (`BC False False False`) and base *nhe* (`BC True False True`) can be defined as

```
no :: Red
no = (sub False False False) >=> (reb False False False True True)
```

A hybrid strategy is equivalent to the composition of its subsidiary and its corresponding readback stage, *i.e.*, $\Downarrow_h = \Downarrow_{rb} \circ \Downarrow_s$. In what follows, we prove this correspondence formally by fusing the `reb` and `sub` above by lightweight fusion by fixed-point promotion (Ohori & Sasano, 2007). We will arrive at an implementation consisting of two mutually recursive functions which coincide with the hybrid and subsidiary strategies entailed by combinators `hybridise` and `cube2red`, and by the generic reducer `genred`.

We follow the transformation steps in (Ohori & Sasano, 2007) and adopt their terminology. Let us write `fix` $s.\lambda x.E_s$ and `fix` $rb.\lambda x.E_{rb}$ for the recursive definitions of $\Downarrow_s$ and $\Downarrow_{rb}$ respectively. In the code, the name definitions and the parameters `sub las ar1 ar2s` and `reb las ar1 ar2s lab ar2b` stand for the `fix` $s.$ and `fix` $rb.$ respectively. The term argument that comes next corresponds to the $\lambda x.$ part. We obtain the composition $\Downarrow_{rb} \circ \lambda x.E_s$, which beta reduces to $\lambda x.\ \Downarrow_{rb} E_s$, and name it `reb1`.

```
reb1 :: Bool -> Bool -> Bool -> Bool -> Bool -> Red
reb1 las ar1 ar2s lab ar2b v@(Var _) =
  do v' <- return v
     (reb las ar1 ar2s lab ar2b) v'
reb1 las ar1 ar2s lab ar2b (Lam v b) =
  do b' <- (sel las (sub las ar1 ar2s)) b
     l' <- return (Lam v b')
     (reb las ar1 ar2s lab ar2b) l'
reb1 las ar1 ar2s lab ar2b (App m n) =
  do m' <- (sub las ar1 ar2s) m
     case m'
       of (Lam v b) ->
            do n' <- (sel ar1 (sub las ar1 ar2s)) n
               s' <- (sub las ar1 ar2s) (subst b n' v)
               (reb las ar1 ar2s lab ar2b) s'
          _         ->
            do n' <- (sel ar2s (sub las ar1 ar2s)) n
               a' <- return (App m' n')
               (reb las ar1 ar2s lab ar2b) a'
```

The `do` notation forces the invocations of $\Downarrow_{rb}$ (*i.e.*, the `(reb ...)`) to be already in tail position. Now we simplify the `do` notation by applying right identity of monadic bind and by using the Kleisli composition `>=>`.

```
reb2 :: Bool -> Bool -> Bool -> Bool -> Bool -> Red
reb2 las ar1 ar2s lab ar2b v@(Var _) =
  (reb las ar1 ar2s lab ar2b) v
reb2 las ar1 ar2s lab ar2b (Lam v b) =
  do b' <- (sel las (sub las ar1 ar2s)) b
     (reb las ar1 ar2s lab ar2b) (Lam v b')
reb2 las ar1 ar2s lab ar2b (App m n) =
  do m' <- (sub las ar1 ar2s) m
     case m'
       of (Lam v b) ->
            do n' <- (sel ar1 (sub las ar1 ar2s)) n
               ((sub las ar1 ar2s)
                >=> (reb las ar1 ar2s lab ar2b)) (subst b n' v)
          _         ->
            do n' <- (sel ar2s (sub las ar1 ar2s)) n
               (reb las ar1 ar2s lab ar2b) (App m' n')
```

Next we inline $\Downarrow_{rb}$ in E_s (*i.e.*, inline `reb` in `reb2`, obtaining `reb3`).

```
reb3 :: Bool -> Bool -> Bool -> Bool -> Bool -> Red
reb3 las ar1 ar2s lab ar2b v@(Var _) = return v
reb3 las ar1 ar2s lab ar2b (Lam v b) =
  do b' <- (sel las (sub las ar1 ar2s)) b
     b'' <- ((sel (xor las lab) (sub las ar1 ar2s))
             >=> (sel lab (reb las ar1 ar2s lab ar2b))) b'
     return (Lam v b'')
reb3 las ar1 ar2s lab ar2b (App m n) =
  do m' <- (sub las ar1 ar2s) m
     case m'
       of (Lam v b) ->
            do n' <- (sel ar1 (sub las ar1 ar2s)) n
               ((sub las ar1 ar2s)
                >=> (reb las ar1 ar2s lab ar2b)) (subst b n' v)
          _         ->
            do n' <- (sel ar2s (sub las ar1 ar2s)) n
               m'' <- (reb las ar1 ar2s lab ar2b) m'
               n'' <- ((sel (xor ar2s ar2b) (sub las ar1 ar2s))
                       >=> (sel ar2b (reb las ar1 ar2s lab ar2b))) n
               return (App m'' n'')
```

We simplify again and apply associativity of monadic bind to sequence together the computations that run over the same intermediate results.

```
reb4 :: Bool -> Bool -> Bool -> Bool -> Bool -> Red
reb4 las ar1 ar2s lab ar2b v@(Var _) = return v
reb4 las ar1 ar2s lab ar2b (Lam v b) =
  do b' <- ((sel las (sub las ar1 ar2s))
            >=> ((sel (xor las lab) (sub las ar1 ar2s))
                 >=> (sel lab (reb las ar1 ar2s lab ar2b)))) b
     return (Lam v b')
reb4 las ar1 ar2s lab ar2b (App m n) =
  do m' <- (sub las ar1 ar2s) m
     case m'
       of (Lam v b) ->
            do n' <- (sel ar1 (sub las ar1 ar2s)) n
               ((sub las ar1 ar2s)
                >=> (reb las ar1 ar2s lab ar2b)) (subst b n' v)
          _         ->
            do m'' <- (reb las ar1 ar2s lab ar2b) m'
               n' <- ((sel ar2s (sub las ar1 ar2s))
                      >=> ((sel (xor ar2s ar2b) (sub las ar1 ar2s))
                           >=> (sel ar2b (reb las ar1 ar2s lab ar2b)))) n
               return (App m'' n')
```

Now we rename the fragments

```
((sel p (sub ...))
  >=> ((sel (xor p p') (sub ...))
       >=> (sel p' (reb ...))))
```

into `(sel p' ((sub ...) >=> (reb ...)))`. This transformation holds because of condition $\texttt{p} \sqsubseteq \texttt{p'}$ among the coordinates of subsidiary and base in Proposition 3.5.1 (*i.e.*, the subsidiary has to be less or equal than the base in the β-cube lattice).

```
reb5 :: Bool -> Bool -> Bool -> Bool -> Bool -> Red
reb5 las ar1 ar2s lab ar2b v@(Var _) = return v
reb5 las ar1 ar2s lab ar2b (Lam v b) =
  do b' <- (sel lab ((sub las ar1 ar2s)
                     >=> (reb las ar1 ar2s lab ar2b))) b
     return (Lam v b')
reb5 las ar1 ar2s lab ar2b (App m n) =
  do m' <- (sub las ar1 ar2s) m
     case m'
       of (Lam v b) ->
            do n' <- (sel ar1 (sub las ar1 ar2s)) n
               ((sub las ar1 ar2s)
                >=> (reb las ar1 ar2s lab ar2b)) (subst b n' v)
          _         ->
            do m'' <- (reb las ar1 ar2s lab ar2b) m'
               n' <- (sel ar2b ((sub las ar1 ar2s)
                                >=> (reb las ar1 ar2s lab ar2b))) n
               return (App m'' n')
```

We prepone an invocation of the subsidiary `(sub las ar1 ar2s)` before the readback `(reb las ar1 ar2s lab ar2b)` that reduces the operator `m` in applications.

```
reb6 :: Bool -> Bool -> Bool -> Bool -> Bool -> Red
reb6 las ar1 ar2s lab ar2b v@(Var _) = return v
reb6 las ar1 ar2s lab ar2b (Lam v b) =
  do b' <- (sel lab ((sub las ar1 ar2s)
                     >=> (reb las ar1 ar2s lab ar2b))) b
     return (Lam v b')
reb6 las ar1 ar2s lab ar2b (App m n) =
  do m' <- (sub las ar1 ar2s) m
     case m'
       of (Lam v b) ->
            do n' <- (sel ar1 (sub las ar1 ar2s)) n
               ((sub las ar1 ar2s)
                >=> (reb las ar1 ar2s lab ar2b)) (subst b n' v)
          _          ->
            do m'' <- ((sub las ar1 ar2s)
                       >=> (reb las ar1 ar2s lab ar2b)) m'
               n' <- (sel ar2b ((sub las ar1 ar2s)
                                >=> (reb las ar1 ar2s lab ar2b))) n
               return (App m'' n')
```

This additional reduction, whose aim is to match the behaviour of op_1 and op_2 in the generic reducer `genred` of Figure 3.2, does not alter the strategy because of idempotency of strategies. (Recall that `genred` invokes op_2 over the unreduced operator `m` to enforce independence between op_1 and op_2 parameters. Here, these two parameters are no longer independent because of the staged character of NBE.) Now to the last step. Let E_{rb_s} be the body of the simplified `(reb6 ...)`. We replace the occurrences of $\Downarrow_{rb} \circ \Downarrow_s$ in E_{rb_s} by a new function name $\Downarrow_{rb\_s}$, and we generate a new binding $\Downarrow_{rb\_s}$= `fix` $rb\_s.\lambda x.E_{rb\_s}$. This is, we rename `(sub ...) >=> (reb ...)` into `(reb7 ...)` in the definition of new `reb7`.

```
reb7 :: Bool -> Bool -> Bool -> Bool -> Bool -> Red
reb7 las ar1 ar2s lab ar2b v@(Var _) = return v
reb7 las ar1 ar2s lab ar2b (Lam v b) =
  do b' <- (sel lab (reb7 las ar1 ar2s lab ar2b)) b
     return (Lam v b')
reb7 las ar1 ar2s lab ar2b (App m n) =
  do m' <- (sub las ar1 ar2s) m
     case m'
       of (Lam v b) ->
            do n' <- (sel ar1 (sub las ar1 ar2s)) n
               (reb7 las ar1 ar2s lab ar2b) (subst b n' v)
          _          ->
            do m'' <- ((sub las ar1 ar2s)
                       >=> (reb7 las ar1 ar2s lab ar2b)) m'
               n' <- (sel ar2b (reb7 las ar1 ar2s lab ar2b)) n
               return (App m'' n')
```

Functions `sub` and `reb7` match the implementation of the subsidiary and hybrid strategies entailed by combinators `hybridise`, `cube2red`, and `genred` as follows.[2] The `sub las ar1 ar2s` coincides with `cube2red (BC las ar1 ar2s)` and

```
(sub las ar1 ar2s) >=> (reb las ar1 ar2s lab ar2b)
```

coincides with

```
hybridise (cube2red (BC las ar1 ar2s)) (BC lab ar1 ar2b)
```

Lemma 3.8.1 (Correspondence to NBE)**.** *For every hybrid strategy h obtained from subsidiary s and base b from the cube where $s \sqsubseteq b$ and where both s and b have the same ar_1 parameter, there exists a readback loop rb such that $\Downarrow_h = \Downarrow_{rb} \circ \Downarrow_s$.*

Proof. By the program derivation presented in this section, taking functions `sub` and `reb` as the eval and readback stages respectively. □

Absorption is now proven easily by resorting to the NBE approach.

Proof of Proposition 3.5.1 (Absorption). By Lemma 3.8.1 any hybrid strategy is decomposed into eval and readback stages $\Downarrow_h = \Downarrow_{rb} \circ \Downarrow_s$. We need $(\Downarrow_{rb} \circ \Downarrow_s) \circ \Downarrow_s = \Downarrow_{rb} \circ \Downarrow_s$, which holds by idempotency of $\Downarrow_s$. □

As a consequence of Absorption, the `hybridise` function in Section 3.4 can be optimised by replacing the composition of subsidiary and hybrid (*i.e.*, `s >=> h`) with only the hybrid (*i.e.*, `h`). Similarly, when defining hybrid strategies by instantiating the generic reducer `genred` of Section 3.2, it is enough to pass the hybrid for the op_2 parameter. The following definition,

```
no = genred no bn id no no no
```

is equivalent to the definition in Section 3.4.

3.9 Balanced generic template

The generic template and reducer in Section 3.2 have been designed with the aim of making the parameters in the template orthogonal to each other. This explains why, in Figure 3.1, the M' obtained after the premiss $M \Downarrow_{op_1} M'$ in rule App is not piped to the premiss $M \Downarrow_{op_2} M''$ in the same rule. If it was, parameters op_1 and op_2 would no longer be independent. This is a source for inefficiency of the generic reducer in Figure 3.2, which manifests in the fact that the op_2 parameter may repeat some work that was already done by the op_1 parameter.

The hybridisation function in Section 3.4 has been designed to uphold Absorption (see Section 3.5). In order to achieve absorption, the `ar1` parameter in both the hybrid and

[2] The versed reader may notice that in the generic reducer the `m''` is produced after the `m`, instead of after the `m'` in the code avobe. Again, the equivalence trivially holds by idempotency of `(sub ...)`.

$$\frac{}{x \Downarrow_p x}\ (\textsc{Var}) \qquad \frac{B \Downarrow_{la} B'}{\lambda x.B \Downarrow_p \lambda x.B'}\ (\textsc{Abs})$$

$$\frac{M \Downarrow_{op_1} M' \quad N \Downarrow_{ar_1} N' \quad M' \equiv \lambda x.B \quad [N'/x]B \Downarrow_{su} E}{M\ N \Downarrow_p E}\ (\textsc{Red})$$

$$\frac{M \Downarrow_{op_1} M' \quad N \Downarrow_{ar_1} N' \quad M' \not\equiv \lambda x.B \quad M' \Downarrow_{op_2} M'' \quad N' \Downarrow_{ar_2} N''}{M\ N \Downarrow_p M''\ N''}\ (\textsc{App})$$

Figure 3.6: Balanced template for reduction strategies

the subsidiary has to be the same, *i.e.*, the strategies have to be hybridised *in each face of the β-cube separately*. Besides, in the case of the strict semantics, the operands in redices have to be reduced by the subsidiary. As a consequence, the choice of which strategy for `ar1` in the hybridisation function is not between `return` or the hybrid, but between `return` or the subsidiary. This is the reason why the `byValue` of (Paulson, 1996) is obtained when hybridising *bv* with *ao*, instead of the hybrid applicative order of (Sestoft, 2002). (Hybrid applicative order reduces operands in some redices fully, and thus the calling policy that it implements is neither the by-wnf nor the by-value. We refer to this calling policy as *by-normal-form* (or just *by-nf*), although not all the operands are reduced to normal form. The strategy can be defined by instantiating the generic reducer directly:

```
ha = genred ha bv ha bv ha ha
```

Orthogonality and absorption suggest a refined version of template and reducer which we depict in Figures 3.6 and 3.7. We refer to these as *balanced* (*i.e.*, balanced template, balanced reducer) because the subsidiary and hybrid stages are split, respectively, into the moments before and after contraction of the outermost redex by the external substitution function $[N/x]B$. In rule Red, the premisses reducing both operator M and operand N occur before the side condition $M' \equiv \lambda x.B$. In rule App, the same pattern is followed for the template to be syntax-directed, *i.e.*, rules Red and App share the two first premisses, after which the condition is checked. The balanced template and reducer are optimised in that the intermediate M' and N' obtained in rule App are respectively piped as inputs to the fourth and fifth premisses in the same rule.

The careful reader may notice that with this balanced version, the four strategies in the strict face (the back face) of the β-cube collapse into a pair of strategies, since the ar_1 and ar_2 parameters are no longer independent. This is akin to the observation that it does not matter anymore which `ar2` (`return` or the hybrid) is passed in the definition of `bv` and `ao` in Figure 3.7. The β-cube turns into the triangular β-prism of Figure 3.8.

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

ao = bal_genred ao     ao ao     ao return return
bv = bal_genred return bv bv     bv return return
bn = bal_genred return bn return bn return return
he = bal_genred he     he return he return return
...
```

Figure 3.7: Balanced generic reducer

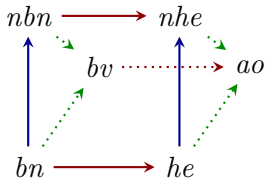


Figure 3.8: The collapsed β-prism

However, the `ar2` parameter still has a role in the definition of hybrid strategies. Consider the balanced functions for defining uniform and hybrid strategies in Figure 3.9. Although the strict segment of the β-prism only contains two uniform strategies (*i.e.*, *bv* and *ao*) these can be hybridised in various ways, attending to whether the subsidiary and the base have the same `ar2` parameter. Paulson's `byValue` and Paolini's ahead machine $\Downarrow_a^0$ can be defined as follows:

```
byValue = bal_hybridise (BC False True False, BC True True True)
a0      = bal_hybridise (BC False True False, BC True True False)
```

The subsidiary is *bv* in both cases, and the base strategies differ only in the `ar2`. In the uniform strategies, the differences in the `ar2` are shadowed. In the base strategies, these differences stand out when defining hybrids.

```
bal_cube2red :: BetaCube -> Red
bal_cube2red (BC la ar1 ar2) =
  let r = bal_genred (sel la r) r (sel ar1 r) r r (sel ar2 r)
  in r

bal_hybridise :: (BetaCube, BetaCube) -> Red
bal_hybridise (sub, (BC lab ar1b ar2b)) =
  let s = bal_cube2red sub
      h = bal_genred (sel lab h) s (sel ar1b s) h h (sel ar2b h)
  in h
```

Figure 3.9: Balanced functions for uniform and hybrid strategies

3.10 Spine strategies

The strategy head spine order (*he*) (Sestoft, 2002) is founded on the notion of head spine reduction in (Barendregt *et al.*, 1987), where the 'spine' terminology is introduced. The terminology comes from taking the anatomical metaphor of 'head reduction' and 'head normal forms' further, which considers the abstract syntax tree of a term as a skeleton, with the head on the top and the spine on the leftmost front of the tree (*i.e.*, the leftmost edges starting in the head). Head spine order is the deterministic strategy that contract the innermost head spine redex. The well-known head reduction (*h*) (Barendregt, 1984) is the standard strategy that contracts the outermost head spine redex (*i.e.*, it reduces the leftmost-outermost redex up to head normal form). Both head spine order and head reduction are complete with respect to head normal form. The latter keeps out of abstractions as long as possible, and the former enters abstractions as eagerly as possible, but never reduces at the right of a variable, thus traversing the spine of the term. Both strategies deliver the same result, but they entail different reduction sequences. Head spine order is a uniform strategy,

```
he = cube2red (BC True False False)
```

and head reduction is the hybrid of subsidiary call-by-name and base head spine order:

```
h = hybridise bn (BC True False False)
```

Broadly, we use 'spine' to refer to the strategies that enter abstractions but that reduce their bodies only enough as to uphold some property, usually completeness with respect to some notion of irreducible form. Hybrid normal order (*hn*) (Sestoft, 2002) is the spine strategy that delivers the same result than normal order (Barendregt, 1984) does. The former is the hybrid of subsidiary head spine and base *nhe*, and the latter is the hybrid of subsidiary call-by-name and base *nhe*. Both are complete with respect to normal form. Hybrid normal order is a spine strategy, normal order is a standard (*i.e.*, outermost) strategy.

$$\frac{}{x \Downarrow_{hao} x}\ (\textsc{Hao-Var}) \qquad \frac{B \Downarrow_{hao} B'}{\lambda x.B \Downarrow_{hao} \lambda x.B'}\ (\textsc{Hao-Abs})$$

$$\frac{M \Downarrow_{hao} M' \quad M' \equiv \lambda x.B \quad N \Downarrow_{hao} N' \quad [N'/x]B \Downarrow_{hao} E}{M\ N \Downarrow_{hao} E}\ (\textsc{Hao-Red})$$

$$\frac{M \Downarrow_{hao} M' \quad M' \not\equiv \lambda x.B}{M\ N \Downarrow_{hao} M'\ N}\ (\textsc{Hao-App})$$

$$\frac{}{x \Downarrow_{sa} x}\ (\textsc{Sa-Var}) \qquad \frac{B \Downarrow_{sa} B'}{\lambda x.B \Downarrow_{sa} \lambda x.B'}\ (\textsc{Sa-Abs})$$

$$\frac{M \Downarrow_{hao} M' \quad M' \equiv \lambda x.B \quad N \Downarrow_{sa} N' \quad [N'/x]B \Downarrow_{sa} E}{M\ N \Downarrow_{sa} E}\ (\textsc{Sa-Red})$$

$$\frac{M \Downarrow_{hao} M' \quad M' \not\equiv \lambda x.B \quad M' \Downarrow_{sa} M'' \quad N \Downarrow_{sa} N'}{M\ N \Downarrow_{sa} M''\ N'}\ (\textsc{Sa-App})$$

Figure 3.10: Natural semantics of spine applicative order

3.10.1 Spine applicative order

Figure 3.10 introduces spine applicative order (*sa*), which is the strict-functional-semantics counterpart of hybrid normal order. Spine applicative order is the hybrid strategy obtained from subsidiary head applicative order (*hao*) (see Section 3.3) and from base applicative order (*ao*) but such that, similar to hybrid applicative order (see Section 3.9), the hybrid is used to reduce operands in redices, thus implementing the by-nf calling policy. Spine applicative order is defined below by instantiating the (non-balanced) generic reducer `genred` directly:

```
hao = genred hao hao hao hao hao id
sa  = genred sa  hao sa  sa  sa  sa
```

Spine applicative order optimises reduction by fully reducing operands and by reducing operators (eagerly) to head normal form. Spine applicative order is the most eager strategy that, differently from applicative order, allows to implement some thunking mechanism (Danvy & Hatcliff, 1992; Hatcliff & Danvy, 1997). A thunking mechanism is needed when implementing recursive functions by means of a fixed-point combinator: the recursive call, which is a divergent term, is placed inside a thunk that forbids to reduce the divergent term fully. The thunking mechanism that we have in mind consists of a variation of the call-

by-value CPS translation (Plotkin, 1975) where the continuations are delimited, *i.e.*, the intermediate results are thrown prematurely into the current continuation. For illustration, we present below the encoding of the factorial function using such a thunking mechanism:[3]

$$\begin{aligned} FACT \equiv \lambda f.\lambda n.\lambda k.IFTE\ & (ISZERO\, n) \\ & (k\, ONE) \\ & (k(f(PRED\, n)(MULT\, n))) \end{aligned}$$

The term $Y\ FACT\ N\ I$ computes the factorial of n, where $Y \equiv \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$ is Curry's fixed-point combinator, N is the Church encoding of numeral n and the identity I is the initial continuation. The factorial is implemented by passing $MULT\, n$ as a continuation to the recursive call $f(PRED\, n)$. The essence of this thunking mechanism could be summarised as 'protecting by a variable', *i.e.*, the divergent subterm occurs always at the right of the continuation variable k. Although spine applicative order entails strict functional semantics, there is no need to use the call-by-value fixed-point combinator $Z \equiv \lambda f.(\lambda x.f(\lambda t.x\,x\,t))(\lambda x.f(\lambda t.x\,x\,t))$. Both Z and Y keep the divergent subterms at the right of the continuation variable k, and they could both be used with spine applicative order. However, the Y is enough here. The additional 'protecting by a lambda' thunking mechanism that Z introduces is of no use, since spine applicative order reduces the bodies of such thunks.

Different from Paulson's `byValue`—or from any other strategy which relies on the 'protecting by a lambda' thunking mechanism—spine applicative order optimises the operands of the f in each recursive invocation to normal form, and reduces the diverging $Y\ FACT$ to head normal form. This enables a significant speed-up when codifying the general recursive functions by means of fixed-point combinators (Hindley & Seldin, 2008, §4C).

3.11 Generic template and the λ_V-calculus

So far, none of the the strict strategies that have been considered implement the by-value calling policy proper (recall Section 1.2), which restricts operands in redices to *values*, *i.e.*, non-applications. The balanced generic template and reducer can be adapted to implement λ_V-calculus strategies by forbidding contraction of redices when the operand is not a value. The λ_V generic template and reducer are depicted in Figures 3.11 and 3.12.

The uniform pure call-by-value $\Downarrow_{pv}$, and the hybrids value normal order $\Downarrow_{vn}$, value head reduction $\Downarrow_{vh}$, and value spine order $\Downarrow_{vs}$ (see Section 4.9) can be defined as fixed points of `val_genred` in Figure 3.12. Pure call-by-value is a generalisation of eval_V in (Plotkin, 1975) to pure λ_V. Value normal order is the standard full-reducing strategy in λ_V, which only differs from evaluation relation **G** of (Ronchi Della Rocca & Paolini, 2004)

[3] We assume that *ONE* stands for the Church encoding of natural 1, and that *IFTE*, *ISZERO*, *MULT*, and *PRED* encode respectively the logical if-then-else operator, natural comparison with 0, multiplication, and predecessor functions.

$$\frac{}{x \Downarrow_p x}\ (\textsc{Var}) \qquad \frac{B \Downarrow_{la} B'}{\lambda x.B \Downarrow_p \lambda x.B'}\ (\textsc{Abs})$$

$$\frac{M \Downarrow_{op_1} M' \quad N \Downarrow_{ar_1} N' \quad (M' \equiv \lambda x.B) \wedge (N' \in \mathsf{Val}) \quad [N'/x]B \Downarrow_{su} E}{M\ N \Downarrow_p E}\ (\textsc{Red})$$

$$\frac{M \Downarrow_{op_1} M' \quad N \Downarrow_{ar_1} N' \quad (M' \not\equiv \lambda x.B) \vee (N' \notin \mathsf{Val}) \quad M' \Downarrow_{op_2} M'' \quad N' \Downarrow_{ar_2} N''}{M\ N \Downarrow_p M''\ N''}\ (\textsc{App})$$

Figure 3.11: Template for λ_V-reduction strategies

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

pv = val_genred return pv pv pv return return
vn = val_genred vn     pv pv vn vn     vn
vh = val_genred vh     pv pv vh return return
vs = val_genred vs     vh pv vs vs     vs
```

Figure 3.12: Generic reducer for λ_V

in that the latter reduces operands fully before reducing the bodies of *blocks*, *i.e.*, stuck redices. Value head reduction and value spine order are novel strategies that we introduce in Section 4.9. Value head reduction is the λ_V analogous of the ahead machine (Paolini & Ronchi Della Rocca, 1999), and value spine order is a complete full-reducing λ_V-strategy that traverses the spine of the terms (see Section 3.10). Figure 3.12 defines these strategies as a direct instantiation of the by-value generic template. Functions generating uniform and hybrid strategies from points in the cube (analogous to `cube2red` and `hybridise`) are straightforward and omitted.

Table 3.1: All the relevant strategies at a glance

		Full		Head		Weak
		outermost	spine	outermost	spine	
`bal_genred`	by-name	***no***	***hn***	***h***	*he*	*bn*
	by-wnf	`byValue`	**	$\Downarrow_a^0$	**	*bv*
`genred`	by-nf	***ha***	***sa***	—	—	—
`val_genred`	by-value	***vn***	***vs***	***vh***	**	*pv*

3.12 Relevant strategies

The most relevant strategies are depicted in Table 3.1. The strategies in boldface (together with Paulson's `byValue` and Paolini's ahead machine $\Downarrow_a^0$) are hybrid, and the rest are uniform. The top of the table indicates which kind of reduction is performed (full, head, weak) and whether reduction is outermost or spine, and the left of the table indicates which generic reducer is used to instantiate the strategy (`bal_genred`, `genred`, `val_genred`) and which calling policy is implemented (by-name, by-wnf, by-nf, by-value). Notice that the spine applicative order (*sa*), the value normal order (*vn*),[4] the value spine reduction (*vs*), and the value head reduction (*vh*) are novel strategies introduced in this thesis. Applicative order does not appear in Table 3.1. The table only contains strategies that allow some thunking mechanism, which are suitable for the encodings of general recursive functions that use fixed-point combinators. All the strategies in the table are *complete* for some of such encodings, with respect to the corresponding notion of irreducible terms.

The strategies marked as '**' in Table 3.1 can be defined by instantiating the generic reducers, or by the `bal_sube2red` and `bal_hybridise` functions, but we restrain from contriving new names for them. The strategies marked as '—' in Table 3.1 do not rather make sense, since the by-nf policy (recall form Section 3.9 that the by-nf policy reduces *some* operands fully) already requires full-reduction. There are other novel strategies that can be produced by the generic reducers and by the functions for generating uniform and hybrid strategies. Table 3.1 only collects the most relevant ones.

[4]Recall from Section 3.11 that value normal order only differs from evaluation relation **G** of (Ronchi Della Rocca & Paolini, 2004) in that the latter reduces operands fully before reducing the bodies of *blocks*, *i.e.*, stuck redices.

4

Towards a Standard Theory for the Lambda-Value Calculus

Small-soulded men, no matter how agile their fingers, should not attempt it.

(*Preface to Chopin's Etude Op. 25, No. 11*, James Huneker)

The classical lambda calculus has a well-established 'standard theory' in which the notion of solvability characterises the operational relevance of terms. Solvable terms, defined as solutions to a beta-equation, have a 'syntactic' characterisation as terms with head normal form. Unsolvable terms are irrelevant and can be beta-equated without affecting consistency. The derived notions of sensibility and Böhm trees connect the consistent theory with models and with a representation of approximate normal forms.

The lambda-value calculus is the calculus that corresponds to a strict functional programming language whose operational semantics is defined by the SECD machine. The beta-equational definition of solvability has been duly adapted to the pure lambda-value calculus, but the syntactic characterisation (value head normal forms and the ahead machine) involves beta reduction and not beta-value reduction. The v-unsolvable terms cannot be equated without affecting consistency, and some v-normal forms are v-unsolvable and have to be considered irrelevant. This has been ignored in the context of weak reduction (not going under lambda, an ingredient of call-by-value reduction as specified by the SECD machine) because of the existence of initial models in this scenario. However, that does not answer in full the question of v-solvability nor provides a consistent 'standard theory' for pure lambda-value. The problem lies in the emphasis on operational equivalence of closed terms according to SECD. When considering open terms, different v-normal forms with stuck subterms which are operationally equivalent may differ on the scope at which a stuck term pops up. A notion of solvability should take into account this distinction which reflects 'preserving confluence by preserving potential divergence' intrinsic in the by-value mechanism.

We introduce the syntactic notion of quasi-v-solvability, which shows that beta-value reduction is appropriate for solvability and restores the validity of v-normal forms and full reduction. Quasi-v-solvability suffices to prove the pure lambda-value versions of the Genericity Lemma and the 'preservation of unsolvables by application and composition'. All the quasi-v-unsolvables of equal v-order can be consistently equated. This makes possible the definitions of Böhm trees for lambda-value. We also characterise complete full-reducing strategies in lambda-value, broadening Plotkin's standardisation theorem (for there are complete strategies that are not v-standard) by stating and proving the analogous in lambda-value of the Quasi-Leftmost Reduction Theorem.

4.1 Introduction

The classical lambda calculus, λK, has a well-established 'standard theory' consisting of equality (conversion) and reduction theories which enjoy confluence and substitutivity. There are standard and complete reduction strategies, and well-known models. The operational relevance of terms is established by the notion of *solvability*. An unsolvable subterm cannot be effectively used inside a term that delivers a definite result. A term is 'effectively used' if it cannot be replaced by any other term without affecting the result. In λK, the notion of effective use is captured by *head contexts* (this is referred to as *λK's applicative behaviour* in the folklore) and definite results are *normal forms* (nfs). An early characterisation of solvability is that a term M is solvable iff there exists a head context $\mathbf{C}[\,]$ such that $\mathbf{C}[M]$ has a nf (Barendregt, 1971; Wadsworth, 1976). Another equational characterisation, equivalent to the former, replaces 'has a nf' by 'β-converts to the term $I \equiv \lambda x.x$'. The two can be proven equivalent by the following proposition (Wadsworth, 1976):

Proposition 4.1.1. *In λK, for any closed M in nf and any arbitrary term P there exist $N_1, \ldots, N_m$ with $m \geq 0$ such that $M\, N_1 \ldots N_m =_\beta P$.*

An equivalent syntactic characterisation defines solvable terms as those having a *head normal form* (hnf) (Wadsworth, 1976; Barendregt, 1984). Head normal forms are 'normal forms at the first level' (Wadsworth, 1976). Any term with a nf also has a hnf, and there is a recursive reduction strategy known as *head reduction* which delivers a hnf if the term has some (Barendregt, 1984). Unsolvable terms are operationally irrelevant because they can take part in a computation that delivers a definite result but 'such usages are trivial as the use of any term in place of the unsolvable would give the same result' (Wadsworth, 1976). Unsolvable terms remain unsolvable when they are effectively used, since 'unsolvability is preserved by application and composition' (Wadsworth, 1976). Unsolvables can be equated without affecting λK's consistency. In particular, solvability enables the definition of Böhm trees and of a consistent theory $\mathcal{H}$ that equates unsolvable closed terms (Barendregt, 1984). The λK-calculus has a sensible (*i.e.*, satisfying $\mathcal{H}$) model D_∞ that is a solution to Scott's equation $D \cong [D \to D]$ (Scott, 1970; Wadsworth, 1976).

A programming language corresponds with a calculus when programs are closed terms of the calculus and the operational semantics (reduction strategy) is a sub-relation of the reduction relation of the calculus. The study of such correspondence was pioneered by Peter Landin, *e.g.*, (Landin, 1964).

As noted by Abramsky (1990), there is no canonical solution to Scott's equation and, furthermore, non-strict functional programming languages reduce closed terms to *weak head normal form* (whnf), *i.e.*, do not reduce abstractions. Abramsky is concerned with the correspondence between calculi and programming languages. He proposed the 'lazy lambda calculus' (Abramsky, 1990), hereafter referred to as $\lambda\ell$, in which unsolvable closed terms such as $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ and $\lambda x.\Omega$ are not equal because the latter is in whnf, a convergent term according to weak operational semantics, whereas the former is not.

Solvability as defined for λK still captures operational relevance in $\lambda\ell$ because solvability is coterminous with the world of full-reduction and nfs, and solvable terms under-approximate operational relevance in frameworks that reduce to hnf or whnf because nfs are included in hnfs and whnfs. Thus, some unsolvables are definite results (whnfs), as illustrated by $\lambda x.\Omega$. Abramsky reconciles unsolvability with divergence (not having whnf) by imposing a notion of order on unsolvable terms. Solvable terms and unsolvable terms of order greater than zero converge. Unsolvables of order zero diverge but can be consistently equated. Moreover, the $\lambda\ell$-calculus has a canonical model D that is a solution to the domain equation $D \cong [D \to D]_\perp$. However, the theory induced by D is not sensible nor semi-sensible: it equates solvable with unsolvable terms.

The lambda-value calculus was introduced by Plotkin (1975) as the calculus that corresponds to a strict call-by-value functional programming language whose operational semantics is given by the SECD machine (Landin, 1964). The *pure* version of the calculus, λ_V, is obtained by removing constants and δ-rules. Broadly, the β-rule changes to the β_V-rule which adds the side condition that operands in redices must be *values* (*i.e.*, non-applications). The calculus enjoys substitutivity, confluence, and there is a complete small-step reduction strategy $\to_V$ which delivers a value for a term. Plotkin proves that, for closed terms, the reduction strategy induced by the SECD machine coincides with the reflexive and transitive closure of $\to_V$.

Although in its applied version λ_V was motivated by the relation between call-by-value programming languages and the lambda calculus, pure λ_V is a full-blown calculus with good meta-theoretic properties. Certainly $\to_V$ is a weak (*i.e.*, does not go under lambda) strategy, but v-standardisation guarantees that complete full-reducing strategies exist which deliver definite results, in this case *v-normal forms* (v-nfs). Values are not definite results in λ_V, merely a side-condition for confluence. The calculus can be studied for its own sake, in particular the notion of solvability and sensible models.

The apparatus of $\lambda\ell$ and the notion of solvability has been carried successfully to what is called 'lazy λ_V' (Egidi *et al.*, 1991, 1992; Paolini & Ronchi Della Rocca, 1999). 'Lazy' is used only in the sense of 'weak' (abstractions are not reduced), losing the sense of non-strictness (operands are not reduced in redices) that is also present in $\lambda\ell$ and commonly associated with the word 'lazy'. In lazy λ_V, an abstraction is a value and therefore a definite

result which is not reduced by the reduction strategy induced by the SECD machine. There is a canonical solution to the domain equation $D \cong [D \to_\perp D]_\perp$ from which a domain H is constructed that is fully abstract with respect to SECD (Egidi *et al.*, 1991). The definition of solvability is not the same as in λK because it must now involve β_V-conversion. The equational definition involving the identity term is duly adapted by (Paolini & Ronchi Della Rocca, 1999) and called v-solvability, giving rise to concomitant notions of v-sensible and v-semi-sensible models.

The problem The notion of v-solvability does not accurately capture operational relevance in λ_V due to the following:

(i) The analogous to Proposition 4.1.1 in λ_V does not hold and therefore the various equational definitions of v-solvability are no longer equivalent.

The v-solvability under-approximates weak operational relevance, but fails to capture full operational relevance. Paolini and Ronchi Della Rocca (1999) introduce an order for v-unsolvables, in the spirit of $\lambda\ell$, to reconcile v-solvability and termination in SECD: all the v-unsolvables of order greater than zero converge and the v-unsolvables of order zero can be consistently equated. Consequently, the model H is not v-sensible nor v-semi-sensible.

In (Paolini & Ronchi Della Rocca, 1999, p. 4), full reduction and v-nfs are disregarded because there are v-nfs such as $U \equiv \lambda x.(\lambda y.\Delta)(x\, I)\Delta$, where $\Delta \equiv \lambda x.xx$, which are v-unsolvable. (Notice U is also a closed value and therefore a definite result in lazy λ_V.) And moreover, U (a v-nf) and $\lambda x.\Omega$ (a term without v-nf) are operationally equivalent.

Contrariwise, we interpret U as proof of (i) and that v-solvability is insufficient with respect to λ_V. In the setting of a programming language and SECD, input terms and contexts are closed, and U and $\lambda x.\Omega$ are operational equivalent. In λ_V, terms and contexts are open, full-reduction requires going under lambda (where free variables may pop up), and the objections against v-nfs disappear. Clearly, U and $\lambda x.\Omega$ can be distinguished according to SECD if we consider open contexts. For instance, $\mathbf{C}[U]$ delivers a stuck term (execution error) whereas $\mathbf{C}[\lambda x.\Omega]$ diverges when $\mathbf{C}[\,] \equiv [\,]\, y$. Execution errors and divergence are observationally distinguishable, as supported by the computability theorems and the introduction of type systems which aim at forbidding the former.

The interest of full reduction has been acknowledged in (Crégut, 1990). Typical applications are program optimisation by partial evaluation, and type checking in proof assistants (Grégoire & Leroy, 2002). Stuck terms have a meaning as open terms in a local scope for which reduction is deferred until given values for its free variables. But, as said before, values are not definite results in λ_V, merely a side-condition to preserve confluence by preserving potential divergence, an idea which appears informally in (Plotkin, 1975). The relevant notion of operational equivalence should also take into account the scope at which a stuck term pops up in a reduction sequence.

Contributions Our aim is to arrive at a 'standard theory' for λ_V that is the natural analogous of λK's. This means reinstating full reduction, v-nfs, and, stuck terms. The

guiding theme is the notion of solvability for λ_V. We first recall v-solvability in detail. Then we introduce the syntactic notion of quasi-solvability and quasi-v-solvability which rely on respective notions of needed and v-needed reduction, and are sufficient to capture preservation and genericity in their corresponding calculi. In λK, quasi-solvability coincides with solvability and needed normal forms coincide with head normal forms. In λ_V, having v-needed normal form is the characterisation of quasi-v-solvable terms. We introduce the notion of v-order of a quasi-v-unsolvable term and the λ_V-theory $\mathcal{H}_V$ which equates all quasi-v-unsolvables of equal v-order. We prove that the theory is a consistent extension of λ_V. We introduce ω-sensibility (*i.e.*, satisfying $\mathcal{H}_V$) and conjecture the existence of ω-sensible lattice models for λ_V.

The intuition is that that v-needed contexts characterise 'effective use' in that calculus. This provides a syntactic machinery for the goal of defining Böhm trees for λ_V.

We wrap up characterising all complete strategies in λK and λ_V. First, we give a novel proof of the Quasi-Leftmost Reduction Theorem (Hindley & Seldin, 2008) in terms of needed reduction. Then, we redeploy definitions and proofs for λ_V and prove the analogous Recursive v-Needed Reduction Theorem. This theorem broadens Plotkin's v-Standardisation Theorem showing that there are complete v-strategies which fail to be v-standard. We introduce the big-step definitions of two complete strategies, value normal order (which is v-standard) and value spine order (which is not v-standard). The former is the analogous to normal order in λK, and equivalent to evaluation relation $\mathbf{G}$ in (Ronchi Della Rocca & Paolini, 2004), and the latter is the most eager λ_V strategy that is complete. We conclude the chapter discussing that models for λ_V should reflect an operational distinction on terms based on the scope at which a stuck term pops up in their reduction sequence.

4.2 Preliminaries: solvability

The first definition of solvability appears in (Barendregt, 1971):

Definition 4.2.1. *A closed term M is solvable iff exist $N_1, \ldots, N_m$ with $m \geq 0$ such that $M\, N_1 \ldots N_m$ has a nf. An open term M is solvable if its closure $\lambda x_1 \ldots x_n.M$ with $\{x_1, \ldots, x_n\} = \mathrm{FV}(M)$ is solvable.*

A term is operationally characterised by providing arguments to it. Wadsworth (1976) provides an alternative definition:

Definition 4.2.2. *An arbitrary term M is solvable iff there exists a head context such that $\mathbf{C}[M]$ has a nf.*

A head context has the form $(\lambda x_1 \ldots x_n.[\,])N_1 \ldots N_m$, where $n, m \geq 0$. Typically, $\mathrm{FV}(M) \subseteq \{x_1, \ldots, x_n\}$. The previous definition is transformed into an equational definition in two equivalent ways (Wadsworth, 1976):

Definition 4.2.3. *An arbitrary term M is solvable iff there exists a head context $\mathbf{C}[\,]$ such that $\mathbf{C}[M] =_\beta I$.*

Definition 4.2.4. *An arbitrary term M is solvable iff there exists a head context $\mathbf{C}[\,]$ such that $\mathbf{C}[M] =_\beta P$ for arbitrary P.*

The equivalence between Definitions 4.2.3 and 4.2.4 is provided, in the ($\Longrightarrow$) direction, by appending the P as an additional operand in the head context, and in the ($\Longleftarrow$) direction, by letting $P \equiv I$. The equivalence between Definition 4.2.2 and Definitions 4.2.3 and 4.2.4 is provided, in the ($\Longrightarrow$) direction, by Proposition 4.1.1, and in the ($\Longleftarrow$) direction, by letting $P \in \mathsf{NF}$. Finally, solvable terms can be characterised syntactically (Wadsworth, 1976):

Definition 4.2.5. *An arbitrary term M is solvable iff M has a head normal form.*

The theory $\mathcal{H}$ equates all the *unsolvable* terms and is consistent. A theory $\mathcal{T}$ satisfying $\mathcal{H} \subseteq \mathcal{T}$ is said to be *sensible*. A theory that does not equate a solvable term with an unsolvable term is said to be *semi-sensible* (Barendregt, 1984).

Definition 4.2.3 is duly translated to λ_V in (Paolini & Ronchi Della Rocca, 1999)

Definition 4.2.6. *An arbitrary term M is v-solvable iff there exists a head context*

$$\mathbf{C}[\,] \equiv (\lambda x_1 \ldots x_n.[\,])\, N_1 \ \ldots \ N_m$$

where $n, m \geq 0$, $\{x_1, \ldots, x_n\} = \mathrm{FV}(M)$, and N_i closed values such that $\mathbf{C}[M] =_V I$.

The proviso that N_i must be values can be left out because it is implicit in $=_V$. The analogous to Proposition 4.1.1 in λ_V does not follow, the term U mentioned in the introduction is a counterexample. Thus, different equational definitions analogous to Definitions 4.2.1 and 4.2.4 are no longer equivalent in λ_V. In (Paolini & Ronchi Della Rocca, 1999), v-solvable terms are characterised syntactically as those having so-called v-head normal forms, and there exists a complete reduction strategy, the ahead machine, that delivers the v-head normal form of a term if it exists. However, the ahead machine is not a strategy of λ_V but of λK (*i.e.*, performs β-reduction, not β_V-reduction).

4.3 Quasi-solvability

We introduce the syntactic notion of quasi-(v-)solvability (quasi-solvability in λK and quasi-v-solvability in λ_V) which is based on a generalisation to λK and λ_V of the notion of *needed reduction* (TeReSe, 2003). Needed reduction reduces needed redices, which are those that are always reduced in any reduction sequence to an irreducible form. In λK, needed redices are those never occurring inside the operand of an operator, otherwise, the operator could eventually discard them. In λK, quasi-solvability coincides with the syntactic characterisation of solvability (existence of hnf). In λ_V, v-needed redices are those never occurring inside the value (a lambda abstraction) of some operator. They may occur, however, in an applicative context (a context compatible with application but not with abstraction) in the operand of some operator. As we shall see below, potential v-needed redices may be promoted (become v-redices) when contracting a v-needed redex occurring in the operand.

Definition 4.3.1 (Subterm Position)**.** *Function $\mathcal{F}$ annotates all the subterms of a term according to the function or argument position in which they occur.*

$$\begin{array}{rcl} \mathcal{F}(x) & = & x^{\mathrm{F}} \\ \mathcal{F}(\lambda x.B) & = & (\lambda x.\mathcal{F}(B))^{\mathrm{F}} \\ \mathcal{F}(M\,N) & = & (\mathcal{F}(M)\,\mathcal{A}(N))^{\mathrm{F}} \end{array}$$

$$\begin{array}{rcl} \mathcal{A}(x) & = & x^{\mathrm{A}} \\ \mathcal{A}(\lambda x.B) & = & (\lambda x.\mathcal{A}(B))^{\mathrm{A}} \\ \mathcal{A}(M\,N) & = & (\mathcal{A}(M)\,\mathcal{A}(N))^{\mathrm{A}} \end{array}$$

All the subterms of a term in argument position are in argument position. Positions are relative to the root term, for example, in $M(\lambda x.N)$ term N is in argument position relative to $M(\lambda x.N)$ but in function position relative to $(\lambda x.N)$. By abuse of language we say a term is function or argument if it is in function or argument position respectively.

Definition 4.3.2 (Needed Context)**.** *A needed context, $\mathbf{N}[\,]$, is defined as follows:*

$$\mathbf{N}[\,] \;::=\; [\,] \;\mid\; \mathbf{N}[\,]\,\Lambda \;\mid\; \lambda x.\mathbf{N}[\,]$$

Needed positions, represented by the hole, occur only at function position.

Definition 4.3.3 (v-Needed Context)**.** *A v-needed context, $\mathbf{N}_V[\,]$, is defined as follows:*

$$\begin{array}{rcl} \mathbf{N}_V[\,] & ::= & [\,] \mid \mathbf{N}_V[\,]\,\Lambda \mid \Lambda\,\mathbf{N}_V^A[\,] \mid \lambda x.\mathbf{N}_V[\,] \\ \mathbf{N}_V^A[\,] & ::= & [\,] \mid \mathbf{N}_V^A[\,]\,\Lambda \mid \Lambda\,\mathbf{N}_V^A[\,] \end{array}$$

Now v-needed positions, represented by the hole, occur only at function position or at argument positions not inside abstractions in argument position. To clarify when an argument position is a v-needed position consider the term $\lambda x.x(M(\lambda x.N))$. Both M and N are in argument position and inside an abstraction in function position (the root term), however, N is inside an abstraction $\lambda x.N$ in argument position and consequently is not in v-needed position.

Definition 4.3.4 (Needed Reduction)**.** *Needed reduction, $\to_{ne}^*$, is the reflexive and transitive closure of the following relation:*

$$\mathbf{N}[(\lambda x.B)N] \to_{ne} \mathbf{N}[[N/x]B] \quad \text{where } N \in \Lambda$$

A needed redex is a redex in needed position.

Definition 4.3.5 (v-Needed Reduction)**.** *v-needed reduction, $\to_{ne_V}^*$, is the reflexive and transitive closure of the following relation:*

$$\mathbf{N}_V[(\lambda x.B)V] \to_{ne_V} \mathbf{N}_V[[V/x]B] \quad \text{where } V \in \mathsf{Val}$$

A potential v-redex is an abstraction applied to some application which has a value. We say a potential v-redex is promoted to a v-redex when its operand is contracted resulting in a value. A (potential) v-needed redex is a (potential) v-redex in v-needed position.

We assume the reader familiar with the notions of *descendant* of a term and *residual* of a redex (TeReSe, 2003).

Notation 4.3.5.1. *For a context* $\mathbf{C}[\,]$*, the capture-avoiding substitution function* $[N/x](\mathbf{C}[\,])$ *is the usual capture-avoiding substitution function extended to contexts in the trivial way, where the hole* $[\,]$ *acts as a variable different from* x *and from any other variable in* $\mathbf{C}[\,]$*.*

Lemma 4.3.6 (Descendants of Needed Terms)**.** *(i) The descendants of terms in (v-)needed position are also in (v-)needed position.*

(ii) The residual of a v-needed redex is also a v-needed redex.

Proof. By the definition of (v-)needed position and capture-avoiding substitution for contexts. □

Definition 4.3.7 (v-Substitutive (Plotkin, 1975))**.** *A reduction relation* $\to_\rho^*$ *is v-substitutive iff* $M \to_\rho^* M'$ implies $\forall N \in \mathsf{Val}.\ [N/x]M \to_\rho^* [N/x]M'$*.*

Theorem 4.3.8 (β_V is v-Substitutive (Theorem 4.1 in Plotkin, 1975))**.**

$$\forall N \in \mathsf{Val}.\ M \to_{\beta_V}^* M' \text{ implies } [N/x]M \to_{\beta_V}^* [N/x]M'$$

Theorem 4.3.9 (v-Needed Reduction is v-Substitutive)**.**

$$M \to_{ne_V}^* M' \text{ implies } \forall N \in \mathsf{Val}.\ [N/x]M \to_{ne_V}^* [N/x]M'$$

Proof. By Lemma 4.3.6 and Theorem 4.3.8 □

Definition 4.3.10 (Quasi-(v-)Solvability)**.** *M is quasi-(v-)solvable (abbreviated q.(v-)s.) iff M has a (v-)needed normal form, that is, it exists M' such that $M \to_{ne_{(V)}}^* M'$ and $M' \not\to_{ne_{(V)}}$. The term M is quasi-(v-)unsolv-able (abbreviated q.(v-)u.) if it is not q.(v-)s.*

Observe that having needed normal forms coincides in λK with having head normal forms. Now we give a syntactic characterisation of *v-needed normal forms* (v-nenf).

Definition 4.3.11 (v-Weak Normal Form)**.** *A term is in v-weak normal form (v-wnf) if it does not have any v-redex except under abstraction. The set* VWNF *of v-wnfs is defined below, where* $\mathsf{Stuck_W}$ *is the set of stuck terms in v-wnf.*

$$\begin{array}{rcl} \mathsf{VWNF} & ::= & \mathsf{Val} \\ & | & \mathsf{Stuck_W} \\ \mathsf{Stuck_W} & ::= & (x\,\mathsf{VWNF})\{\mathsf{VWNF}\}^* \\ & | & ((\lambda x.\Lambda)\mathsf{Stuck_W})\{\mathsf{VWNF}\}^* \end{array}$$

Definition 4.3.12 (v-Needed Normal Form)**.** *A term is in v-needed normal form (v-nenf) if it does not have any v-needed redex. The set* VNeNF *of v-nenfs is defined below, where* $\mathsf{Stuck_{Ne}}$ *is the set of stuck terms in v-nenf.*

$$\begin{array}{rcl} \mathsf{VNeNF} & ::= & \lambda x.\mathsf{VNeNF} \\ & | & x \\ & | & \mathsf{Stuck_{Ne}} \\ \mathsf{Stuck_{Ne}} & ::= & (x\,\mathsf{VWNF})\{\mathsf{VWNF}\}^* \\ & | & ((\lambda x.\mathsf{VNeNF})\mathsf{Stuck_W})\{\mathsf{VWNF}\}^* \end{array}$$

The v-nenfs are terms that only have v-redices to the right of free variables and under abstractions in argument position. They are the λ_V analogous of hnfs in λK.

Definition 4.3.13 (v-Order of a Term)**.** *A term M is of v-order n iff*

$$n = max\{\ i \mid \exists N \equiv \lambda x_1 \ldots . \lambda x_i . B \text{ such that } M =_V N\ \}$$

*If such n is not defined (*i.e.*, n is unbounded) we say M is of v-order ω.*

Unless stated otherwise, we include the case $n = \omega$ when saying that a term M is of a generic v-order n.

Notation 4.3.13.1. *Given a set of terms* S*, we write* S_n *for the subset of terms with v-order at most n. For example, values consist of variables plus terms of v-order greater than 0:* $\mathsf{Val} = \mathsf{Var} \cup (\Lambda \setminus \Lambda_0)$.

The v-order of a q.v-u. term determines the number of operands that can be passed to it while still delivering a value.

Proposition 4.3.14. *Let M be q.v-u. of v-order $n > 0$ then M has a value, and $M\,V_1 \ldots V_i$ with $V_i \in \mathsf{Val}$ has a value for $i \leq n-1$ when n finite, or i arbitrary when $n = \omega$. The proof is immediate from the definition of v-order.*

4.4 Genericity lemma

Quasi-v-solvability is enough to prove the λ_V analogous of 'preservation of unsolvables by application and composition' (Wadsworth, 1976; Barendregt, 1984) and the Genericity Lemma (Barendregt, 1984).

Definition 4.4.1. *The context $\mathbf{C}^Z$ is zero-level iff it does not contain any trailing lambdas. This is $\mathbf{C}^Z[\ ] ::= [\ ] \mid \mathbf{C}[\ ]\,\Lambda \mid \Lambda\,\mathbf{C}[\ ]$ where the $\mathbf{C}[\ ]$ contexts are arbitrary. Zero-level contexts have the hole or an application at the top level.*

Definition 4.4.2 (Level of a Subterm)**.** *Let $N \subset M$, N is of level n iff*

$$n = max\{\ i \mid M \equiv \lambda x_1 . \lambda x_2 . \ldots . \lambda x_i . \mathbf{C}^Z[N]\ \}$$

The notion of descendants is generalised to contexts in the trivial way.

Lemma 4.4.3. *The descendant of a zero-level contexts is a zero-level context.*

Proof. Immediate by Definition 4.4.1. □

Lemma 4.4.4. *If M is q.v-u. of v-order n then it exists $R \equiv (\lambda x.B)V$ with $V \in \mathsf{Val}$ such that $M \to^*_{ne_V} \lambda x_1 \ldots \lambda x_n.\mathbf{C}^Z[R]$.*

Proof. By Definition 4.3.3, the context $\lambda x_1 \ldots . \lambda x_n.\mathbf{C}^Z[\,]$ is also a v-needed context and then R is a v-needed redex. Since $\to_{ne_V}$ commutes (Theorem 4.8.4), $\to^*_{ne_V}$ will eventually contract next the (residual of the) v-needed redex R in the (descendant of the) context $\lambda x_1 \ldots . \lambda x_n.\mathbf{C}^Z[\,]$, and then the lemma follows by Lemma 4.4.3. □

Lemma 4.4.5. *M is q.v-u. of v-order n iff, after some point in the infinite v-needed reduction sequence, all the v-redices contracted at each step are of level n. (Notice that this proposition only makes sense for finite n.)*

Proof. Let M be q.v-u. of v-order n, therefore exists $N \equiv \lambda x_1 \ldots . \lambda x_i.B$ such that $M =_V N$ and M's v-needed reduction sequence is infinite. By Lemma 4.4.4 we have $M \to^*_{ne_V} \lambda x_1 \ldots \lambda x_n.\mathbf{C}^Z[R]$ with $B \equiv \mathbf{C}^Z[R]$ and R a v-redex. From that point the reduction sequence is infinite with all reducts of that form for particular $\mathbf{C}^Z[R']$ with v-redex R'. □

Theorem 4.4.6 (Preservation of Quasi-v-Unsolvability)**.** *If M is q.v-u. of v-order n then, for any $N \in \mathsf{Val}$,*

(i) $M\,N$ is q.v-u. of v-order $n-1$ ('$-$' is subtraction in naturals, i.e., *$0 - n = 0$).*

(ii) $\lambda x.M$ is q.v-u. of v-order $n+1$.

(iii) $[N/x]M$ is q.v-u. of v-order n.

Proof. First we consider n finite. Now consider (iii):

Case $x \notin \mathrm{FV}(M)$ Trivial.

Case $x \in \mathrm{FV}(M)$ By Theorem 4.3.9 and Lemma 4.4.5 at some point in the infinite v-needed reduction sequence of $[N/x]M$ we find the reduct $\lambda x_1 \ldots x_n.[N/x](\mathbf{C}^Z[R])$. From the assumptions $x \in \mathrm{FV}(\mathbf{C}^Z[R])$, and $[N/x](\mathbf{C}^Z[R]) \equiv [N/x](\mathbf{C}^Z_1[x])$ because R is a v-redex (an application) and there is another zero-level context $\mathbf{C}^Z_1[\,]$ where x occurs free in $\mathbf{C}^Z_1[x]$. Clearly, $[N/x](\mathbf{C}^Z_1[x]) \equiv ([N/x](\mathbf{C}^Z_1[\,]))[N]$ and $[N/x](\mathbf{C}^Z_1[\,])$ is a zero-level context. By Lemma 4.4.5, $[N/x]M$ is q.v-u. of v-order n.

Now consider (i):

Case $n = 0$ It is immediate by Definition 4.3.10 because M is in v-needed position relative to $M\,N$.

Case $n > 0$ It holds because M reduces to an abstraction $(\lambda x.B)$ and with $[N'/x]B$ and $N =_V N'$ we are in case (iii).

Finally, (ii) is immediate by adding a lambda to M.

Second, we consider $n = \omega$. We generalise the proof of (iii) as follows: *does not exist* n such that $M \twoheadrightarrow^*_{ne_V} \lambda x_1 \dots x_n.\mathbf{C}^Z[x]$ and by Theorem 4.3.9 and Lemma 4.4.5 it is also not the case that $[N/x]M \twoheadrightarrow^*_{ne_V} \lambda x_1 \dots x_n.[N/x](\mathbf{C}^Z[x])$. Cases (i) and (ii) are proven similarly as before with the proviso $\omega - 1 = \omega$ and $\omega + 1 = \omega$. □

Theorem 4.4.7 (Genericity for λ_V)**.** *If N is q.v-u. of v-order n and $M\,N$ has a v-nf then $\forall V \in \mathsf{Val}_m$ $(m \geq n)$. $M\,N =_V M\,V$.*

Proof. This proof assumes the existence of a complete reduction relation, *recursive v-needed reduction*, which finds v-nfs via v-nenfs and whose introduction has been postponed to Theorem 4.8.7. The notions of v-needed positions and contexts are trivially generalised to *recursive v-needed positions and contexts*. All descendants of N are discarded at some point in the recursive v-needed reduction sequence by some operators in v-needed position, since N do not have v-nf and its descendants are q.v-u. of order n by Theorem 4.4.6(iii). This can only be the case if the descendants of N do not occur in recursive v-needed positions. The descendants of N may get applied to at most $n - 1$ operands. Otherwise, some descendant of N will deliver a quasi-v-unsolvable of v-order 0 which could occur in recursive v-needed position, making the sequence diverge. Consequently, any other term accepting at least $n - 1$ operands can be replaced for N. If $n = \omega$ then N may be applied to an arbitrary large number of operands and V has to be in Val_ω. □

4.5 $\beta_V\Omega_\omega$-reduction

In this section we introduce Ω_ω-reduction which, along with β_V-reduction, will generate the theory $\mathcal{H}_V$ that equates all the quasi-v-unsolvables of v-order n and is a consistent extension of λ_V (Section 4.6). We prove $\beta_V\Omega_\omega$ is Church-Rosser following the steps in (Barendregt, 1984, Section 15.2) save for the adoption of the Z-property technique (Van Oostrom, 2008). We assume the reader is familiar with the concept of notion of reduction in (Barendregt, 1984). Notions of reduction β_V and Ω_ω correspond to the β_V-rule (Plotkin, 1975) and relation Ω_ω in Definition 4.5.1 respectively.

Notation 4.5.0.1. *Let $\Omega_n \equiv \lambda x_1.\dots.\lambda x_n.\Delta\,\Delta$ where $\Delta \equiv (\lambda x.x\,x)$ and $n \geq 0$. We write Ω_ω for Ω_n with n arbitrary large.*

Definition 4.5.1 (Ω_ω-Reduction)**.** *Ω_ω-reduction, $\twoheadrightarrow^*_{\Omega_\omega}$, is the contextual, reflexive, and transitive closure of the relation*

$$\Omega_\omega = \{(M, \Omega_n) \mid M \text{ q.}v\text{-u. of } v\text{-order } n \text{ and } M \not\equiv \Omega_n\}$$

We use $\to_{\Omega_\omega}$ for the one-step Ω_ω-reduction relation.

The relation of Ω_ω-conversion, $=_{\Omega_\omega}$, is the symmetric closure of Ω_ω-reduction.

Definition 4.5.2 ($\beta_V\Omega_\omega$-Reduction)**.** *$\beta_V\Omega_\omega$-reduction, $\to^*_{\beta_V\Omega_\omega}$, is the reduction relation generated by the notions of reduction β_V and Ω_ω,* i.e.*, the reflexive and transitive closure of $\to_{\beta_V} \cup \to_{\Omega_\omega}$.*

The relation of $\beta_V\Omega_\omega$-conversion, $=_{\beta_V\Omega_\omega}$, is the symmetric closure of $\beta_V\Omega_\omega$-reduction.

Proposition 4.5.3. *Given two notions of reduction ρ_1 and ρ_2, if both are v-substitutive, the union $\rho_1 \cup \rho_2$ is also v-substitutive.*

Proof. By considering each ρ_1 or ρ_2 step individually in $\rho_1\rho_2$-reduction sequences. □

Theorem 4.5.4 ($\beta_V\Omega_\omega$ is v-Substitutive)**.** *By Proposition 4.5.3, it is enough to prove that Ω_ω is v-substitutive.*

Proof. Let $M \to_{\Omega_\omega} \Omega_n$, by Theorem 4.4.6 $[N/x]M$ is q.v-u. of v-order n for any $N \in \mathsf{Val}$. By the definition of Ω_ω-reduction, $[N/x]M \to_{\Omega_\omega} \Omega_n$, and trivially $\Omega_n \equiv [N/x]\Omega_n$. □

Definition 4.5.5 (Quasi-v-Solvably Equivalence)**.** *M and N are quasi-v-solvably equivalent, $M \sim_q N$, iff for every context $\mathbf{C}[\,]$ then $\mathbf{C}[M]$ is q.v-u. of v-order n iff $\mathbf{C}[N]$ is q.v-u. of v-order n. Clearly, $\sim_q$ is an equivalence relation.*

Lemma 4.5.6. *(i)* $M =_V N \Rightarrow M \sim_q N$

(ii) $M =_{\Omega_\omega} N \Rightarrow M \sim_q N$

Proof. We first consider (i). If $M =_V N$, by confluence of λ_V, there exist M' such that $M \to^*_{\beta_V} M' \,{}^*_{\beta_V}\!\leftarrow N$ and, since $\to^*_{\beta_V}$ is compatible, $\mathbf{C}[M] \to^*_{\beta_V} \mathbf{C}[M'] \,{}^*_{\beta_V}\!\leftarrow \mathbf{C}[N]$. Then both $\mathbf{C}[M]$ and $\mathbf{C}[N]$ are q.v-u. of v-order n iff $\mathbf{C}[M']$ is, and the lemma follows.

Now consider (ii). We assume the notion of *recursive v-needed context* in Theorem 4.8.7. Since $\sim_q$ is an equivalence relation, it is enough to show for $(M, \Omega_n) \in \Omega_\omega$ that $M \sim_q \Omega_n$:

Case n is finite By Definitions 4.3.10 and 4.3.13, $M =_V M'$ such that $M' \equiv \lambda x_1 \ldots \lambda x_n.B$ and B is q.v-u. of v-order 0. By (i), $\mathbf{C}[M] \sim_q \mathbf{C}[M']$. Let $\mathbf{C}[M'] \equiv \mathbf{C}'[B]$, then $\mathbf{C}[\Omega_n] \equiv \mathbf{C}'[\Omega_0]$. We consider $\mathbf{C}'[\,]$. If $\mathbf{C}'[\,]$ is not a recursive v-needed context, by Theorem 4.8.7 the M is discarded at some point and being q.v-u. of some v-order only depends on $\mathbf{C}'[\,]$, and then the lemma follows. Otherwise, by Lemma 4.4.4, $\mathbf{C}'[B] =_V \lambda x_1 \ldots \lambda x_m.\mathbf{C}^Z[R]$. Clearly, $B =_V \mathbf{C}^Z[R]$ because B is q.v-u. of v-order 0. Then, $\mathbf{C}'[\Omega_0] =_V \lambda x_1 \ldots \lambda x_m.\Omega_0$ and the lemma follows.

Case $n = \omega$ If $\mathbf{C}[\,]$ is a recursive v-needed context, both $\mathbf{C}[M]$ and $\mathbf{C}[\Omega_\omega]$ are q.v-u. of v-order ω. Otherwise, by Theorem 4.8.7 being q.v-u. only depends on $\mathbf{C}[\,]$. □

We assume the reader is familiar with the concept of disjoint subterms (no common symbol occurrences) of a given term (Barendregt, 1984, p. 25). By extension, we say a subterm is disjoint with a set of subterms when the subterm is disjoint with every element of the set.

Notation 4.5.6.1. *The symbol $\overset{R}{\to}_\rho$ denotes a ρ-reduction step where the ρ-redex R is contracted.*

Theorem 4.5.7 (Confluence of Ω_ω)**.** *Ω_ω-reduction is Church-Rosser.*

Proof. It is enough to prove that $(\to_{\Omega_\omega} \cup \equiv)$ has the diamond property. Consider

$$M \overset{U_1}{\to}_{\Omega_\omega} M_1 \text{ and } M \overset{U_2}{\to}_{\Omega_\omega} M_2$$

where U_1 and U_2 are the q.v-u. Ω_ω-redices contracted in each case. We consider the cases:

Case U_1 and U_2 are disjoint Trivial.

Case U_1 and U_2 overlap Let U_1 be the subterm q.v-u. of v-order n and U_2 the superterm q.v-u. of v-order m. The following diagram commutes because $\mathbf{C}_2[\Omega_n] \sim_q U_2$ by Lemma 4.5.6:

$$\begin{array}{ccc} M \equiv \mathbf{C}_1[U_2] \equiv \mathbf{C}_1[\mathbf{C}_2[U_1]] & \xrightarrow{\quad U_2 \quad} & M_2 \equiv \mathbf{C}_1[\Omega_m] \\ \Big\downarrow {\scriptstyle U_1} & & \Big\| \\ M_1 \equiv \mathbf{C}_1[\mathbf{C}_2[\Omega_n]] & \xrightarrow{\quad \mathbf{C}_2[\Omega_n] \quad} & \mathbf{C}_1[\Omega_m] \end{array}$$

□

Definition 4.5.8 (Maximal Ω_ω-Redices)**.** *A Ω_ω-redex is maximal iff it is not properly contained (is not a proper subterm) in another Ω_ω-redex.*

Theorem 4.5.9. *Every term has a unique Ω_ω-normal form.*

Proof. The maximal Ω_ω-redices are mutually disjoint. By replacing them by the appropriate Ω_ns, no new Ω_ω-redices are created, since $Q_n \sim_q \Omega_n$ for Q_n q.v-u. of order n. The Ω_ω-normal form is unique since Ω_ω-reduction is Church-Rosser. □

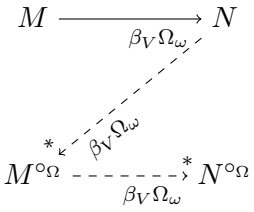
Notation 4.5.9.1. *We write $\Omega_\omega(M)$ for the Ω_ω-normal form of the term M.*

We assume the reader familiar with the notion of *complete development* M° of a term M (TeReSe, 2003).

Definition 4.5.10. *The complete Ω_ω-development $M^{\circ\Omega}$ of a term M consists of the complete development of the Ω_ω-normal form of M, that is, $M^{\circ\Omega} = (\Omega_\omega(M))^\circ$*

Theorem 4.5.11 (Confluence of $\beta_V\Omega_\omega$)**.** *$\beta_V\Omega_\omega$-reduction is Church-Rosser*

Proof. It is enough to prove that $\to_{\beta_V\Omega_\omega}$ has the Z property (Van Oostrom, 2008)



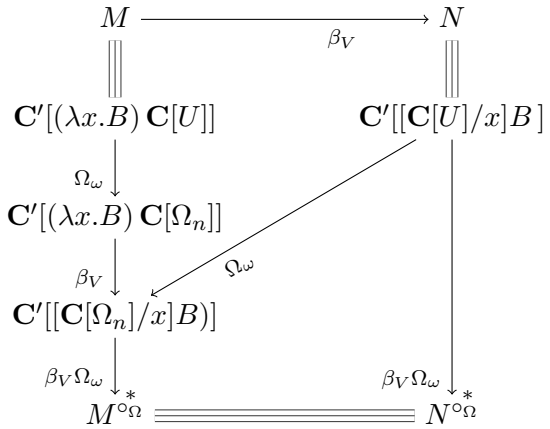
There are two cases $M \to_{\Omega_\omega} N$ and $M \to_{\beta_V} N$:

Case $M \to_{\Omega_\omega} N$ It follows that $\Omega_\omega(M) \equiv \Omega_\omega(N)$ and $M^{\circ\Omega} \equiv N^{\circ\Omega}$ therefore $N \to^*_{\beta_V\Omega_\omega} M^{\circ\Omega}$ and $M^{\circ\Omega} \to^*_{\beta_V\Omega_\omega} N^{\circ\Omega}$.

Case $M \to_{\beta_V} N$ Let R be the v-redex contracted in $M \to_{\beta_V} N$. Let S be the set of maximal Ω_ω-redices in M. If R is disjoint with S then $M^{\circ\Omega} \equiv N^{\circ\Omega}$ and the theorem follows as in the previous case. If R is not disjoint with some $U \in \mathsf{S}$ then consider sub-cases:

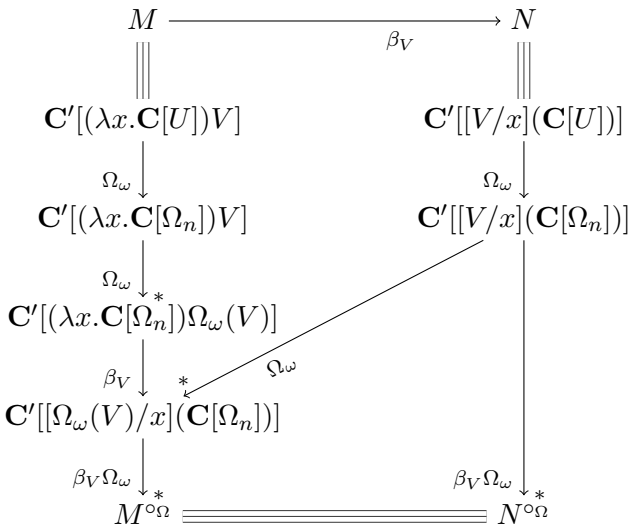
Case (i) $U \equiv \mathbf{C}[R]$ is q.v-u. of order n. Let R' be the contractum of R. By Lemma 4.5.6 we have $\Omega_\omega(\mathbf{C}[R]) \equiv \Omega_\omega(\mathbf{C}[R'])$ and therefore $\Omega_\omega(M) \equiv \Omega_\omega(N)$ and therefore $M^{\circ\Omega} \equiv N^{\circ\Omega}$.

Case (ii) $R \equiv (\lambda x.B)\,\mathbf{C}[U]$ is q.v-s. with B disjoint with S. Let n be the v-order of U. The following diagram commutes:



because $\mathbf{C}'[[\mathbf{C}[\Omega_n]/x]B)] \to^*_{\beta_V\Omega_\omega} M^{\circ\Omega} \equiv N^{\circ\Omega}$ since B and S are disjoint.

Case (iii) $R \equiv (\lambda x.\mathbf{C}[U])V$ is q.v-s. with $V \in \mathsf{Val}$ not necessarily disjoint with S. Let n be the v-order of U. The following diagram commutes:



because $\mathbf{C}'[(\lambda x.\mathbf{C}[\Omega_n])\Omega_\omega(V)] \to_{\beta_V} \mathbf{C}'[[\Omega_\omega(V)/x](\mathbf{C}[\Omega_n])] \to_{\beta_V\Omega_\omega} M^{\circ\Omega} \equiv N^{\circ\Omega}$ since $\mathbf{C}[\Omega_n]$ and $\mathsf{S} \setminus \{U\}$ are disjoint and by v-substitutivity of Ω_ω. The derivation

$$\mathbf{C}'[[V/x](\mathbf{C}[\Omega_n])] \to^*_{\Omega_\omega} \mathbf{C}'[[\Omega_\omega(V)/x](\mathbf{C}[\Omega_n])]$$

follows by an immediate lemma about substitution similar to Proposition 2.1.17(ii) in (Barendregt, 1984). □

4.6 The theory $\mathcal{H}_V$

We assume the reader is familiar with Chapters 4 and 16 of (Barendregt, 1984). We generalise to λ_V some of the notions related to λK.

Definition 4.6.1 (Consistency of a Theory)**.** *Let $\mathcal{T}$ be a theory with equations as formulae. $\mathcal{T}$ is consistent, Con($\mathcal{T}$), iff $\mathcal{T}$ does not prove every closed equation.*

Definition 4.6.2 (λ_V-Theory)**.** *Let $\mathcal{T}$ be a set of closed equations.*

(i) $\mathcal{T}^{+_V}$ is the set of closed equations provable in λ_V+$\mathcal{T}$.

(ii) $\mathcal{T}$ is a λ_V-theory if it is consistent and $\mathcal{T} = \mathcal{T}^{+_V}$.

Proposition 4.6.3. *The set of closed equations provable in λ_V (λ_V from now on for short) is a λ_V-theory.*

Proof. Elementary since λ_V is the trivial λ_V-theory (the one where $\mathcal{T} = \emptyset$) and consistency is entailed by confluence in the λ_V-calculus. □

We now define $\mathcal{H}_V$ as the λ_V-theory equating all the q.v-u. of equal v-order.

Definition 4.6.4 (Theory $\mathcal{H}_V$). *(i)* $\mathcal{H}_{v0} = \{M = N \mid M, N$ *are q.v-u. of equal v-order*$\}$

(ii) $\mathcal{H}_V = \mathcal{H}_{v0}^{+V}$

(iii) A λ_V*-theory* $\mathcal{T}$ *is* ω*-sensible iff* $\mathcal{H}_V \subseteq \mathcal{T}$.

The name ω-sensibility comes from the presence of ω equivalence classes in $\mathcal{H}_{v0}$. Now we prove that $\mathcal{H}_V$ is consistent via confluence of $\beta_V\Omega_\omega$-reduction.

Lemma 4.6.5 ($\beta_V\Omega_\omega$-Reduction Generates $\mathcal{H}_V$)**.**

$$\mathcal{H}_V \vdash M = N \quad \text{iff} \quad M =_{\beta_V\Omega_\omega} N$$

Proof($\Longrightarrow$) If $\mathcal{H}_{v0} \vdash M = N$ then $M \to_{\Omega_\omega} \Omega_n$ and $M \to_{\Omega_\omega} \Omega_n$ because both M and N are q.v-u. of v-order n. Consequently, for all axioms $M = N$ of $\mathcal{H}_V$ we have $M =_{\beta_V\Omega_\omega} N$ and the result follows.

($\Longleftarrow$) Each $\to_{\beta_V}$ or $\to_{\Omega_\omega}$ reduction step is provable in $\mathcal{H}_V$. The same holds for $=_{\beta_V\Omega_\omega}$. □

Theorem 4.6.6 (Consistency of $\mathcal{H}_V$)**.** Con($\mathcal{H}_V$)

Proof. By Lemma 4.6.5 and by Theorem 4.5.11. □

4.7 Completeness of needed reduction

Needed positions in λK are in relation to the Quasi-Leftmost Reduction Theorem (Hindley & Seldin, 2008, Theorem 3.22). This theorem is the most general characterisation of complete strategies of λK. In words, the theorem states that a reduction strategy which eventually contracts the leftmost redex is complete, *i.e.*, finds the normal form of a term if it exists. In this section we prove that needed reduction $\to_{ne}$ is complete. This result is equivalent to the Quasi-Leftmost Reduction Theorem. The intuitions and technical machinery will be redeployed in Section 4.8 to prove completeness of v-needed reduction $\to_{neV}$. Recall that in λK needed and non-needed positions are synonyms of function and argument positions respectively. We use the terminology interchangeably.

Lemma 4.7.1. *Contracting non-needed redices does not create any needed redices.*

Proof. If the redices occur in argument position, their contractum will remain in argument position. Some argument redices may be discarded, replicated, or created. However, the number of function redices remains the same by definition. □

We adapt the definition of projection of a reduction step over a redex (TeReSe, 2003). As introduced in page 69, we annotate reduction steps with the redex that is contracted.

Definition 4.7.2. *The projection of $\xrightarrow{R}_\rho$ over the ρ-redex S, written $\xrightarrow{R/S}_\rho$, is the ρ-reduction sequence obtained by contracting the residuals of R after contracting S. We append projections with *, $^+$, or none to indicate respectively whether the reduction sequence consists of either zero-or-more-steps, one-or-more-steps, or one-step. We also omit the notion of reduction ρ when it is clear from the context.*

Notice that the ρ-redices contracted in a projection $\xrightarrow{R/S}{}^*_\rho$ are always disjoint.

Lemma 4.7.3 (Postponement of Non-Needed Reduction)**.** *Given a reduction sequence there exists an equivalent reduction sequence where non-needed reduction is postponed to the end of the sequence. (Two reduction sequences are equivalent if both start with the same term M and both end with the same term N for arbitrary M and N.)*

Proof. Let $M \xrightarrow{A} M' \xrightarrow{F} N$ be the double-step reduction sequence where A and F are argument and function redices respectively. F' is the function redex such that $\xrightarrow{F}$ coincides with $\xrightarrow{F'/A}$. By Lemma 4.7.1, F' is unique. The following diagram illustrates:

$$\begin{array}{ccc} M & \xrightarrow{\quad A \quad} & M' \\ \Big\downarrow{\scriptstyle F'} & & \Big\downarrow{\scriptstyle F \equiv F'/A} \\ M'' & \xrightarrow{\quad A/F' \quad}{}^* & N \end{array}$$

Observe that all redices (if any) contracted in the sequence $\xrightarrow{A/F'}{}^*$ are disjoint. The function redices are contracted first, obtaining a reduction sequence equivalent to $M \xrightarrow{A} M' \xrightarrow{F} N$ in which non-needed reduction is postponed to the end. For any arbitrary reduction sequence we repeat the swapping operation, postponing all the non-needed steps. Notice that when swapping redices A and F we do not recurse the swapping operation over $\xrightarrow{A/F'}{}^*$ but trivially pick the function redices first, since all of them are disjoint. No reasoning by induction is needed over $\xrightarrow{A/F'}{}^*$. □

Lemma 4.7.4. *Needed redices have exactly one residual.*

Proof. If the contracted redex is in argument position, by Lemma 4.7.1 the number of function redices will remain the same and everyone of them have exactly one descendant. If the contracted redex is in function position, new function redices may appear, but the already existing ones (others than the one being contracted) cannot be discarded nor replicated, because function redices never occur in the argument of any redex. Again, everyone of them have exactly one descendant. □

Lemma 4.7.5 (Needed Reduction Commutes)**.** *Let M be a term with function redices F_1 and F_2. Contracting them in any order delivers the same term.*

Proof. Consider the diagram

$$\begin{array}{ccc} M & \xrightarrow{F_1} & M' \\ \downarrow{\scriptstyle F_2} & & \downarrow{\scriptstyle F_2/F_1} \\ M'' & \xrightarrow{F_1/F_2} & N. \end{array}$$

By Lemma 4.7.4 there is exactly one residual of any function redex. By substitutivity, the above diagram commutes. □

Remark 4.7.5.1. *New function redices may appear after the contraction of a function redex. Clearly, the newly created ones cannot commute with the one creating them. However, for the existing needed redices in the term (at a particular moment in the reduction sequence) the order in which they (or their residuals) are contracted does not matter because they commute.*

Theorem 4.7.6 (Needed Reduction)**.** *Contracting all the function redices in any order delivers a hnf if it exists.*

Proof. By Lemmas 4.7.1 and 4.7.3, for any reduction sequence ending in hnf there exists an equivalent one which contracts function redices first. Once the function redices are contracted, a hnf is reached. The hnf may have argument redices but contracting them is unnecessary. By Lemma 4.7.5, the order in which the function redices are contracted does not matter because they commute. □

The set of normal forms NF was defined in Section 2.1. Normal forms have the shape

$$\lambda x_1 \ldots x_n.x\, N_1\, \ldots\, N_m$$

where $n, m \geq 0$, x is called the *head variable*, and $N_i \in \mathsf{NF}$. Head normal forms have a similar shape save for $N_i \in \Lambda$.

Definition 4.7.7 (Degree of a Normal Form)**.** *The degree of a normal form, $\mathcal{D}(N)$, is recursively defined as follows:*

$$\begin{array}{rcll} \mathcal{D}(\lambda x_1 \ldots x_n.x) & = & 0 & n \geq 0 \\ \mathcal{D}(\lambda x_1 \ldots x_n.x\, N_1\, \ldots\, N_m) & = & 1 + max\{\mathcal{D}(N_i) \mid 0 \leq i \leq m\} & n, m \geq 0 \end{array}$$

Theorem 4.7.8 (Recursive Needed Reduction)**.** *A reduction strategy that eventually contracts the function redices and, recursively, eventually contracts the function redices relative to the arguments of the head variable, finds the normal form if it exists.*

Proof. For a term $M =_\beta N$ with $N \in \mathsf{NF}$ we proceed by induction on $\mathcal{D}(N)$. By Theorem 4.7.6, contracting needed redices delivers a hnf which is β-equivalent to N. The base case $\mathcal{D}(N) = 0$ is trivial. By the induction hypothesis, recursive needed reduction delivers the normal forms of head variable's arguments, and the final result is N. □

Remark 4.7.8.1. *The Recursive Needed Reduction Theorem is equivalent to the Quasi-Leftmost Reduction Theorem* (Hindley & Seldin, 2008)*. The function redices of M are in the leftmost segment and, inductively on $\mathcal{D}(N)$, so are the function redices relative to the leftmost N_i which is not in nf.*

A complete strategy has to contract all needed redices to get a nf. By Lemma 4.7.5 needed redices can be contracted in any order, as far as they are eventually contracted. Theorem 4.7.8 characterises the normalising reduction sequences in the most general way: no normalising reduction sequence exists without recursively contracting the needed redices.

4.8 Completeness of v-needed reduction

We prove the λ_V analogous of Theorem 4.7.8 in similar fashion as in the previous section. The main difference is that v-needed and non-v-needed (Definition 4.3.3) do not correspond now to function and argument. Fortunately, however, the proof is almost identical save for the v qualifier.

The notion of potential v-needed redex has been introduced in Definition 4.3.5.

Lemma 4.8.1. *Contracting non-v-needed redices does not create any (potential) v-needed redex.*

Proof. If the non-v-needed redices occur in some position which is under an argument abstraction their contractum will remain in that position. Some (potential) non-v-needed redices may be discarded, replicated or created. However, the number of (potential) v-needed redices remains the same by definition. □

Remark 4.8.1.1. *Non-v-needed contraction is not enough to promote a potential v-needed redex because contraction takes place under an abstraction in argument position,* i.e.*, inside a term which is already a value. Only the v-needed redices which are in argument position can promote potential v-needed redices. We say that a v-needed redex in argument position blocks a potential v-redex when it has a value and it occurs as the operand of an abstraction. Only v-needed redices in argument position which are not under an abstraction in argument position can block a potential v-needed redex. In other words, the blocking v-needed redices are in an applicative context.*

Lemma 4.8.2 (Postponement of Non-v-Needed Reduction)**.** *Given a v-reduction sequence there exists an equivalent v-reduction sequence where non-v-needed reduction is postponed to the end of the sequence.*

Proof. Let $M \xrightarrow{U} M' \xrightarrow{D} N$ be the double-step reduction sequence where U and D are non-v-needed and v-needed redices respectively. D' is the v-needed redex such that $\xrightarrow{D}$ coincides with $\xrightarrow{D'/U}$. By Lemma 4.8.1, D' is unique. The following diagram illustrates:

$$\begin{array}{ccc} M & \xrightarrow{U} & M' \\ \Big\downarrow{\scriptstyle D'} & & \Big\downarrow{\scriptstyle D \equiv D'/U} \\ M'' & \xrightarrow{U/D'}{}^{*} & N \end{array}$$

Observe that all v-redices (if any) contracted in the sequence $\xrightarrow{U/D'}{}^{*}$ are disjoint. The v-needed redices are contracted first, obtaining a v-reduction sequence equivalent to $M \xrightarrow{U} M' \xrightarrow{D} N$ in which non-v-needed reduction is postponed to the end. For any arbitrary v-reduction sequence we repeat the swapping operation, postponing all the non-v-needed steps. Notice that when swapping v-redices U and D we do not recurse the swapping operation over $\xrightarrow{U/D'}{}^{*}$ but trivially pick the v-needed redices first, since all of them are disjoint. No reasoning by induction is needed. □

Lemma 4.8.3. *(Potential) v-needed redices have exactly one descendant.*

Proof. There are three cases to consider:

(i) If the contracted v-redex is under an abstraction in argument position, *i.e.*, is in non-v-needed position, by Lemma 4.8.1 the number of (potential) v-needed redices remains the same and every one of them has exactly one descendant.

(ii) If the contracted v-redex is in argument position but not under an abstraction in argument position, *i.e.*, is in v-needed position, that redex may block a potential v-needed redex that will be promoted. New (potential) v-needed redices may appear in argument position but the already existing ones (including the potential v-redex promoted to an actual v-redex) have exactly one descendant.

(iii) If the contracted v-redex is in function position, new (potential) v-redices may appear but the already existing ones (other than the one being contracted) cannot be discarded nor replicated because (potential) v-needed redices never occur in the operand of any v-redex. All the already existing (potential) v-needed redices have exactly one descendant. □

Lemma 4.8.4 (v-Needed Reduction Commutes)**.** *Let M be a term with v-needed redices D_1 and D_2. Contracting them in any order delivers the same term.*

Proof. Consider the diagram

$$\begin{array}{ccc} M & \xrightarrow{D_1} & M' \\ \Big\downarrow{\scriptstyle D_2} & & \Big\downarrow{\scriptstyle D_2/D_1} \\ M'' & \xrightarrow{D_1/D_2} & N. \end{array}$$

By Lemma 4.7.4 there is exactly one residual of any v-needed redex. By v-substitutivity of $\rightarrow^*_{ne_V}$, the above diagram commutes. □

Remark 4.8.4.1. *New v-needed redices may appear after the contraction of a v-needed redex. In particular, already existing potential v-needed redices may be promoted. As in the proof of Lemma 4.8.2, the newly created ones cannot commute with the one creating or promoting them. However, for the existing v-needed redices in the term (at a particular moment in the v-reduction sequence) the order in which they (or their residuals) are contracted does not matter because they commute.*

Theorem 4.8.5 (v-Needed Reduction)**.** *Contracting all the v-needed redices in any order delivers a v-nenf if it exists.*

Proof. By Lemmas 4.8.1 and 4.8.2, for any v-reduction sequence ending in v-nenf there exists an equivalent one which contracts v-needed redices first. Once the v-needed redices are contracted, a v-nenf is reached. The v-nenf may have non-v-needed redices but contracting them is unnecessary. By Lemma 4.8.4, the order in which the v-needed redices are contracted does not matter because they commute. □

The set of v-normal forms VNF is defined by the EBNF grammar:

$$\begin{array}{rcl} \mathsf{VNF} & ::= & \lambda x.\mathsf{VNF} \\ & | & x \\ & | & \mathsf{Stuck_V} \\ \mathsf{Stuck_V} & ::= & (x\,\mathsf{VNF})\{\mathsf{VNF}\}^* \\ & | & ((\lambda x.\mathsf{VNF})\mathsf{Stuck_V})\{\mathsf{VNF}\}^* \end{array}$$

The v-normal forms have the shape $\lambda x_1 \ldots x_m.H\,N_1\,\ldots\,N_n$ where $n, m \geq 0$, H is called the head term, and $N_i \in \mathsf{VNF}$. Head terms can be either variables or blocks, in terminology of (Ronchi Della Rocca & Paolini, 2004), of the form $(\lambda z.\mathsf{VNF})\mathsf{Stuck_V}$. The v-nenfs have a similar shape, save for the body in the block which is also a v-nenf, its argument is in $\mathsf{Stuck_{Ne}}$ and $N_i \in \mathsf{VWNF}$.

Definition 4.8.6 (Degree of a v-Normal Form)**.** *The degree of a v-normal form, $\mathcal{D}_V(N)$, is recursively defined as follows:*

$$\begin{array}{rcl} \mathcal{D}_V(\lambda x_1 \ldots x_n.x) & = & 0 \\ \mathcal{D}_V(\lambda x_1 \ldots x_n.x\,N_1\,\ldots\,N_m) & = & 1 + max\{\mathcal{D}_V(N_1), \ldots, \mathcal{D}_V(N_m)\} \\ \mathcal{D}_V(\lambda x_1 \ldots x_n.(\lambda z.B)S) & = & 1 + max\{\mathcal{D}_V(B), \mathcal{D}_V(S))\} \\ \mathcal{D}_V(\lambda x_1 \ldots x_n.((\lambda z.B)S)N_1\,\ldots\,N_m) & = & \\ & & 1 + max\{\mathcal{D}_V(B), \mathcal{D}_V(S), \mathcal{D}_V(N_1), \ldots, \mathcal{D}_V(N_m)\} \\ & & \text{where } n, m \geq 0 \end{array}$$

Theorem 4.8.7 (Recursive v-Needed Reduction)**.** *A reduction strategy that eventually contracts the v-needed redices and, recursively, eventually contracts the v-needed redices relative to the arguments of the head term (and its body and argument if it is a block), finds the v-normal form if it exists.*

Proof. For a term $M =_\beta N$ with $N \in \mathsf{VNF}$ we proceed by induction on $\mathcal{D}_V(N)$. By Theorem 4.8.5, contracting v-needed redices delivers a v-nenf which is β_V-equivalent to N. The base case $\mathcal{D}_V(N) = 0$ is trivial. By the induction hypothesis, recursive v-needed reduction delivers the v-normal forms of head term's arguments (and its body and argument if it is a block), and the final result is N. □

Remark 4.8.7.1. *The Recursive v-Needed Reduction Theorem is the analogous in λ_V of the Quasi-Leftmost Reduction Theorem in λK. The notions of v-needed positions and contexts are generalised in the trivial way to that of recursive v-needed positions and contexts. These notions are used in the proofs of Theorem 4.4.7 and Lemma 4.5.6.*

A complete v-strategy has to contract all v-needed redices to get a v-nf. By Lemma 4.8.4 v-needed redices can be contracted in any order, as far as they are eventually contracted. Theorem 4.8.7 characterises the v-normalising reduction sequences in the most general way: no v-normalising reduction sequence exists without recursively contracting the v-needed redices.

4.9 Leftmost, standard, needed, and spine

In this section we show that v-leftmost strategies are not complete, that v-standard strategies are complete, but there are complete v-strategies which are not v-standard, and that all complete strategies are v-needed. We define big-step-wise the full-reducing, complete strategies which we call value normal order and value spine order. The former is the v-standard strategy analogous to normal order in λK. We show its big-step definition which corresponds to evaluation relation **G** in (Ronchi Della Rocca & Paolini, 2004). The latter is the v-needed, non-v-standard, spine version of the former, and is analogous to hybrid normal order (Sestoft, 2002) and `byName` (Paulson, 1996, p. 390) in λK. Value spine order is the most eager while complete strategy in λ_V.

The notion of v-leftmost is a direct translation to λ_V of the definition of leftmost strategy (Curry & Feys, 1958), which does not capture completeness.

Definition 4.9.1 (v-Leftmost Strategy)**.** *A v-leftmost strategy contracts the leftmost v-redex first.*

Consider the term $L \equiv (\lambda x.y)((\lambda x.\lambda y.\Omega)(I\,I))$ where $\lambda y.\Omega$ is a q.v-u. value. The leftmost v-redex is Ω. The other redex-looking subterms are potential v-redices because their operands are not values. Contracting Ω leaves the term unchanged and thus v-leftmost

reduction diverges. However, observe that the following v-reduction sequence (hereafter referred to as StdSeq) terminates:

$$\begin{array}{lll}(\lambda x.y)((\lambda x.\lambda y.\Omega)(I\,I)) & \to_{\beta_V} & (\lambda x.y)((\lambda x.\lambda y.\Omega)I)\\ & \to_{\beta_V} & (\lambda x.y)(\lambda y.\Omega)\\ & \to_{\beta_V} & y\end{array}$$

The seminal paper (Plotkin, 1975) introduced the small-step v-strategy $\to_V$ which we adapt to pure λ_V,

$$\frac{N \in \mathsf{Val}}{(\lambda x.B)N \to_V [N/x]B}\ (\beta_V)$$

$$\frac{M \to_V M'}{M\,N \to_V M'\,N}\ (\mu_V) \qquad \frac{M \in \mathsf{Val} \quad N \to_V N'}{M\,N \to_V M\,N'}\ (\nu_V)$$

and gave a definition of v-standard reduction sequences which we adapt to pure λ_V in the following definition.

Definition 4.9.2 (v-Standard Reduction Sequence (Plotkin, 1975, p. 137))**.** *A v-standard reduction sequence (v-s.r.s.) is defined inductively as follows:*

(i) Any variable x is a v-s.r.s.

(ii) If $N_2, ..., N_k$ is a v-s.r.s. and $N_1 \to_V N_2$, then $N_1, ..., N_k$ is a v-s.r.s.

(iii) If $N_1, ..., N_k$ is a v-s.r.s. then $\lambda x.N_1, ..., \lambda x.N_k$ is a v-s.r.s.

(iv) If $M_1, ..., M_j$ and $N_1, ..., N_k$ are v-s.r.s. then $M_1\,N_1, ..., M_j\,N_1, ..., M_j\,N_k$ is a v-s.r.s.

The $\to_V$ v-strategy is v-standard by (ii) but it is a weak strategy (does not reduce abstractions) and is not v-normalising. Cases (i), (iii), and (iv) recursively specify how to lift a *whole* v-s.r.s. to an arbitrary context. Informally, the v-s.r.s. $N_2, ..., N_k$ is lifted to $\mathbf{C}[N_2], ..., \mathbf{C}[N_k]$. A step $N_1 \to_V \mathbf{C}[N_2]$ can be prepended to the sequence, but had the new step required a new context $\mathbf{D}[\,]$ (*i.e.*, $\mathbf{D}[N_1] \to_V \mathbf{C}[N_2]$) then the rest of the sequence would have to be lifted to that context too, $\mathbf{D}[N_1], \mathbf{D}[\mathbf{C}[N_2]], ..., \mathbf{D}[\mathbf{C}[N_k]]$. Consequently, a $\to_V$ step cannot be prepended with the source term inside an arbitrary context, otherwise, any v-reduction sequence would be v-standard and Definition 4.9.2 would be void.

The v-Standardisation Theorem (Plotkin, 1975, Theorem 4.3) characterises completeness in λ_V. In words, it says that for any v-reduction sequence, there exists an equivalent v-standard one. In particular, if some term has a v-nf, there exists a v-s.r.s. ending in it. Observe that StdSeq above is v-standard because every step is a $\to_V$ step.

The small-step evaluation relation $\mathbf{G}$ in (Ronchi Della Rocca & Paolini, 2004) is a v-standard complete strategy delivering a v-nf. We present the Hilbert-style inference

rules of its big-step counterpart, which we call *value normal order* and denote by $\Downarrow_{vn}$:

$$\frac{}{x \Downarrow_{vn} x}\ (\textsc{Var}) \qquad \frac{B \Downarrow_{vn} B'}{\lambda x.B \Downarrow_{vn} \lambda x.B'}\ (\textsc{Abs})$$

$$\frac{M \Downarrow_{pv} M' \quad N \Downarrow_{pv} N' \quad (M' \equiv \lambda x.B \wedge N' \in \mathsf{Val}) \quad [N'/x]B \Downarrow_{vn} B'}{M\,N \Downarrow_{vn} B'}\ (\textsc{Con})$$

$$\frac{M \Downarrow_{pv} M' \quad N \Downarrow_{pv} N' \quad (M' \not\equiv \lambda x.B \vee N' \notin \mathsf{Val}) \qquad M' \Downarrow_{vn} M'' \quad N' \Downarrow_{vn} N''}{M\,N \Downarrow_{vn} M''\,N''}\ (\textsc{Stk})$$

Variables are already v-normal forms (rule Var). Abstractions are reduced to v-normal form (rule Abs). Operators and operands are reduced to v-weak normal forms (rules Con and Stk) by the weak *pure call-by-value* strategy (pv) discussed below. If the resulting application is a v-redex then the contractum is reduced to v-normal form (rule Con). Otherwise the application is a stuck term and then both the operator and the operand are reduced to v-normal form.

Pure call-by-value is a generalisation of the call-by-value strategy called eval_V in (Plotkin, 1975). Pure call-by-value takes into account free variables and stuck terms of pure λ_V and delivers a v-weak normal forms.

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

$$\frac{M \Downarrow_{pv} M' \quad N \Downarrow_{pv} N' \quad (M' \equiv \lambda x.B \wedge N' \in \mathsf{Val}) \quad [N'/x]B \Downarrow_{pv} B'}{M\,N \Downarrow_{pv} B'}\ (\textsc{Con})$$

$$\frac{M \Downarrow_{pv} M' \quad N \Downarrow_{pv} N' \quad (M' \not\equiv \lambda x.B \vee N' \notin \mathsf{Val})}{M\,N \Downarrow_{pv} M'\,N'}\ (\textsc{Stk})$$

As noted by (Herbelin & Zimmermann, 2009), v-standard reduction sequences are not *unique*: for a term M, v-standardisation characterises some set of v-reduction sequences ending in M's v-nf. We show that this set does not contain *all* the complete v-reduction sequences. There are complete v-reduction sequences which are not v-standard. For example, consider the sequence (hereafter referred to as NeSeq):

$$\begin{aligned} & (\lambda z.(\lambda x.\lambda y.x)z\,t)I && (4.1)\\ \to_{\beta_V} \quad & (\lambda z.(\lambda y.z)t)I && (4.2)\\ \to_{\beta_V} \quad & (\lambda z.z)I && (4.3)\\ \to_{\beta_V} \quad & I && (4.4) \end{aligned}$$

Steps (4.2) and (4.3) are not $\to_V$ steps. Although sequences (4.1),(4.2), and (4.3),(4.4) are v-s.r.s., two arbitrary v-s.r.s.'s cannot be concatenated to give a complete v-reduction sequence, according to Definition 4.9.2, and thus NeSeq is not v-s.r.s.. However, NeSeq is a recursive v-needed reduction sequence as characterised by Theorem 4.8.7.

4.9.1 Spine strategies

A recursive v-needed strategy (complete v-strategy) can reduce abstractions provided that operands are reduced weakly, consequently the spine of the term can be traversed eagerly. We introduce the v-strategy *value spine order* denoted by $\Downarrow_{vs}$:

$$\frac{}{x \Downarrow_{vs} x}\ (\textsc{Var}) \qquad \frac{B \Downarrow_{vs} B'}{\lambda x.B \Downarrow_{vs} \lambda x.B'}\ (\textsc{Abs})$$

$$\frac{M \Downarrow_{vh} M' \quad N \Downarrow_{pv} N' \quad (M' \equiv \lambda x.B \wedge N' \in \mathsf{Val}) \quad [N'/x]B \Downarrow_{vs} B'}{M\,N \Downarrow_{vs} B'}\ (\textsc{Con})$$

$$\frac{M \Downarrow_{vh} M' \quad N \Downarrow_{pv} N' \quad (M' \not\equiv \lambda x.B \vee N' \notin \mathsf{Val}) \qquad M' \Downarrow_{vs} M'' \quad N' \Downarrow_{vs} N''}{M\,N \Downarrow_{vs} M''\,N''}\ (\textsc{Stk})$$

As before, variables are already v-normal forms (rule Var). Value spine order reduces abstractions to v-normal forms (rule Abs). But it uses *value head reduction* $\Downarrow_{vh}$ on the operator and pure call-by-value on the operand (rule Con). In the case of an stuck term it reduces operator and operand to v-normal form (rule Stk).

Value head reduction is defined by the rules:

$$\frac{}{x \Downarrow_{vh} x}\ (\textsc{Var}) \qquad \frac{B \Downarrow_{vh} B'}{\lambda x.B \Downarrow_{vh} \lambda x.B'}\ (\textsc{Abs})$$

$$\frac{M \Downarrow_{vh} M' \quad N \Downarrow_{pv} N' \quad (M' \equiv \lambda x.B \wedge N' \in \mathsf{Val}) \quad [N'/x]B \Downarrow_{vh} B'}{M\,N \Downarrow_{vh} B'}\ (\textsc{Con})$$

$$\frac{M \Downarrow_{vh} M' \quad N \Downarrow_{pv} N' \quad (M' \not\equiv \lambda x.B \vee N' \notin \mathsf{Val})}{M\,N \Downarrow_{vh} M'\,N'}\ (\textsc{Stk})$$

As value spine order, value head reduction traverses the spine of the term reducing the operands weakly using pure call-by-value. But now, in the case of stuck terms the operator and the operand are not fully reduced. Value head reduction delivers a v-nenf.

4.10 Intuition for models of λ_V

Given two terms M and N, assume the set of open contexts can be partitioned in three subsets according to their reduction under SECD ($\Downarrow_s$): both $\mathbf{C}_1[M]$ and $\mathbf{C}_1[N]$ have a value, both $\mathbf{C}_2[M]$ and $\mathbf{C}_2[N]$ get stuck, or both $\mathbf{C}_3[M]$ and $\mathbf{C}_3[N]$ diverge. Then M and N are operationally equivalent broadly conceived. However, lattice models for λ_V should reflect a finer notion of operational equivalence that distinguishes in $\mathbf{C}_2[M]$ and $\mathbf{C}_2[N]$ the scope at which a stuck term pops up. More precisely, consider $M \equiv (\lambda z.I)(x\,y)P$ and $N \equiv I((\lambda z.P)(x\,y))$ which are v-nfs if P is in v-nf. For those contexts $\mathbf{C}_1[\ ]$ for which

$(x\,y)$ becomes a value then $\mathbf{C}_1[M] \Downarrow_s P$ and $\mathbf{C}_1[N] \Downarrow_s P$. For those contexts $\mathbf{C}_1[\ ]$ for which $(x\,y)$ is stuck then $\mathbf{C}_2[M]$ and $\mathbf{C}_2[N]$ can be considered observationally different with regards to the scope at which the stuck $(x\,y)$ pops up. This suggests a meaning for v-nfs.

Addendum

4.11 Needed reduction

When we first wrote the manuscript on which Chapter 4 is built, we gave a syntactic characterisation of needed reduction in λK and λ_V which was inspired by the notion of neededness in (TeReSe, 2003, Section 9.2). Later on, we discovered that needed reduction was fully explored for λK in (Barendregt *et al.*, 1987). Thus, there exist some overlap between our Chapter 4 and the latter, in particular in relation to the classical lambda calculus. However, the notion of needed reduction in the lambda-value calculus in Chapter 4 is relevant and genuinely original.

4.12 Operational relevance in λ_V

We introduce a definition of operational relevance in lambda-value which considers the lambda-value normal forms as the definite results:

Definition 4.12.1. *A term M is operationally relevant in λ_V iff there exists a context $\mathbf{C}[\ ]$ such that $\mathbf{C}[M] \to^*_{\beta_V} N \in \mathsf{VNF}$, and it is not the case that for a term P belonging to a class of terms which are not β_V-equivalent to M, P could be substituted by M in $\mathbf{C}[\ ]$ without affecting the result,* i.e.*, it is not the case that $\mathbf{C}[P] \to^*_{\beta_V} N \in \mathsf{VNF}$.*

In words, a term is operationally relevant in λ_V iff it could be used effectively in λ_V in order to produce a v-normal form.

Along Chapter 4 we inquired whether the v-needed contexts were enough to characterise effective use in λ_V. In Section 4.3 we introduced the syntactic notion of quasi-v-solvability in the belief that it would capture operational relevance in lambda-value as in Definition 4.12.1. Unfortunately this is not the case, since there are terms in VNeNF which cannot be sent to a VNF. Consider the term $M \equiv x((\lambda y.\Omega)(z\,I))$. M is a VNeNF because the operand $(\lambda y.\Omega)(z\,I)$ is a VWNF (see Definition 4.3.11), *i.e.*, it only has redices under lambda abstraction, and the v-needed reduction will not hit the divergent body of $\lambda y.\Omega$ because $z\,I$ is a stuck term. However, the Ω could not be discarded by turning $z\,I$ into a value, nor it could be reduced to a VNF. Thus, M is not operationally relevant in λ_V.

The recursive v-needed redices characterise effective use and complete strategies in λ_V. Quasi-v-solvability can be defined in terms of β_V-reduction, and it is enough to prove the Genericity Lemma in λ_V(see Section 4.4). The λ_V-theory $\mathcal{H}_V$ equates the quasi-v-unsolvables of the same order in a consistent way (see Section 4.6). However, having a

VNeNF falls short as to capture operational relevance in λ_V, since some of the 'points' in $\mathcal{H}_V$ are not operationally relevant in λ_V. In a way, the 'intermediate normal forms' (*i.e.*, the λ_V analogous of the head normal forms) should be reduced further in order to filter terms as the M example above.

This fact claims for a refinement of our syntactic characterisation. In some ongoing work, we introduce a novel 'ribcage normal form' which takes this problem into account and only allows arbitrary terms in some 'rib ends', *i.e.*, in positions which are discardable according to λ_V reduction. We are developing a proof to show that the ribcage normal forms can be sent to a VNF by providing arguments to it. Providing arguments is what distinguish the relevant uses from the 'trivial uses' where a term would only be discarded and hence it could be replaced by a family of terms (see the Genericity Lemma in Section 4.4). We focus on the occurrences of what we call *stuck accumulators*, of the form $x\,\mathsf{Val}_1 \dots \mathsf{Val}_n$, inside the ribcage normal forms. We only allow a stuck accumulator to block a v-redex with operator $\lambda y.B$ when the B is reduced enough as for the application $(\lambda y.B)(x\,\mathsf{Val}_1 \dots \mathsf{Val}_n)$ to be *hereditarily solvable* by solving the stuck accumulators inside B. Term B also has to preserve the ability of the application $(\lambda x.B)(x\,\mathsf{Val}_1 \dots \mathsf{Val}_n)$ to be *hereditarily freezable* (*i.e.*, it could be kept as a stuck normal form) in order to avoid the issues that are brought up by terms similar to the $U \equiv (\lambda x.\Delta)(x\,I)\Delta$ in (Paolini & Ronchi Della Rocca, 1999). Our ribcage normal forms can be characterised in terms of β_V-reduction. The definition of the ribcage normal forms is rather intricate, and the details of this material are omitted here, since the proofs are not fully finished. A preliminary version has been presented in the *International Workshop on Domain Theory and Application*, Paris, 8-10 September 2014.

Part II

Full-Reducing Machines

5

On the Syntactic and Functional Correspondence between Hybrid (or Layered) Normalisers and Abstract Machines

El modo de dar una vez en el clavo es dar cien veces en la herradura.[1]

(Miguel de Unamuno)

We show how to connect the syntactic and the functional correspondence for normalisers and abstract machines implementing hybrid (or layered) reduction strategies, that is, strategies that depend on subsidiary sub-strategies. Many fundamental strategies in the literature are hybrid, in particular, many full-reducing strategies, and many full-reducing and complete strategies that deliver a fully reduced result when it exists. If we follow the standard program-transformation steps the abstract machines obtained for hybrids after the syntactic correspondence cannot be refunctionalised, and the junction with the functional correspondence is severed. However, a solution is possible based on establishing the shape invariant of well-formed continuation stacks. We illustrate the problem and the solution with the derivation of substitution-based normalisers for normal order, a hybrid, full-reducing, and complete strategy of the pure lambda calculus. The machine we obtain is a substitution-based, eval/apply, open-terms version of Pierre Crégut's full-reducing Krivine machine KN.

[1] To strike a nail once, you must strike the horseshoe a hundred times. (Translation by Great Thoughts Treasury `http://www.greatthoughtstreasury.com/node/517259`.)

5.1 Introduction

Figure 2.3 depicts the derivation path for semantic artefacts. Reduction-based and reduction-free normalisers (and intermediate abstract machines) are equivalent because the transformation steps are equivalence-preserving. Consequently, the artefacts implement the same reduction strategy. A search function is a simpler artefact which, although not strictly equivalent, is sufficient to characterise the structural operational semantics (Danvy *et al.*, 2011). It connects the structural and context-based semantics, recomposition is straightforward to add and, more importantly for us, the simplification and defunctionalisation of the search function reveals the continuation stack, which is not the case if the starting point is a whole implementation of the structural operational semantics that searches the input term, contracts the redex, and delivers the next reduct.

The Problem If we follow the standard program-transformation steps (Ager *et al.*, 2003b,a; Danvy & Nielsen, 2004; Danvy, 2005; Danvy & Millikin, 2008; Biernacka & Danvy, 2007; Danvy *et al.*, 2011) it is not possible to connect the syntactic and the functional correspondence for normalisers implementing 'hybrid' strategies. The correspondences and the connection have been successfully established for 'uniform' strategies such as call-by-name and call-by-value, and a functional correspondence between a big-step virtual machine and a reduction-free normaliser has been established for a hybrid strategy, namely, normal order (Ager *et al.*, 2003a).

We have borrowed the uniform/hybrid terminology from (Sestoft, 2002) where it is used informally. A strategy is *uniform* when it is defined as a *single function* that only depends on itself, *e.g.*, no other function occurs in the premisses of the inference rules of its natural semantics. In contrast, a strategy is *hybrid* (or layered) when it is defined as a *single function* that depends on (at least) another *subsidiary* strategy.[2] For example, the natural semantics of a hybrid $\Downarrow_h$ will have inference rules where a subsidiary $\Downarrow_s$ occurs in one or more premisses. Here is a possible example rule:

$$\frac{M \Downarrow_s M' \quad M' \Downarrow_h N}{M \Downarrow_h N} \text{ (Rule)}$$

In words, and reading relational notation functionally, Rule says that $\Downarrow_h$ reduces M to N by first reducing M to M' using $\Downarrow_s$ and then reducing M' to N recursively. The term M' is the point at which $\Downarrow_s$ stops and $\Downarrow_h$ resumes.

In many practical strategies (see Section 5.8), the subsidiary is employed by the hybrid to reduce some subterms *less* in order to uphold some properties. Which strategy, subsidiary or hybrid, is to start, continue, or resume the next reduction is clear in the semantics. In the syntactic correspondence, several semantic artefacts (subsidiaries and hybrid) are written in parallel. However, the refocusing and inlining-of-iterate-function

[2] We insist on 'single-function' to avoid confusion with definitions in *eval-readback style* (Section 5.3), a degenerate case of normalisation-by-evaluation where a strategy is defined as the composition of *two* single functions, *e.g.*, two natural semantics.

steps become context dependent, and the dispatcher of the abstract machine has to inspect the continuation stack (the arguments of value constructors that represent defunctionalised continuations) deeply to find out which strategy is to continue. This prevents the refunctionalisation of the machine. Defunctionalisation and refunctionalisation require 'shallow inspection' of the continuation stack (Ager *et al.*, 2003b). In this thesis we shall refer to this requirement as the *shallow inspection property.*

Hereafter, when we say that a strategy is 'hybrid' we will not only mean that the strategy is defined in hybrid *style*, as discussed before. We will use the name for strategies that are also hybrid *in nature*, that is, the subsidiaries are intrinsic to the strategy even if not written explicitly in a particular semantic definition. We define 'hybrid nature' formally and further discuss this point in Section 5.3. The technical solution we present applies to strategies in hybrid *style* independently of whether they are hybrid in nature or uniforms in a hybrid-style disguise. However, the hybrid style is the natural way of defining a hybrid strategy, and defining in uniform style a strategy that is hybrid in nature is a recipe for unnatural derivations, as simply illustrated by outermost reduction (Danvy & Johannsen, 2013) discussed in Section 5.8.

Why the problem is important Many fundamental strategies in the literature are hybrid, in particular, many *full-reducing* strategies, and many full-reducing and *complete* strategies. A strategy is full-reducing if its final results are normal forms, *i.e.*, terms with no redices. A full-reducing strategy is complete if it delivers a normal form when the input term has one. A strategy is incomplete if it diverges for some terms which have normal form. Full-reducing strategies are important in program optimisation by partial evaluation and type checking in proof assistants (Crégut, 1990).

Examples of hybrid strategies of the lambda calculus are normal order and head reduction (Barendregt, 1984), hybrid normal order and hybrid applicative order (Sestoft, 2002), strong reduction (Grégoire & Leroy, 2002) (similar to `byValue` in (Paulson, 1996, p.390) but with right to left reduction of applications in the subsidiary as we explain in Section 5.8), the ahead machine (Paolini & Ronchi Della Rocca, 1999) (which delivers so-called 'value head normal forms' in the pure lambda calculus), and the **G** reduction relation whose implementation as an instance of the principal reduction machine of the parametric lambda calculus realises an outermost normalising strategy of the pure lambda-value calculus (Ronchi Della Rocca & Paolini, 2004, chp.5). A hybrid strategy outside the lambda calculus is the outermost strategy for arithmetic expressions (Danvy & Johannsen, 2013). Of the above, normal order, hybrid normal order, **G**, and outermost reduction for arithmetic expressions are full-reducing and complete. Head reduction is complete with respect to head normal forms, strong reduction is complete within the strongly-normalising calculus where it is meant to be used, hybrid applicative order is full-reducing but incomplete, and so on. In contrast, examples of uniform strategies are call-by-name, call-by-value, applicative order,[3] and head spine (Sestoft, 2002; Barendregt *et al.*, 1987), the latter similar

[3]There is often a confusion with applicative order and call-by-value. Applicative order is a full-reducing but incomplete strategy of the pure lambda calculus that realises the idea of passing parameters in normal

to `headNF` in (Paulson, 1996, p.390). The natural semantics of some of the above strategies, uniform and hybrid, can be found in (Sestoft, 2002). In Section 5.8 we cherry-pick a few hybrids from the list to discuss the general applicability of our technique.

Contributions We show that refocusing and inlining-of-iterate-function steps become context dependent, the resulting abstract machine cannot be refunctionalised, and thus the junction with the functional correspondence is severed. However, the grammar of well-formed continuation stacks has a shape invariant that informs of which configuration of the abstract machine (hybrid or subsidiary) has to be invoked. This invariant can be used to recover shallow inspection. An expert inter-derivationist may observe the shape invariant in the code for one particular strategy, but producing the grammar of well-formed continuation stacks, and from there the invariant, is the systematic and formal step. Moreover, a non-deterministic finite automaton can be constructed that establishes the formal correspondence between well-formed continuation stacks and reduction contexts. Finally, thanks to the single-function nature of the semantics, the artefacts are implemented using single-layer continuation-passing style without control delimiters, as opposed to a two-layer CPS or a single-layer CPS with control delimiters (Biernacka *et al.*, 2005).

In this chapter we illustrate the problem and the solution by deriving substitution-based[4] semantic artefacts for normal order, the standard full-reducing and complete strategy of the pure lambda calculus. The code of the derivation is available on line.[5] We use the same programming language (Standard ML) and follow the same steps in the presentation of the derivation as (Danvy, 2005) and (Danvy *et al.*, 2011).

As further evidence of the applicability of our solution (which we discuss further in Section 5.8) we have also derived substitution-based semantic artefacts for hybrid applicative order (Sestoft, 2002), a hybrid strategy that is full-reducing but incomplete. The code of the derivation is also available on line.[6] The grammar of well-formed continuation stacks and the shape invariant are documented in the code.

In the normal order derivation we obtain a substitution-based, eval/apply, open-terms version of Pierre Crégut's full-reducing Krivine machine KN (Crégut, 2007). In (García-Pérez *et al.*, 2013) we have recently derived the original environment-based, push/enter, closed-terms, etc, version of the machine from a lambda calculus of closures with de Bruijn indices and levels. In that work we used the grammar of well-formed continuation stacks not only to recover shallow inspection but also to remove explicit control from an abstract machine. There are, therefore, other applications of well-formed continuation stacks that

form. Call-by-value in the pure lambda calculus realises the idea of passing parameters in weak normal form (Sestoft, 2002). Call-by-value in the applied lambda-value calculus of (Plotkin, 1975) and in the pure lambda-value calculus of (Ronchi Della Rocca & Paolini, 2004) realises the idea of passing parameters in 'non-application' form.

[4] A normaliser is substitution-based when it relies on an external capture-avoiding substitution function. In contrast, environment-based normalisers carry an environment parameter with the delayed substitutions (Biernacka & Danvy, 2007).

[5] `http://babel.ls.fi.upm.es/~agarcia/papers/SCICO-PEPM/normal-order.sml`

[6] `http://babel.ls.fi.upm.es/~agarcia/papers/SCICO-PEPM/hybrid-applicative-order.sml`

are worth exploring.

Structure of the chapter In Section 5.2 we show the structural, natural, and context-based operational semantics of normal order. In Section 5.3 we elaborate on the issue of the hybrid *nature* of a strategy. In Sections 5.4 to 5.7 we discuss the derivation of the semantic artefacts. We start from search functions and arrive at a reduction-free normaliser. The following sections are of particular interest: Section 5.4.5 where we define the grammar of continuation stacks from the type of defunctionalised continuations, Section 5.5.1 where we obtain the grammar of *well-formed* continuation stacks, prove the shape invariant, and construct the NFA, Section 5.6.2 where we illustrate why refocus becomes context-dependent, and Section 5.6.6 where we illustrate the loss of shallow inspection and recover it using the shape invariant. There, we arrive at a version of the full-reducing Krivine machine. In Section 5.8 we present other hybrid strategies and discuss the general applicability of our technique which applies not just to normal order but to any hybrid strategy. As usual, we close this chapter with related and future work, and conclusions.

5.2 Normal order, a hybrid strategy

Normal order is the standard full-reducing and complete strategy of the pure lambda calculus. It is defined by the slogan 'contract the leftmost redex first' understanding 'leftmost' as in (Curry & Feys, 1958) or 'leftmost-outermost' when referring to the redex's position in the abstract syntax tree of the term. Intuitively, given an abstraction $\lambda x.B$, normal order 'goes under lambda' and reduces B to nf. However, given an application $M\,N$, if M is β-reducible in an arbitrary number of steps to an arbitrary abstraction $\lambda x.B$, at that point the leftmost-outermost redex is $(\lambda x.B)N$ and normal order must reduce that redex and not reduce $\lambda x.B$. Since normal order reduces abstractions fully it cannot invoke itself recursively on M. It must rely on a less reducing strategy, one that does not reduce abstractions and does not reduce operands. In other words, it must rely on call-by-name.

The following sections present the structural, natural, and context-based operational semantics of normal order. The structural and natural versions are defined in the typical format of Hilbert-style logical theories where inference rules have zero or more premisses and at most one conclusion. The context-based or reduction semantics is the starting point of the syntactic correspondence.

5.2.1 Structural operational semantics

Figure 5.1 defines the structural operational semantics of small-step normal order $\rightarrow_{no}$ together with its subsidiary small-step call-by-name $\rightarrow_{bn}$. It also defines whnfs and nfs grammatically.

Call-by-name is a *uniform* strategy defined only in terms of itself. It is defined by two rules for applications. There are no rules for variables and abstractions because these are

$$\frac{}{(\lambda x.B)N \to_{bn} [N/x]B}\ (\textsc{Bn}\text{-}\beta) \qquad \frac{M \notin \mathsf{WHNF} \quad M \to_{bn} M'}{M\,N \to_{bn} M'\,N}\ (\textsc{Bn}\text{-}\mu)$$

$$\frac{}{(\lambda x.B)N \to_{no} [N/x]B}\ (\textsc{No}\text{-}\beta) \qquad \frac{M \notin \mathsf{WHNF} \quad M \to_{bn} M'}{M\,N \to_{no} M'\,N}\ (\textsc{No}\text{-}\mu 1)$$

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

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

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

Figure 5.1: Structural operational semantics of normal order. The dependency of normal order on call-by-name is highlighted. The greyed premisses are redundant but explanatory.

whnfs. Axiom Bn-β applies when the operator is an abstraction. This rule realises call-by-name β-contraction. Rule Bn-μ applies when the operator M is an application that is not a whnf. The greyed first premiss in Bn-μ is redundant, for if the second premiss holds then the greyed premiss holds. Nevertheless we include it for readability and to better explain the search functions of Section 5.4.2.

Normal order is a *hybrid* strategy that depends on subsidiary call-by-name. Normal order is defined by one rule for abstractions and four rules for applications. Axiom No-β realises normal order β-contraction, and rule No-ξ means 'go under lambda'. Rule No-μ1 applies when the operator is not in whnf. The greyed first premiss is also redundant. The second premiss holds when (and therefore prescribes that) call-by-name reduction is performed on the operator. This premiss is where the dependency on call-by-name occurs and is highlighted in the figure. Rule No-μ2 applies when the operator is in whnf but not an abstraction nor a nf. (If it were an abstraction then No-β would have been applicable, if it were a nf then Rule No-ν would be the candidate for applicability.) Rule No-ν applies when the operator is in nf but not an abstraction and the operand is not a nf. Although a term in nf is also in whnf, No-μ2 and No-ν are non-overlapping because the third premiss of No-μ2 is not the case when M is a nf. In both cases, the application MN is a *neutral term*, *i.e.*, an application that does not reduce to a redex and is an instance of the regular expression $x\,\Lambda\,\{\Lambda\}^*$. In No-$\nu$, M is either a variable or a neutral term in normal form $x\,\mathsf{NF}\,\{\mathsf{NF}\}^*$, or more succinctly, a term in NNF (Figure 5.1) which by abuse of language we shall call a neutral term in normal form because including variables will simplify later

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

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

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

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

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

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

Figure 5.2: Natural semantics of normal order. The dependency of normal order on call-by-name is highlighted.

definitions.

5.2.2 Natural semantics

Figure 5.2 defines the natural semantics of big-step normal order $\Downarrow_{no}$ together with subsidiary big-step call-by-name $\Downarrow_{bn}$. Call-by-name is a uniform strategy defined only in terms of itself. Rule Bn-Var says call-by-name is an identity on variables. Rule Bn-Abs says it does not go under lambda. Rule Bn-Con says it reduces redices without reducing the operands. Rule Bn-Neu says it does not reduce operands in neutral terms.[7]

Normal order is a hybrid strategy that depends on subsidiary call-by-name (the dependencies are highlighted in the figure). Rule No-Var says normal order is an identity on variables. Rule No-Abs says normal order goes under lambda. Rules No-Con and No-Neu say in their first premiss that normal order depends on call-by-name to reduce operators in applications MN which are potential redices. If the result is an abstraction (second premiss of rule No-Con) then normal order reduces the redex without reducing the operand. If the result is not an abstraction (second premiss of rule No-Neu) then the

[7] As explained in (Sestoft, 2002, p.421), this rule is what distinguishes call-by-name in the *pure* lambda calculus from its definition in the *applied* by-name calculus of (Plotkin, 1975) where there are no neutrals.

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

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

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

Figure 5.3: Canonical substitution-based reduction-free normaliser for normal order

original application MN is a neutral term. In that case normal order reduces the operator and the operand fully. It is immediate to prove that call-by-name is a right identity of normal order ($\Downarrow_{no} = \Downarrow_{no} \circ \Downarrow_{bn}$) and, moreover, that the order in which call-by-name reduces the redices of M corresponds to the order in which normal order is to reduce the redices of M.

Both $\Downarrow_{no}$ and $\Downarrow_{bn}$ are syntax-directed partial functions. The Boolean conditions in premisses are non-overlapping and so the rules can be applied deterministically and translate directly to a strict functional program in which a term matching the left-hand-side of the conclusion is recursively reduced according to the premisses from left to right, with conditions corresponding to case analysis. Such a program is the canonical substitution-based reduction-free normaliser shown in Figure 5.3, where the dependency on subsidiary `bn` is highlighted in the code of hybrid `no`. We have chosen a de-Bruijn-indices representation for lambda terms (Barendregt, 1984). Function `subst` implements the standard substitution operator for de Bruijn terms. We omit its definition here for brevity and refer the reader to the on-line code.

5.2.3 (Context-based) reduction semantics

A reduction semantics is the starting point of a syntactic correspondence (Danvy & Nielsen, 2004). The reduction semantics of call-by-name and normal order are shown in Figure 5.4. In Section 5.4 we derive the reduction-based normaliser that implements the reduction

$$
\begin{array}{lcl}
\mathbf{C}_{bn}[\,] & ::= & [\,] \mid \mathbf{C}_{bn}[\,]\,\Lambda \\
\mathbf{C}_{bn}[(\lambda x.B)N] \rightarrow_{bn} \mathbf{C}_{bn}[[N/x]B] & & \\
\mathbf{C}_{no}[\,] & ::= & [\,] \mid \mathbf{C}_{bn}[\,]\,\Lambda \mid \lambda x.\mathbf{C}_{no}[\,] \mid \mathbf{C}_{ne}[\,] \\
\mathbf{C}_{ne}[\,] & ::= & \mathsf{NNF}\,\mathbf{C}_{no}[\,] \mid \mathbf{C}_{ne}[\,]\,\Lambda \\
\mathbf{C}_{no}[(\lambda x.B)N] \rightarrow_{no} \mathbf{C}_{no}[[N/x]B] & &
\end{array}
$$

Figure 5.4: (Context-based) reduction semantics for normal order.

semantics of normal order. A reduction semantics consists of a grammar for reduction contexts and a contraction rule for redices within context holes. Reduction is defined as the iteration of (i) uniquely decomposing the term into a reduction context plus a redex within the hole, (ii) contracting the redex within the hole and, (iii) recomposing the resulting term. The iteration either terminates when the term is irreducible or otherwise diverges.

Call-by-name is a uniform strategy, its context grammar only involves call-by-name contexts $\mathbf{C}_{bn}[\,]$. Normal order is a hybrid strategy, its context grammar involves call-by-name contexts (highlighted). The greyed call-by-name contraction rule is not part of the normal order reduction which has its own unique contraction rule. Call-by-name contexts are defined by two productions, one for the empty context when a redex is found, and another for recursively going over operators in applications. Normal order contexts $\mathbf{C}_{no}[\,]$ are defined by four productions. The first production derives an empty context (the input term is a redex). The second derives an application with a call-by-name context in operator position. The third production derives an abstraction with the context in the body ('going under lambda'). The fourth derives a neutral context $\mathbf{C}_{ne}[\,]$ which is in turn defined by two productions, one for neutral terms in normal form to the left of a normal order context, and another for going recursively over operators of neutral terms.

Neutral contexts do not define a strategy because they do not derive an empty context. Observe that the last two productions for $\mathbf{C}_{no}[\,]$ precisely match the shape of a whnf and 'specify' the further reduction of the whnf obtained by call-by-name reduction.

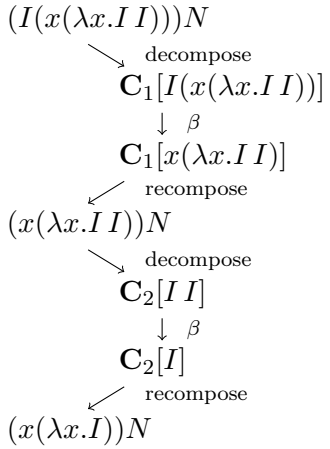
Figure 5.5 shows an example of a reduction sequence for a particular term. The reduction contexts obtained are shown on the right.

Theorem 5.2.1 (Unique decomposition)**.** *A term T is either a nf or there exists a unique context $\mathbf{C}_{no}[\,]$ and redex R such that $T \equiv \mathbf{C}_{no}[R]$.*

Proof. By structural induction on T.

Case $T \equiv x$**:** T is in nf.

Case $T \equiv \lambda x.B$**:** if B is in nf then T is in nf. Otherwise, by the ind. hyp. we have $B \equiv \mathbf{C}_{no}[R]$ and therefore $T \equiv \lambda x.\mathbf{C}_{no}[R]$. The unique context for T is $\lambda x.\mathbf{C}_{no}[\,]$



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

Figure 5.5: Example of a normal order reduction sequence in the context-based reduction semantics. The terms I and N stand respectively for the identity abstraction $(\lambda x.x)$ and for a term in normal form.

derivable from the axiom as follows: $\mathbf{C}_{no}[\,] \Rightarrow \lambda x.\mathbf{C}_{no}[\,]$.

Case $T \equiv M\,N$ with $M \equiv \lambda x.B$: whether B is in nf or not, T is a redex and the unique context for T is $[\,]$ derivable from the axiom as follows: $\mathbf{C}_{no}[\,] \Rightarrow [\,]$.

Case $T \equiv M\,N$ with $M \not\equiv \lambda x.B$: there are two sub-cases:

Case $M \in \mathsf{NNF}$: if N is in nf then T is in nf. Otherwise, by the ind. hyp. we have $N \equiv \mathbf{C}_{no}[R]$ and therefore $T \equiv M\,\mathbf{C}_{no}[R]$. The unique context for T is $M\,\mathbf{C}_{no}[\,]$ derivable from the axiom as follows: $\mathbf{C}_{no}[\,] \Rightarrow \mathbf{C}_{ne}[\,] \Rightarrow \mathsf{NNF}\,\mathbf{C}_{no}[\,] \Rightarrow M\,\mathbf{C}_{no}[\,]$.

Case $M \notin \mathsf{NNF}$: M is not in nf or otherwise it would be an abstraction and we are assuming $M \not\equiv \lambda x.B$. By the ind. hyp. we have $M \equiv \mathbf{C}_{no}[R]$ and therefore $T \equiv \mathbf{C}_{no}[R]\,N$. The only non-terminals leading to a redex in operator position are $\mathbf{C}_{bn}[\,]$ and $\mathbf{C}_{ne}[\,]$. These are disjoint cases: in $\mathbf{C}_{bn}[\,]$ the redex is located in the leftmost operator of a multiple application whereas in $\mathbf{C}_{ne}[\,]$ the redex is located at the right of a NNF in the leftmost neutral operator of a multiple application. In the first case the unique context for T is $\mathbf{C}_{bn}[\,]\,N$ derived from the axiom as follows: $\mathbf{C}_{no}[\,] \Rightarrow \mathbf{C}_{bn}[\,]\,\Lambda \Rightarrow \mathbf{C}_{bn}[\,]\,N$. In the second case the unique context for T is $\mathbf{C}_{ne}[\,]\,N$ derived as follows: $\mathbf{C}_{no}[\,] \Rightarrow \mathbf{C}_{ne}[\,] \Rightarrow \mathbf{C}_{ne}[\,]\,\Lambda \Rightarrow \mathbf{C}_{ne}[\,]\,N$. □

5.3 Hybrid style and hybrid nature

In Section 5.1 we defined uniform and hybrid *style*, and contrasted it to the notion of uniform and hybrid *nature*. We concentrate the discussion on the latter in this section,

where we define it formally and discuss its significance. Let $\mathcal{L}(\mathbf{C}_{st}[\ ])$ be the language of reduction contexts of strategy st.

Definition 5.3.1. *(i) A strategy u is uniform in nature iff given contexts $\mathbf{C}[\ ] \in \mathcal{L}(\mathbf{C}_u[\ ])$ and $\mathbf{C}'[\ ] \equiv \mathbf{C}[\mathbf{C}''[\ ]]$, then $\mathbf{C}'[\ ] \in \mathcal{L}(\mathbf{C}_u[\ ])$ iff $\mathbf{C}''[\ ] \in \mathcal{L}(\mathbf{C}_u[\ ])$.*

(ii) A strategy h is hybrid in nature iff h is not uniform in nature.

In words, given a reduction context $\mathbf{C}[\ ]$ of u, all the reduction contexts $\mathbf{C}'[\ ]$ of u that have $\mathbf{C}[\ ]$ as a prefix, *i.e.*, $\mathbf{C}'[\ ] \equiv \mathbf{C}[\mathbf{C}''[\ ]]$, are generated by inclusion of the language $\mathcal{L}(\mathbf{C}_u[\ ])$ in $\mathbf{C}[\ ]$. The latter implies that the language $\mathcal{L}(\mathbf{C}_u[\ ])$ is closed by inclusion. For example, given call-by-name contexts $[\ ]M$ and $[\ ]N$ the inclusions $([\ ]M)N$ and $([\ ]N)M$ are call-by-name contexts. This applies to every call-by-name context and thus the strategy is uniform in nature. In contrast, given normal order contexts $[\ ]M$ and $\lambda x.[\ ]$ the inclusion $(\lambda x.[\ ])M$ is not a normal order context. The strategy is hybrid in nature.

Definition 5.3.2. *A strategy h that is hybrid in nature depends on a (different) subsidiary strategy s iff for some context $\mathbf{C}[\ ] \in \mathcal{L}(\mathbf{C}_h[\ ])$ and for any context $\mathbf{C}'[\ ] \equiv \mathbf{C}[\mathbf{C}''[\ ]]$, then $\mathbf{C}'[\ ] \in \mathcal{L}(\mathbf{C}_h[\ ])$ iff $\mathbf{C}''[\ ] \in \mathcal{L}(\mathbf{C}_s[\ ])$.*

In words, for some reduction context $\mathbf{C}[\ ]$ of h, all the reduction contexts $\mathbf{C}'[\ ]$ of h that have $\mathbf{C}[\ ]$ as a prefix, *i.e.*, $\mathbf{C}'[\ ] \equiv \mathbf{C}[\mathbf{C}''[\ ]]$, are generated by inclusion of the subsidiary language $\mathcal{L}(\mathbf{C}_s[\ ])$ in $\mathbf{C}[\ ]$. The latter implies that the inclusion of any context of s inside $\mathbf{C}[\ ]$ is a context of h, and that there may exist a context $\mathbf{C}'[\ ]$ of h such that the inclusion of $\mathbf{C}'[\ ]$ inside $\mathbf{C}[\ ]$ is not a context of h. For example, given the normal order context $\mathbf{C}[\ ] \equiv [\ ]M$ the inclusion of call-by-name context $[\ ]N$ inside $\mathbf{C}[\ ]$ gives the normal order context $([\ ]N)M$. However, the inclusion of the normal order context $\lambda x.[\ ]$ inside $\mathbf{C}[\ ]$ gives $(\lambda x.[\ ])M$ which is not a normal order context. We are assuming that a strategy has the empty context $[\ ]$ in its language of contexts, a so-called strategy that does not contract the input redex term is a questionable device. For example, $\mathbf{C}_{ne}[\ ]$ in Section 5.2.3 is not a subsidiary strategy of normal order. Had $\mathbf{C}_{ne}[\ ]$ included the empty context (by adding the production $\mathbf{C}_{ne}[\ ] ::= [\ ]$) then it would be a subsidiary strategy of $\mathbf{C}_{no}[\ ]$, as well as a super-strategy of $\mathbf{C}_{bn}[\ ]$, and thus $\mathbf{C}_{no}[\ ]$ would depend on $\mathbf{C}_{bn}[\ ]$ through $\mathbf{C}_{ne}[\ ]$. Notice that in such a case, both *no* and *ne* would be hybrid strategies depending on each other. We assume that a hybrid strategy depends on at least one subsidiary strategy, which may in turn be uniform or hybrid.

The distinction between style (a property of a semantic definition) and nature (a property of a strategy) means that it is possible to define a hybrid in nature in uniform style by making the subsidiaries implicit. However, due to the hybrid nature, the subsidiaries can be 'unearthed' and made explicit in hybrid style across semantic definitions (structural, context-based, natural, etc). For example, the various uniform-style structural operational semantics of normal order discussed in (García-Pérez & Nogueira, 2013, Section 3) (Figure 1 and the instantiation of the principal reduction machine (Ronchi Della Rocca & Paolini, 2004)), and Solution 5.3.6 of (Pierce, 2002, p. 502), can be rewritten in hybrid

style (Section 5.2.1) for the reasons explained in that work. Here we discuss another illustrative example: the natural semantics of normal order in uniform style, in the fashion of the inner and ahead machines of (Paolini & Ronchi Della Rocca, 1999):

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

$$\frac{[N_0/x]B\, N_1 \dots N_n \Downarrow_{no} N'}{(\lambda x.B) N_0\, N_1 \dots N_n \Downarrow_{no} N'}\ (\textsc{Con})$$

$$\frac{N_i \Downarrow_{no} N_i' \qquad 1 \leq i \leq n}{x\, N_1 \dots N_n \Downarrow_{no} x\, N_1' \dots N_n'}\ (\textsc{Neu})$$

This definition uses a flattened representation of multiple applications and therefore the input term must be deeply inspected down to the leftmost operator. Such an inspection amounts to a search and test for whnf-ness (rules Con and Neu), in other words, a strategy that reduces non-strictly up to whnf (call-by-name). Rules Var and Con in isolation implement call-by-name. Rules Var, Abs, and Neu match the shape of a whnf.

A strategy can be defined in a natural semantics as the *composition* of two functions in so-called *eval-readback* style (Grégoire & Leroy, 2002). The 'eval' function delivers intermediate results and the 'readback' function distributes reduction over the subterms of the intermediate result. The eval-readback approach is a degenerate case of normalisation-by-evaluation (Aehlig & Joachimski, 2004) in which the value domain is the set of terms, and readback is 'reify' without the translation from domain values to terms. Normal order is defined in eval-readback as $\Downarrow_{no} \;=\; \Downarrow_{rn} \circ \Downarrow_{bn}$, where 'eval' is call-by-name (unsurprisingly, 'eval' is performed by the subsidiary of the hybrid-style definition) and readback $\Downarrow_{rn}$ distributes reduction over whnfs and is defined as follows:

$$\frac{}{x \Downarrow_{rn} x}\ (\textsc{Rn-Var}) \qquad \frac{B \Downarrow_{bn} B' \quad B' \Downarrow_{rn} B''}{\lambda x.B \Downarrow_{rn} \lambda x.B''}\ (\textsc{Rn-Abs})$$

$$\frac{M \Downarrow_{rn} M' \quad N \Downarrow_{bn} N' \quad N' \Downarrow_{rn} N''}{M\, N \Downarrow_{rn} M'\, N''}\ (\textsc{Rn-Neu})$$

Notice that $\Downarrow_{rn}$ has no Rn-Con rule because it operates on terms in whnf, and a term in whnf does not have an outermost redex. In Section 5.9 we discuss the advantages and disadvantages of deriving syntactic correspondences for strategies defined in eval-readback style. We do not use such style in this chapter.[8]

[8] An eval-readback strategy can be defined in hybrid style using the eval as subsidiary, but not every hybrid in nature can be defined in eval-readback style, only those hybrids for which the subsidiary is a sub-relation, *i.e.*, every subsidiary reduction context is in the hybrid's language of reduction contexts.

5.4 From search functions to reduction-based normaliser

Following the steps of (Danvy *et al.*, 2011), we derive the reduction-based normaliser for normal order from search functions, one for normal order and another for call-by-name, which mirror the compatibility rules of their respective structural operational semantics. In Section 5.5.1 we introduce the shape invariant of well-formed continuation stacks which will allow us to recover the shallow inspection property in Section 5.6.6.

5.4.1 One datatype for irreducible forms

The datatype `term` representing lambda calculus terms in a de-Bruijn-indices representation has been given in Figure 5.3 on page 94. As for nfs and whnfs, we could define one datatype for each but this would complicate defunctionalisation. Instead, since a term in nf is also in whnf (a nf is a fully-reduced whnf) we use a single datatype `whnf` for both, reuse most of the datatypes (irreducible forms, redices, defunctionalised continuations, etc.) and use single-layer CPS. The definition of `whnf` follows the 'function' and 'accumulator' representation of (Grégoire & Leroy, 2002) and conforms to the definition of whnf in Figure 5.1. In words, a `whnf` is an abstraction (value constructor `FUN`), an index (value constructor `ACC` applied to the index and an empty list), or a neutral term (value constructor `ACC` applied to an index and a non-empty list of term arguments). Function `embed` recovers a `term` from a `whnf`, and `apply_acc` appends a `term` operand to a `whnf`.

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

(*  embed : whnf -> term *)
fun embed (FUN b)       = LAM b
  | embed (ACC (n, ts)) = foldl (fn (s, t) => APP (s, t)) (IND 0) ts

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

5.4.2 Search functions

As suggested by the hybrid definition of the strategy, two search functions are required: a `search_nf` function for the hybrid that searches for a nf or the next redex to be contracted, and a `search_whnf` function for the subsidiary that searches for a whnf or the next redex in the call-by-name sub-reduction to be contracted. In the next sections the two search functions will be transformed into decomposition functions which, additionally to the next redex, deliver the context where the redex appears. The search functions mirror the compatibility rules of the structural operational semantics of Figure 5.1. For those input terms for which there is no compatibility rule (*e.g.*, variables and abstractions in call-by-name) the search functions return them as results in the form of a `whnf` (recall it embeds nfs).

Below, the datatype `redex` represents redices, and the datatype `found` consists of a `whnf` or a `redex`.

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

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

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

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

5.4.3 CPS-transformed search functions

The search functions are CPS-transformed by naming intermediate results of computation and by turning all the calls into tail calls. Our whnfs are evaluated at call time and there is no difference between denotable values and expressible values (Danvy, 2006b). Neither thunks (nor their CPS counterpart) are used to represent whnfs and hence we use the call-by-value CPS transformation.

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

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

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

5.4.4 Simplifying the CPS-transformed search functions

The CPS-transformed search functions are simplified by making them return a result only when a redex or a whnf is found. The simplification rests on an isomorphism between the type signatures of continuations (Ager *et al.*, 2005). Datatype `found` is the disjoint sum of `whnf` and `redex`, and the type `(whnf + redex -> 'a)` is isomorphic to the type `(whnf -> 'a) * (redex -> 'a)`. The `(redex -> 'a)` continuations are always the identity and can be optimised away. Since the result is always of type `found`, we can instantiate the type variable `'a` to `found`, leaving `term * (whnf -> found) -> found` as the type signature for the CPS-simplified search functions.

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

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

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

5.4.5 Defunctionalising continuations

Continuations are defunctionalised by enumerating the inhabitants of the function space, collected in datatype `continuation`, and by introducing the `apply_cont` function that dispatches on them.

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

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

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

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

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

The constructors of datatype `continuation` are numbered according to the chronological occurrence of the anonymous functions in Section 5.4.4 that they represent, except for `C0`, which represents the initial continuation. Value constructor `C1` represents the continuation in `search_whnf_sim`. Value constructors `C2`, `C3`, `C4` and `C5` represent the continuations in `search_nf_sim`. Following (Danvy *et al.*, 2011) we write `C5 of continuation * whnf` with the `continuation` argument first because in the code of Section 5.4.4 the continuation `k` appears before the free `nm` in the body of the function that `C5` stands for. A defunctionalised continuation is a stack of value constructors where additionally there live terms. The type `continuation` is isomorphic to continuation stacks K defined by the following grammar:

$$\mathsf{K} \;::=\; \texttt{C0} \;\mid\; \texttt{C1(term)} : \mathsf{K} \;\mid\; \texttt{C2} : \mathsf{K} \;\mid\; \texttt{C3(term)} : \mathsf{K} \;\mid\; \texttt{C4(term)} : \mathsf{K} \;\mid\; \texttt{C5(whnf)} : \mathsf{K}$$

The symbols `term` and `whnf` stand for ML expressions of the corresponding type. The datatype notation and the sugared stack notation of the grammar are equivalent. For example, `C3(n, C0)` corresponds to `C3(n) : C0`. We will heavily use the sugared stack notation in Section 5.5.

5.4.6 From search to decomposition

We turn the search functions into decomposition functions that deliver the input term back if in irreducible form (whnf or nf respectively) or the found redex together with the reduction context where it appears. Functions `apply_cont`, `search_whnf`, and `search_nf` in Section 5.4.5 correspond to `decompose_cont`, `decompose_whnf`, and `decompose_nf` respectively.

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

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

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

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

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

5.5 Continuation stacks

In this section we obtain the grammar of well-formed continuation stacks (*i.e.*, the continuations returned by function `decompose`) which are a subset of the K defined in Section 5.4.5. The well-formed continuation stacks enjoy a shape invariant which is used in Section 5.6.6 to recover the shallow inspection property of the abstract machine. We also show the correspondence of well-formed continuation stacks and the reduction contexts of normal order that were presented in Section 5.2.3.

5.5.1 Well-formed continuation stacks and their shape invariant

Well-formed continuation stacks are defined by the following EBNF grammar (Chapter 2 introduced the regular expression notation):

$$
\begin{array}{rcl}
\mathsf{W} & ::= & \mathsf{A}\,\texttt{C0} \\
\mathsf{A} & ::= & \varepsilon \mid \mathsf{B}\,\texttt{C3(term) :} \mid \mathsf{N} \mid \mathsf{A}\,\texttt{C2 :} \\
\mathsf{B} & ::= & \varepsilon \mid \mathsf{B}\,\texttt{C1(term) :} \\
\mathsf{N} & ::= & \mathsf{N}\,\texttt{C4(term) :} \mid \mathsf{A}\,\texttt{C5(whnf) :}
\end{array}
$$

In the rest of this section we prove that `decompose` delivers well-formed continuation stacks and that these have a shape invariant. First, we give a definition of the decomposition functions of Section 5.4.6 using the sugared stack notation (highlighted) which will ease the presentation of the proofs of lemmata and of the theorem about the shape invariant. The shape invariant will be used in Section 5.6.6 to recover the shallow-inspection property.

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

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

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

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

In the following lemmata we use `t` for the ML deep embedding of a term, and write `t` $\in$ WHNF to mean that the term is in whnf.

Lemma 5.5.1 (`decompose_whnf`). *Let* `t` *be any term and* `k0` *any continuation stack.*

(i) If `t` $\notin$ WHNF *then*

```
decompose_whnf (t, k0) = DEC (SUB (b, n), k : k0)
```

for some `b`, `n`, *and* `k`, *where* `k` $\in \{$ `C1(term) :` $\}^*$.

(ii) If `t` $\in$ WHNF *then* `decompose_whnf (t, k0) = decompose_cont (k0, w)` *for some* `w` *where* `embed w = t`.

Proof. Both *(i)* and *(ii)* are true by direct observation of the `decompose_whnf` function. If `t` is an index or a lambda abstraction, then `t` $\in$ WHNF and *(ii)* holds by the first or the second clause of `decompose_whnf` respectively. If `t` is an application, then `decompose_whnf` will traverse it recursively to the left, pushing new `C1` constructors on the stack. If `t` $\notin$ WHNF then the leftmost operator of `t` is a lambda abstraction `LAM b` such that the second clause of `decompose_cont` will eventually return `DEC (SUB (b, n), k : k0)` where `k` $\in \{$`C1(term) :`$\}^*$, and *(i)* holds. If `t` $\in$ WHNF then the leftmost operator of `t` is an index `IND i` such that the first clause of `decompose_cont` (case `wm` $\neq$ `FUN b`) will successively recompose a `whnf` reflecting the input term `t` until throwing it into continuation `k0`, *i.e.*, `decompose_cont (k0, w)` with `w` = `ACC (i, [...])` and `embed w = t`. Then *(ii)* holds. □

Lemma 5.5.2 (`decompose` delivers well-formed continuation stacks W). *Let* `t` *be any term not in nf. Then* `decompose(t) = DEC (SUB (b, n), k)` *for some* `b`, `n`, *and* `k`, *where* `k` $\in$ W.

Proof. In order to deal with an input continuation `k0`, we generalise the lemma as follows. Let `t` be any term not in normal form, and `k0` $\in$ W whose top constructor is neither `C1` nor `C3`. Then:

```
decompose_nf (t, k0) = DEC (SUB (b, n), k1 : k0)
```

for some `b`, `n`, and `k1` where `k1 : k0` $\in$ W. (The `k1 :` could be the empty symbol ε.)

The proof is by structural induction on `t`:

Case `t` $\equiv$ `APP (LAM b, n) :` the term `t` is a redex and then by code expansion:

```
   decompose_nf (APP (LAM b, n), k0)
=  decompose_whnf (LAM b, C3(n) : k0)
=  decompose_cont(C3(n) : k0, FUN b)
=  DEC (SUB (b, n), k0)
```

where `k0` $\in$ W by assumption and `k1 :` is equal to ε. This corresponds to production A ::= ε in the grammar of well-formed stacks.

Case `t` ≡ `APP (IND i, m2)` **:** the `IND i` is in nf, so decomposition will eventually take the `m2` branch. Then

```
   decompose_nf (APP (IND i, m2), k0)
=  decompose_whnf (IND i, C3(m2) : k0)
=  decompose_cont (C3(m2) : k0, ACC (i, []))
=  decompose_nf (IND i, C4(m2) : k0)
=  decompose_cont (C4(m2) : k0, ACC (i, []))
=  decompose_nf (m2, C5(ACC (i, []) : k0)
```

which returns `DEC (SUB (b, n), k1 : C5(ACC (i, [])) : k0)` where the continuation `k1 : C5(ACC (i, [])) : k0` ∈ W by the induction hypothesis. This corresponds to productions A ::= N and N ::= A `C5(whnf)` : in the grammar of well-formed stacks.

Case `t` ≡ `APP (m1, m2)` **where** `m1` ≢ `LAM t'`**:** there are two sub-cases:

Case `m1` **is in whnf :** let `w1` be a `whnf` such that `w1` ≢ `FUN t'` and `embed w1 = m1`. By code expansion:

```
   decompose_nf (APP (m1, m2), k0)
=  decompose_whnf (m1, C3(m2) : k0)
```

which by Lemma 5.5.1 delivers `decompose_cont (C3(m2) : k0, w1)` since the `m1` is in whnf. By code expansion we get

```
decompose_nf (m1, C4(m2) : k0)
```

which, in turn, delivers `DEC (SUB (b, n), k1 : C4(m2) : k0)` where `k1 : C4(m2) : k0` ∈ W by the induction hypothesis.

Moreover, `m1` is a neutral term. Thus, the last step where the induction hypothesis is used may only be the case `m1` ≡ `APP (IND i', m2')` (the previous case) or the case `m1` ≡ `APP (m1', m2')` with `m1'` ≢ `LAM t''` (this very same case). Hence, decomposition will push an arbitrary number of `C4` constructors until pushing a `C5` constructor and then resuming over the operand of the leftmost index. This corresponds to productions A ::= N and N ::= N `C4(term) :` | A `C5(whnf)` : in the grammar of well-formed stacks.

Case `m1` **is not in whnf :** then

```
   decompose_nf (APP (m1, m2), k0)
=  decompose_whnf (m1, C3(m2) : k0)
```

which returns `DEC (SUB (b, n), k1 : C3(m2) : k0)` where `k1` ∈ { `C1(term) :` }* by Lemma 5.5.1. This corresponds to productions A ::= B `C3(term)` : and B ::= ε | B `C1(term)` : in the grammar of well-formed stacks.

Case `t` ≡ `LAM t'`**:** the `t'` is not in nf and then

```
decompose_nf (LAM t', k0) = decompose_nf (t', C2 : k0)
```

which returns `DEC (SUB (b, n), k1 : C2 : k0)` where `k1 : C2 : k0` ∈ W by the induction hypothesis. This corresponds to production A ::= A `C2` : in the grammar of well-formed stacks.

Observe that the use of the induction hypothesis will produce a prefix of a well-formed stack which omits the `C0` constructor. This explains the role of the axiom W and of the auxiliary non-terminal A in the grammar of well-formed stacks. The induction hypothesis can only be applied over stacks whose top constructor is different from `C1` or `C3`, (*i.e.*, different from the fragments generated by non-terminal B, which corresponds to *bn*) and the case for `C4` has been already covered by the cases where `t = APP (m1, m2)`. This explains the occurrences of A on top of constructors `C0`, `C2`, and `C5` in the grammar of well-formed stacks. □

Theorem 5.5.3 (Shape invariant)**.** *Let* `k` ∈ W*.*

(i) In `k`*, a* `C1` *constructor always occurs on top of a* `C1` *or a* `C3` *constructor.*

(ii) In `k`*, a* `C3` *constructor never occurs on top of a* `C1` *or a* `C3` *constructor.*

Proof. Both *(i)* and *(ii)* are immediate by the definition of well-formed stacks W. □

The shape invariant reflects the relative order in which the constructors of defunctionalised continuations appear in the stack, in particular those delimiting the hybrid from the subsidiary stage, which are the ones needed to recover the shallow inspection property (Section 5.6.6).

A hybrid strategy will have its grammar of well-formed continuation stacks. The hybrid and the subsidiary stages alternatively take control in the decomposition function in a deterministic way, and thus we state that the shape invariant is always observable. This is supported by the hybrid strategies that we analyse in Section 5.8.

5.5.2 Correspondence between well-formed continuation stacks and reduction contexts

Figure 5.6 shows the grammar of well-formed continuation stacks on the left and the grammar of reduction contexts on the right. We introduce a nameful representation for well-formed continuation stacks where `term` and `whnf` are replaced by Λ and NNF respectively (the only whnfs appearing in N are neutrals in nf) and where constructor $\mathbf{C}_2(x)$ (sans-serif fonts are used for the nameful representation) carries the variable symbol x which stands for the formal parameter in the abstraction body that is traversed after pushing `C2` on the stack. Symbols Λ and NNF in the grammar are to be considered as terminals.

$$
\begin{array}{rcl}
\mathsf{W} & ::= & \mathsf{A}\,\mathbf{C}_0 \\
\mathsf{A} & ::= & \varepsilon \mid \mathsf{B}\,\mathbf{C}_3(\Lambda) : \mid \mathsf{N} \\
 & \mid & \mathsf{A}\,\mathbf{C}_2(x) : \\
\mathsf{B} & ::= & \varepsilon \mid \mathsf{B}\,\mathbf{C}_1(\Lambda) : \\
\mathsf{N} & ::= & \mathsf{N}\,\mathbf{C}_4(\Lambda) : \mid \mathsf{A}\,\mathbf{C}_5(\mathsf{NNF}) :
\end{array}
\qquad
\begin{array}{rcl}
\mathbf{C}_{no}[\,] & ::= & \mathbf{C}_{au}[\,]\,\varepsilon \\
\mathbf{C}_{au}[\,] & ::= & [\,] \mid \mathbf{C}_{bn}[\,]\,\Lambda \mid \mathbf{C}_{ne}[\,] \\
 & \mid & \lambda x.\mathbf{C}_{au}[\,] \\
\mathbf{C}_{bn}[\,] & ::= & [\,] \mid \mathbf{C}_{bn}[\,]\,\Lambda \\
\mathbf{C}_{ne}[\,] & ::= & \mathbf{C}_{ne}[\,]\,\Lambda \mid \mathsf{NNF}\,\mathbf{C}_{au}[\,]
\end{array}
$$

Figure 5.6: Grammar of well-formed continuation stacks (left) and grammar of reduction contexts (right).

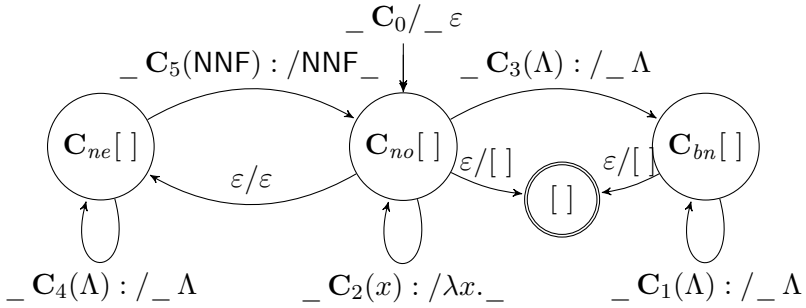


Figure 5.7: NFA accepting well-formed continuation stacks and reduction contexts

The two grammars are proven equivalent by respectively replacing non-terminals W, A, B, and N on the left by non-terminals $\mathbf{C}_{no}[\,]$, $\mathbf{C}_{au}[\,]$, $\mathbf{C}_{bn}[\,]$, and $\mathbf{C}_{ne}[\,]$ on the right, respectively. The terminal symbols are translated as follows. The initial continuation $\mathbf{C}_0$ on the left maps to ε on the right (the extent of the context implicitly represents the bottom of the stack). The ε on the left maps to $[\,]$ on the right (the top of the stack implicitly signals the occurrence of the hole). The other constructors of defunctionalised continuations inform of the relative position of their arguments with respect to the remaining stack symbols, which map to the remaining context symbols. Constructors $\mathbf{C}_1(\Lambda)$, $\mathbf{C}_3(\Lambda)$, and $\mathbf{C}_4[\,](\Lambda)$ indicate that the lambda term occurs at the right of the rest of the context, *i.e.*, $\mathbf{C}_{bn}[\,]\,\Lambda$, $\mathbf{C}_{bn}[\,]\,\Lambda$, and $\mathbf{C}_{ne}[\,]\,\Lambda$ respectively. Constructor $\mathbf{C}_5(\mathsf{NNF})$ indicates that the neutral in normal form occurs at the left of the rest of the context, *i.e.*, $\mathsf{NNF}\,\mathbf{C}_{au}[\,]$. Constructor $\mathbf{C}_2(x)$ indicates that the rest of the context lays under a lambda abstraction with formal parameter x, *i.e.*, $\lambda x.\mathbf{C}_{au}[\,]$ Thus, the constructors of defunctionalised continuations carry the essential information to recover the mixfix notation of reduction contexts from the prefix notation of continuation stacks.

The grammar of reduction contexts in Figure 5.6 defines the same language as the grammar of reduction contexts in Section 5.2.3. This is proven easily by removing the symbol ε and by inlining recursive non-terminal $\mathbf{C}_{au}[\,]$ into $\mathbf{C}_{no}[\,]$, turning the latter into a recursive non-terminal symbol.

The grammar of well-formed continuation stacks and the equivalent grammar of re-

duction contexts characterise the decomposition functions. As further evidence of this, we define the non-deterministic finite automaton (NFA) of Figure 5.7. The NFA accepts the stack and context languages (each transition is labelled with stack and context symbols, separated by a slash). Or alternatively, the NFA is a translating device where the symbols on the left of the slash are consumed and the symbols on the right are produced.

The NFA is obtained by taking the non-terminals of the grammar of reduction contexts in Figure 5.6 as states, and considering the possible productions of the grammar as transitions. Notice that the recursive $\mathbf{C}_{au}[\ ]$ is obviated and production $\mathbf{C}_{no}[\ ] ::= \mathbf{C}_{au}[\ ]$ is taken as the init transition. The underscore symbol stands for the relative position of the remaining stack (respectively, context) that the NFA is to accept. In the prefix notation of continuation stacks the underscore symbol always appears to the left (on top of the stack). In the mixfix notation of contexts the underscore's position varies accordingly. The ε symbol denotes that no symbols are consumed or produced. The position of the underscore is immaterial for the ε symbol, but the right of the init transition is labelled with $\_\ \varepsilon$ in accordance with the grammar of contexts in Figure 5.6.

Although sometimes related to each other in the literature (Danvy & Millikin, 2008), the initial continuation $\mathbf{C}_0$ and the hole $[\ ]$ are different objects. The initial continuation represents the bottom of the stack. The bottom of the stack alone corresponds to the empty context, but it does not correspond to a hole that is a sub-context of another context, *e.g.*, the context $\lambda x.[\ ]$ maps to the stack $\varepsilon\ \mathbf{C}_2(x) : \mathbf{C}_0$ where the bottom occurs at the right side of the stack and the hole is implicitly signaled by the top of the stack in the opposite side (*i.e.*, the greyed ε). Dually, the extent of the context implicitly stands for the bottom of the stack. We should morally write $(\lambda x.[\ ])\varepsilon$ where the greyed ε (we have also added greyed parentheses to indicate precedence) correspond to the bottom of the stack.

5.6 From reduction semantics to abstract machine

To obtain a reduction semantics we need a recomposition function that reconstructs a reduct by plugging a contractum back into the context corresponding to a defunctionalised continuation. This is implemented by a left fold over the continuation stack (alias, reduction context). We show the sugared and desugared versions of recomposition:

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

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

The reduction semantics iterates decomposition, contraction and recomposition until a nf is found. In the rest of this section we carry out the syntactic correspondence from the implementation of the reduction semantics and obtain an abstract machine.

5.6.1 Trampolined-style normaliser

The starting point is the trampolined-style normaliser implementing the reduction semantics.

```
type contractum = term
type result = whnf

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

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


(*  iterate : whnf_or_decomposition -> result *)
fun iterate (WHNF w)       = w
  | iterate (DEC (red, k)) = iterate (refocus (k, contract red))

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

The extensional refocus function is the composition of functions `decompose` and `recompose`. The function `contract` does not have to consider execution errors. A result (if any) can only be a nf (embedded in the `whnf`), otherwise the iteration diverges.

The normaliser implements a small-step state transition machine in *trampolined style* (Ganz *et al.*, 1999), where configurations (states) coincide with decompositions (*i.e.*, datatype `whnf_or_decomposition`). Discrete transitions steps are implemented by the composition of `contract`, `recompose` and `decompose`. The last two constitute extensional refocusing (Danvy & Nielsen, 2004; Danvy *et al.*, 2011) (function `refocus`). The `iterate` function is the trampoline, taking a decomposition, contracting the redex and then recursively invoking itself over the refocused contractum until the decomposition consists of a `whnf` (again, recall it embeds nfs).

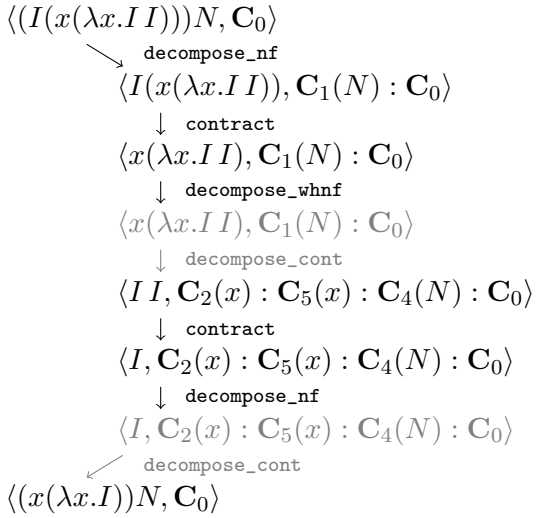


Figure 5.8: Context-dependent intensional refocus

5.6.2 Refocusing intensionally

We deforest recomposition and decomposition into an intensional refocus function (Danvy, 2005). The reduction-free iterate-and-refocus normaliser does away with intermediate reducts:

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

(*  iterate1 : whnf_or_decomposition -> result *)
fun iterate1 (WHNF w)       = w
  | iterate1 (DEC (red, k)) = iterate1 (refocus1 (k, contract red))

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

At this point, if we continue to apply the standard program-transformation steps to derive an artefact that realises the semantics in Section 5.6.1 then the intensional `refocus1` *will inspect the continuation* `k` *that it receives in order to determine which decomposition function has to continue.* This makes `refocus1` a context-dependent function. Function `decompose_whnf` in Section 5.4.6 is only invoked on `C1` and `C3`. Function `decompose_nf` is not invoked on `C1` and `C3`. The case expression in `refocus1` takes care of that.

For illustration, Figure 5.8 adapts the reduction example in Figure 5.5 using context-dependent refocus and where the composition of recompose and decompose is deforested.

Instead of a term within a context, we have a pair ⟨term, continuation stack⟩. Instead of recomposition followed by decomposition we have decomposition with the current continuation `k` (*i.e.*, `decompose_whnf` or `decompose_nf` depending on the outermost constructor in `k`). When an intermediate result reaches a whnf or a nf after invoking `decompose_whnf` or `decompose_nf` respectively, the intermediate result and the invocation on it of the dispatcher `decompose_cont` is shown greyed in the diagram.

In any case, we proceed with the standard derivation which unfortunately will take us to a machine without the shallow inspection property (Section 5.6.5).

5.6.3 Pre-abstract machine

We inline the contraction function `contract` in `iterate1` and derive a *pre-abstract machine* (Danvy & Nielsen, 2004).

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

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

5.6.4 Lightweight fusion by fixed-point promotion

We lightweight-fuse functions `iterate2` and `refocus1` (Danvy & Millikin, 2008). The result is an optimised normaliser where adjacent `iterate3` and `refocus1` have been fused.

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

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

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

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

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

5.6.5 Corridor transitions and inlining-of-iterate-function

There is a configuration (state) with only one possible transition.

```
   normalise3_cont (C0, w)
 =
   iterate3 (WHNF w)
 =
   w
```

We contract this corridor transition by inlining. After that, we perform the inlining-of-iterate-function step by inlining `iterate3` inside `normalise3_cont`. The following code shows the result with indices renamed from 3 to 4.

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

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

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

(*  normalise4 : term -> result *)
fun normalise4 t = normalise4_nf (t, C0)
```

Now `iterate3` is not used. Observe that the case expression introduced in the refocusing step (Section 5.6.2) is inlined twice in `normalise4_cont` (highlighted code). Consequently, *the artefact obtained does not have the shallow inspection property* because `normalise4_cont` pattern-matches on `k` at the inlined points.

5.6.6 Recovering the shallow inspection property

Function `normalise4_cont` dispatches on the continuation on top of the stack. In particular, `normalise4_cont` dispatches on `C1 (n, k)` and `C3 (n, k)` in its second and fourth clauses respectively, which are the clauses where the highlighted case expressions occur. *We use Theorem 5.5.3 to replace the case expressions by the appropriate normalising functions, thus recovering the shallow inspection property.* The corresponding lines are highlighted.

```
(*  normalise5_cont : continuation * whnf -> result *)
fun normalise5_cont (C0, w)          = w
  | normalise5_cont (C1 (n, k), wm)  =
      (case wm
        of (FUN b) => normalise5_whnf (subst (b, n, 0), k)
         | _       => normalise5_cont (k, apply_acc (wm, n)))
  | normalise5_cont (C2 k, wb)       =
      normalise5_cont (k, FUN (embed wb))
  | normalise5_cont (C3 (n, k), wm)  =
      (case wm
        of (FUN b) => normalise5_nf (subst (b, n, 0), k)
         | _       => normalise5_nf (embed wm, C4 (n, k)))
  | normalise5_cont (C4 (n, k), nm)  = normalise5_nf (n, C5 (k, nm))
  | normalise5_cont (C5 (k, nm), nn) =
      normalise5_cont (k, apply_acc (nm, embed nn))

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

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

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

The resulting normaliser is a big-step tail-recursive implementation of the abstract machine in Figure 5.9. The machine uses a nameful representation of terms and type annotations n, w and c to indicate whether the configuration is the hybrid, the subsidiary, or the dispatcher respectively. The dependency of the n configuration on the w configuration is highlighted in the figure. The machine is a substitution-based, eval/apply, open-terms version of the full-reducing Krivine machine KN (Crégut, 2007; García-Pérez *et al.*, 2013).

T	$\rightarrow$	$(T, \mathbf{C}_0)_n$
$(x, S)_n$	$\rightarrow$	$(x, S)_c$
$(\lambda x.B, S)_n$	$\rightarrow$	$(B, \mathbf{C}_2(x) : S)_n$
$(M\,N, S)_n$	$\rightarrow$	$(M, \mathbf{C}_3(N) : S)_w$
$(x, S)_w$	$\rightarrow$	$(x, S)_c$
$(\lambda x.B, S)_w$	$\rightarrow$	$(\lambda x.B, S)_c$
$(M\,N, S)_w$	$\rightarrow$	$(M, \mathbf{C}_1(N) : S)_w$
$(\lambda x.B, \mathbf{C}_1(N) : S)_c$	$\rightarrow$	$([N/x]B, S)_w$
$(M, \mathbf{C}_1(N) : S)_c$	$\rightarrow$	$(M\,N, S)_c$
$(B, \mathbf{C}_2(x) : S)_c$	$\rightarrow$	$(\lambda x.B, S)_c$
$(\lambda x.B, \mathbf{C}_3(N) : S)_c$	$\rightarrow$	$([N/x]B, S)_n$
$(M, \mathbf{C}_3(N) : S)_c$	$\rightarrow$	$(M, \mathbf{C}_4(N) : S)_n$
$(M, \mathbf{C}_4(N) : S)_c$	$\rightarrow$	$(N, \mathbf{C}_5(M) : S)_n$
$(N, \mathbf{C}_5(M) : S)_c$	$\rightarrow$	$(M\,N, S)_c$
$(T, \mathbf{C}_0)_c$	$\rightarrow$	T

Figure 5.9: Normal order abstract machine

5.7 From abstract machine to reduction-free normaliser

From an abstract machine with the shallow inspection property we can now obtain the reduction-free normaliser by applying standard derivation steps.

5.7.1 Refunctionalisation

The abstract machine in Section 5.6.5 is an instance of a defunctionalised CPS program, with a configuration for dispatching on the continuations (`normalise4_cont`) and two configurations for the hybrid and the subsidiary strategies (`normalise4_nf` and `normalise4_whnf`). By refunctionalising the machine we obtain a reduction-free normaliser in CPS.

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

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

(*  normalise6 : term -> result *)
fun normalise6 t = normalise6_nf (t, fn w => w)
```

5.7.2 Back to direct style by inverse CPS transformation

We introduce let expressions for the intermediate results and remove the continuations, obtaining a direct-style reduction-free normaliser.

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

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

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

Save for the ancillary `whnf` datatype and the init function `normalise7`, this is the canonical reduction-free normaliser in Figure 5.3, with `normalise7_whnf` corresponding to `bn` and `normalise7_nf` corresponding to `no`. We arrive at the end of the derivation.

5.8 Applicability

As illustrated in Section 5.4.5, a hybrid strategy has a grammar of well-formed continuation stacks whose shape invariant informs of the relative order in which the constructors of defunctionalised continuations appear on the stack, in particular those delimiting the hybrid from the subsidiary stages, which are the ones needed to recover the shallow inspection property. A hybrid may depend on several subsidiaries, some of which may in turn be hybrid and depend on other subsidiaries. But the separation between the different stages is clear and unambiguous, and the appropriate shape invariant can be obtained by examining the grammar of well-formed continuation stacks.

As discussed in the introduction, many interesting strategies in the literature are hybrid. In this section we show a few examples for which we have derived the grammar of well-formed continuation stacks and determined the shape invariants that enable the recovery of shallow inspection. The derivation of artefacts proceeds in similar fashion as we have done for normal order. For evidence, the complete derivation of a couple of the hybrids discussed below (preponed semantics of normal order and hybrid applicative order) is available on line.[9] The grammars of well-formed continuation stacks and the shape invariants are respectively documented in the code.

Preponed semantics of normal order. Normal order can be presented as the strategy resulting from adding production $\mathbf{C}_{ne}[\,] ::= [\,]$ to the non-terminal $\mathbf{C}_{ne}[\,]$ in the reduction semantics of Section 5.2.3. Here $\mathbf{C}_{no}[\,]$ and $\mathbf{C}_{ne'}$ are hybrids that depend on each other. The reduction semantics is pictured below:

$$\begin{array}{rcl}
\mathbf{C}_{ne'}[\,] & ::= & [\,] \mid \mathbf{C}_{ne'}[\,]\,\Lambda \mid \mathsf{NNF}\,\mathbf{C}_{no}[\,] \\
\mathbf{C}_{no}[\,] & ::= & [\,] \mid \mathbf{C}_{ne'}[\,]\,\Lambda \mid \mathsf{NNF}\,\mathbf{C}_{no}[\,] \mid \lambda x.\mathbf{C}_{no}[\,] \\
\multicolumn{3}{l}{\mathbf{C}_{no}[(\lambda x.B)N] \rightarrow_{no} \mathbf{C}_{no}[[N/x]B]}
\end{array}$$

In the grammar, the mutual dependencies are highlighted. We call this semantics *preponed* because the contraction of operands in neutral terms is preponed to the stage finding the intermediate irreducible form, which in this case is the subset of weak normal forms defined by the EBNF-grammar $\mathsf{WNF}' ::= \lambda x.\Lambda \mid \mathsf{NNF}$. We discuss this semantics further in Section 5.9.

[9] `http://babel.ls.fi.upm.es/~agarcia/papers/SCICO-PEPM/normal-order-preponed.sml`
`http://babel.ls.fi.upm.es/~agarcia/papers/SCICO-PEPM/hybrid-applicative-order.sml`

Head reduction This strategy performs outermost reduction up to head normal form. It is defined in (Barendregt, 1984) using a flat representation for multiple applications. The strategy goes under lambda but the redex $(\lambda x.M_0)M_1$ is contracted before contracting any of the redices in M_0.

$$\frac{n \geq 0 \quad m > 0}{\lambda x_1 \ldots x_n.(\lambda x.M_0)M_1M_2 \ldots M_m \to_h \lambda x_1 \ldots x_n.([M_1/x]M_0)M_2 \ldots M_m}$$

Head reduction is hybrid with implicit subsidiary call-by-name. The hybrid nature of the strategy is explicit in the following structural operational semantics. The dependency on the subsidiary occurs on the highlighted premiss of rule Hr-μ1:

$$\frac{}{(\lambda x.B)N \to_{bn} [N/x]B}\ (\text{Bn-}\beta) \qquad \frac{M \to_{bn} M'}{M\,N \to_{bn} M'\,N}\ (\text{Bn-}\mu)$$

$$\frac{}{(\lambda x.B)N \to_{h} [N/x]B}\ (\text{Hr-}\beta) \qquad \frac{M \to_{bn} M'}{M\,N \to_{h} M'\,N}\ (\text{Hr-}\mu1)$$

$$\frac{B \to_h B'}{\lambda x.B \to_h \lambda x.B'}\ (\text{Hr-}\xi)$$

$$\frac{M \in \mathsf{WHNF} \quad M \not\equiv \lambda x.B \quad M \to_h M'}{M\,N \to_h M'\,N}\ (\text{Hr-}\mu2)$$

Strong reduction and `byValue` The former defined in (Grégoire & Leroy, 2002) as $N(T)$, the latter defined in (Paulson, 1996, p.390) in code. We show their definitions below with a slight change of notation to avoid clashes with the notation we have used throughout this chapter. (In particular, we write $\mathcal{N}(T)$, $\mathcal{V}(T)$, $\mathcal{R}(\mathsf{Val})$, $\tilde{\Lambda}$, Val and $\mathbf{C}[\ ]$ respectively for $N(b)$, $V(b)$, $R(v)$, b, v and Γ_v in (Grégoire & Leroy, 2002).) Our definition of `byValue` uses our `term` datatype (Figure 5.3).

$$\begin{array}{rcl} \tilde{\Lambda} & ::= & x \mid \lambda x.\tilde{\Lambda} \mid \tilde{\Lambda}\,\tilde{\Lambda} \mid \langle \tilde{x}\,\{\mathsf{Val}\}^* \rangle \\ \mathsf{Val} & ::= & \lambda x.\tilde{\Lambda} \mid \langle \tilde{x}\,\{\mathsf{Val}\}^* \rangle \end{array}$$

$$\begin{array}{l} \mathbf{C}[\ ] \quad ::= \quad [\ ]\,\mathsf{Val} \mid \tilde{\Lambda}\,[\ ] \\ \mathbf{C}[(\lambda x.B)\mathsf{Val}] \to \mathbf{C}[[\mathsf{Val}/x]B] \\ \mathbf{C}[\,\langle \tilde{x}\ \mathsf{Val}_1 \ldots \mathsf{Val}_n \rangle\,\mathsf{Val}] \to \mathbf{C}[\,\langle \tilde{x}\ \mathsf{Val}_1 \ldots \mathsf{Val}_n\,\mathsf{Val} \rangle\,] \\ \mathcal{V}(T) \quad = \quad T' \quad (T \to^* T', T' \in \mathsf{Val}) \end{array}$$

$$\begin{array}{rcl} \mathcal{R}(\lambda x.B) & = & \lambda y.\mathcal{N}((\lambda x.B)\,\langle \tilde{y} \rangle) \quad (y \text{ fresh}) \\ \mathcal{R}(\,\langle \tilde{x}\ \mathsf{Val}_1 \ldots \mathsf{Val}_n \rangle\,) & = & x\,\mathcal{R}(\mathsf{Val}_1) \ldots \mathcal{R}(\mathsf{Val}_n) \\ \mathcal{N}(T) & = & \mathcal{R}(\mathcal{V}(T)) \end{array}$$

```
(*  eval : term -> term *)
fun eval (APP (m, n)) =
      (case eval m of LAM b => eval (subst (b, eval n, 0))
                    | m'    => APP (m', eval n))
  | eval t = t

(*  bodies : term -> term *)
fun bodies (LAM b)      = LAM (byValue b)
  | bodies (APP (m, n)) = APP (bodies m, bodies n)
  | bodies t            = t

(*  byValue : term -> term *)
and byValue t = bodies (eval t)
```

Both strategies are defined in eval-readback style (Section 5.3) where `eval` and $\mathcal{V}(T)$ are the eval functions, and `bodies` and $\mathcal{R}(\mathsf{Val})$ are the readback functions. The strategies are equivalent, save for the sequencing order of application reduction in eval functions (left-to-right in `eval` and right-to-left in $\mathcal{V}(T)$). The strategies are full-reducing (deliver nfs) and realise the idea of passing parameters in weak normal form. In (Grégoire & Leroy, 2002) they introduce extended lambda terms ($\tilde{\Lambda}$ in our notation) and η-expand abstractions with fresh formal parameters before normalising their bodies, in conformance with normalisation-by-evaluation, so that the eval stage $\mathcal{V}(T)$ can be replaced by the compiled Zinc Abstract Machine (Leroy, 1991).

Both strategies can be defined as a single hybrid function whose subsidiary is the strategy realised by the eval function which is, in both cases, call-by-value of the pure lambda calculus, $\Downarrow_{bv}$, a strategy that despite its name passes parameters by weak normal form. We show the natural semantics of `byValue` where the dependency is shown by the highlighted premisses:

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

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

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

$$\frac{}{x \Downarrow_{\texttt{byValue}} x}\ (\texttt{byValue}\text{-}\textsc{Var}) \qquad \frac{B \Downarrow_{\texttt{byValue}} B'}{\lambda x.B \Downarrow_{\texttt{byValue}} \lambda x.B'}\ (\texttt{byValue}\text{-}\textsc{Abs})$$

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

$$\frac{M \Downarrow_{bv} M' \quad M' \not\equiv \lambda x.B \quad M' \Downarrow_{\texttt{byValue}} M'' \quad N \Downarrow_{\texttt{byValue}} N'}{M\,N \Downarrow_{\texttt{byValue}} M''\,N'}\ (\texttt{byValue}\text{-}\textsc{Neu})$$

Hybrid applicative order Its natural semantics is defined in (Sestoft, 2002). Hybrid applicative order is full-reducing but incomplete. It uses subsidiary call-by-value like strong reduction and `byValue`. The dependency on call-by-value occurs in the highlighted premisses.

$$\frac{}{x \Downarrow_{ha} x}\ (\textsc{Ha-Var}) \qquad \frac{B \Downarrow_{ha} B'}{\lambda x.B \Downarrow_{ha} \lambda x.B'}\ (\textsc{Ha-Abs})$$

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

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

Hybrid applicative order does not resemble `byValue` as claimed by (Sestoft, 2002) because the former uses the subsidiary call-by-value to reduce operands whereas the latter does not.

Ahead machine The ahead machine (Paolini & Ronchi Della Rocca, 1999) characterises operationally the notion of v-solvability. Similarly to strong reduction and `byValue`, the ahead machine uses call-by-value to reduce operands in redices, passing parameters 'by whnf'. Differently from strong reduction and `byValue`, operands at the right of a variable are reduced weakly. These are the original natural semantics of the ahead machine, where $k \in \{0, 1\}$:

$$\frac{m \geq 0 \quad M_i \Downarrow_a^k N_i\ _{(1 \leq i \leq m)}}{x\,M_1 \ldots M_m \Downarrow_a^k x\,N_1 \ldots N_m}\ (\textsc{Var})$$

$$\frac{N \Downarrow_a^1 N' \quad [N/x]B\,M_1 \ldots M_m \Downarrow_a^k R}{(\lambda x.B)N\,M_1 \ldots M_m \Downarrow_a^k R}\ (\textsc{Head})$$

$$\frac{}{\lambda x.B \Downarrow_a^1 \lambda x.B}\ (\textsc{Lazy}) \qquad \frac{B \Downarrow_a^0 B'}{\lambda x.B \Downarrow_a^0 \lambda x.B'}\ (\lambda 0)$$

The hybrid is $\Downarrow_a^0$ and the subsidiary is $\Downarrow_a^1$ which coincides with $\Downarrow_{bv}$. They only differ in the treatment of abstraction bodies in rules ($\lambda 0$) and Lazy.

Outermost strategy for arithmetic expressions In (Danvy & Johannsen, 2013) an innermost and an outermost strategy for reducing arithmetic expressions are defined. The authors claim that the reduction contexts for the outermost are the same as those of the innermost. However, those reduction contexts fail to define the outermost strategy as a uniquely-decomposable reduction semantics in the manner of (Felleisen, 1987). The authors have to specify an ad-hoc decomposition function which does not correspond to their reduction contexts. We show the proper reduction semantics and contexts by defining the outermost strategy as a hybrid:

$$
\begin{array}{rcll}
t & ::= & 0 \mid S(t) \mid A(t,t) & \qquad \mathbf{C}_{\mathrm{a}}[\,] ::= [\,] \mid A(\mathbf{C}_{\mathrm{a}}[\,], t) \\
t^{\mathrm{nf}} & ::= & 0 \mid S(t^{\mathrm{nf}}) & \qquad \mathbf{C}[\,] ::= [\,] \mid S(\mathbf{C}[\,]) \mid A(\mathbf{C}_{\mathrm{a}}[\,], t) \\
 & & & \qquad \mathbf{C}[A(0,t)] \to \mathbf{C}[t] \\
t^{\mathrm{a}} & ::= & 0 \mid S(t) & \qquad \mathbf{C}[A(S(t_1), t_2)] \to \mathbf{C}[S(A(t_1, t_2))]
\end{array}
$$

The reduction contexts $\mathbf{C}[\,]$ rely on contexts $\mathbf{C}_{\mathrm{a}}[\,]$ for reducing left-addends (highlighted in the definition of $\mathbf{C}[\,]$). The subsidiary strategy for left-addends—implemented by contexts $\mathbf{C}_{\mathrm{a}}[\,]$—never reduces under successor. We write t^{a} for normal left-addends.

In (Danvy & Johannsen, 2013) they hit the mark on the issue of *backward-overlapping rules* (Dershowitz, 1981; Guttag *et al.*, 1983; Geupel, 1989). An outermost strategy in the presence of backward-overlapping rules must be hybrid, since a less reducing subsidiary strategy is needed to prevent full reduction of the terms unifying with the proper sub-parts in the backward-overlapping pairs.

5.9 Related and future work

We initially distilled the reduction semantics of Section 5.2.3 from the derived reduction-based normaliser. To further ensure that the semantics corresponded to the normaliser we implemented and tested the semantics using PLT Redex (Klein *et al.*, 2012), a domain-specific language of the Racket language for programming semantics. The code is available on line.[10]

In (Danvy *et al.*, 2013) a derivation of an eval-readback full-reducing machine of Curien implementing normal order (Curien, 1993) is presented. Our single-function and their eval-readback approach require different CPS transformations. For our single-function artefacts a single-layer CPS without control delimiters is enough. For their eval-readback artefacts, either a 2-layer CPS or a single-layer CPS with control delimiters is required (Danvy *et al.*, 2013; Biernacka *et al.*, 2005). An advantage of the eval-readback approach is that the two stages (with their types, continuations, etc) are disentangled and modular. However, in

[10] `http://babel.ls.fi.upm.es/~agarcia/papers/SCICO-PEPM/normal-order.rkt`

eval-readback the set of CPS and defunctionalisation transformations get more complicated and less direct. We believe that single-function implementations are more amenable to program transformation because no specific CPS techniques nor meta-theory for delimiting control is required.

Section 11.4 of (Munk, 2008) states that a version of KN is derived from the reduction semantics of a full-reducing strategy which, although not stated explicitly, is normal order in eval-readback style in a calculus of closures. A functional correspondence is mentioned, but the intermediate refunctionalised normaliser is missing. A syntactic correspondence is also claimed, but no derivation is provided for it. We have tried to implement the suggested syntactic correspondence unsuccessfully due to several errors in the presentation. The reduction semantics in (Munk, 2008, Section 7.2) is the same as the 'preponed' semantics obtained in Section 6.9.2 and discussed in Section 5.8, save for the minor visual use of inside-out contexts. In (Munk, 2008), the calculus of closures is altered substantially in each transformation to disentangle two auxiliary continuation and meta-continuation dispatchers by means of non-standard derivation steps, which in turn permit a functional correspondence in the same spirit as (Danvy *et al.*, 2013) which uses eval-readback and two-layer CPS.

In Section 5.8 we have shown the appropriate reduction semantics in the manner of (Felleisen, 1987) of the outermost strategy for arithmetic expressions presented in (Danvy & Johannsen, 2013) by unearthing its hybrid nature.

The normalisation-by-evaluation (NBE) normaliser in (Grégoire & Leroy, 2002), which is specified by the $\mathcal{N}(T)$ strategy, uses the ZAM (Leroy, 1991) as the eval stage. We are currently studying the derivation of a machine from the operational semantics of $\mathcal{N}(T)$ presented in Section 5.8. A question to answer is whether the optimisations present in ZAM can be incorporated by program transformation. We are also studying the inter-derivation of a machine from the structural operational semantics of the **G** reduction relation, which is the full-reducing strategy of the pure lambda-value calculus of (Ronchi Della Rocca & Paolini, 2004).

5.10 Conclusions

The inter-derivation techniques (Ager *et al.*, 2003b,a; Danvy & Nielsen, 2004; Danvy, 2005; Danvy & Millikin, 2008; Biernacka & Danvy, 2007; Danvy *et al.*, 2011) can be refined to accommodate hybrid strategies, some of which are full-reducing and complete. The insight is to observe the shape invariant of the grammar of well-formed continuation stacks to recover shallow inspection, and to use single-layer CPS.

Addendum

5.11 Characterising the hybrid nature of a strategy

In Section 5.3 we commented on the hybrid style and the hybrid nature of a strategy. The hybrid character was first noted in (Sestoft, 2002), in relation to the natural semantics formalism, where a hybrid natural semantics relies on an already defined subsidiary natural semantics. As we noted in Section 5.3 there are several sources of confusion regarding whether the hybrid character is just a matter of presentation, or whether it is intrinsic to the strategy and unconnected to any of the representational concerns.

First of all, the property of being defined by several (mutually inter-)dependent natural semantics is not distinctive of the hybrid nature, but only of the hybrid style. Relying on this property to characterise the hybrid nature leads to complications, since a uniform strategy could be partitioned into several superficially different strategies that depend on each other. And then checking whether these strategies are truly the same or not could be non-trivial. Furthermore, a particular representation could make the subsidiary implicit, enabling a uniform-style definition for some strategies with hybrid nature (*i.e.*, the natural semantics in the fashion of the inner and ahead machine of (Paolini & Ronchi Della Rocca, 1999) in Section 5.3). In this example, the flattened representation of multiple applications in lambda terms, together with the unspecified number of nested applications $M\,N_1 \ldots N_n$, defines *rule schemata* for Con and Neu. These rule schemata entail an implicit inspection of the input term down to the leftmost operator. For an inference system with such rule schemata to be syntax directed, the deep inspection of nested applications has to be instrumented as an ancillary function or strategy that locates the leftmost abstraction in an applicative context, which is akin to the call-by-name strategy. The subsidiary could be duly unearthed, as in the hybrid-style definition of $\Downarrow_{no}$ in Figure 5.2.

Problems would also arise when trying to characterise the hybrid nature by the hybrid style of the SOS. For instance, it is possible to define in uniform-style a strategy with hybrid nature. Consider the (uniform-style) definition of $\rightarrow_{no}$ in (García-Pérez & Nogueira, 2013),

which we depict below:

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

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

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

Due to the hybrid nature, the subsidiary could be unearthed and the above could be transformed into a hybrid-style definition in the way described next. Rules (β) and ($\mu 1$) are only applicable when the input term is not in whnf, and rules ($\mu 2$), (ν), and (ξ) are only applicable otherwise. More interestingly, the second premiss $M \to_{no} M'$ in ($\mu 1$) consists of a derivation that only involves rules (β) and ($\mu 1$), because of the side condition $M \notin \mathsf{WHNF}$ in ($\mu 1$)'s first premiss. If we traced a dependency directed graph with the rules in the uniform-style definition—the dependency standing for the possible occurrence of another rule in the premisses of a given rule in a derivation—rules (β) and ($\mu 1$) alone would constitute a sub-graph without exit edges. Thus, rules (β) and ($\mu 1$) could be split away into a separate strategy—which coincides with call-by-name—and the side condition $M \notin \mathsf{WHNF}$ could be removed, as in the hybrid-style definition of $\to_{no}$ in Figure 5.1. At the core of this issue is the check on whnf-ness. Implementing this check as a separate function would be akin to splitting the call-by-name strategy away.

Unearthing the subsidiary could be even more challenging. Consider the SOS of normal order in (Pierce, 2002, p. 502), which we depict below:

$$\frac{}{(\lambda x.B)N \to_{no} [N/x]B}\ (\text{E-AppAbs}) \qquad \frac{M \in \mathsf{NA} \quad M \to_{no} M'}{M\,N \to_{no} M'\,N}\ (\text{E-App1})$$

$$\frac{M \in \mathsf{NNF} \quad N \to_{no} N'}{M\,N \to_{no} M\,N'}\ (\text{E-App2}) \qquad \frac{B \to_{no} B'}{\lambda x.B \to_{no} \lambda x.B'}\ (\text{E-Abs})$$

$$\begin{array}{lcl} \mathsf{NA} & ::= & x \mid \Lambda\,\Lambda \\ \mathsf{NNF} & ::= & x \mid \mathsf{NNF}\,\mathsf{NF} \end{array}$$

NA stands for non-abstraction terms. Recall from Section 5.2 that NNF stands for neutral terms in normal-form, which are non-abstraction normal-forms. Rules (E-AppAbs), (E-App2) and (E-Abs) coincide respectively with rules (β), (ν) and (ξ) above. We notice the following equivalence:

$$\begin{array}{rcll} \mathsf{NA} & = & \Lambda \setminus \{\lambda x.\Lambda\} & \\ & = & (\Lambda \setminus \mathsf{WHNF}) \cup (\mathsf{WHNF} \setminus \{\lambda x.\Lambda\}) & (5.1) \end{array}$$

We can split rule (E-App1) into two, using each addend of Equation 5.1 as a side condition. This precisely gives rules ($\mu 1$) and ($\mu 2$) above and now (β) and ($\mu 1$) could be split away into call-by-name. This time, the subsidiary was obscured because rules ($\mu 1$) and ($\mu 2$) were entangled into single E-App1 rule. The whnfs are still conspicuous, but this time in a negative way, *i.e.*, $\mathsf{WHNF} = (\Lambda \setminus \mathsf{NA}) \cup \mathsf{NNF}$.

Similar to the case for natural semantics, the representation of lambda terms can also obscure the subsidiary in the SOS. Consider the definition of normal order as the instantiation of the principal reduction machine $\rightarrow^p_\Lambda$ of (Ronchi Della Rocca & Paolini, 2004), which we depict below:

$$\frac{B \rightarrow^p_\Lambda B'}{\lambda x.B \rightarrow^p_\Lambda \lambda x.B'}\ (p1) \qquad \frac{i = \min\{j \leq m | M_j \notin \mathsf{NF}\} \quad M_i \rightarrow^p_\Lambda M_i'}{x\, M_1 \ldots M_m \rightarrow^p_\Lambda x\, M_1 \ldots M_i' \ldots M_m}\ (p2)$$

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

The flattened representation of multiple applications in rules $p2$ and $p3$ entails an implicit inspection of the input term down to the leftmost operator. Rule $p3$ alone defines call-by-name. Unsurprisingly, rules $p1$ and $p3$ match the syntax of the whnfs, *i.e.*, the intermediate irreducible forms.

The (context-based) reduction semantics is also prone to the representation issue that could make the subsidiary implicit. The normal order reduction contexts could be given as the following uniform-style grammar:

$$\mathbf{C}_{no}[\,] \quad ::= \quad [\,] \mid ([\,]\{\Lambda\}^*)\Lambda \mid \mathsf{NNF}\,\mathbf{C}_{no}[\,]\{\Lambda\}^* \mid \lambda x.\mathbf{C}_{no}[\,]$$

The flattened representation of multiple applications in the regular expressions of the second and third productions entails an implicit inspection of the input term down to the leftmost operator. The first and second productions alone define the reduction contexts of call-by-name. A hybrid-style definition of these reduction contexts is given in Figure 5.4.

The hybrid nature can be characterised with precision by considering the language of reduction context of the strategy (see Section 5.3). The language of reduction contexts is intrinsic to the strategy itself, and unconnected to any presentation of the strategy. Although the hybrid character is a matter of nature, a hybrid strategy could be partitioned into hybrid and subsidiaries in different ways. Recall form Section 5.3 that by adding the hole $[\,]$ to contexts $\mathbf{C}_{ne}[\,]$ of Section 5.2.1, an alternative definition of normal order is given which consists of inter-dependent hybrid strategies *no* and (modified) *ne*. The requirement that a strategy has to include the hole in its reduction contexts is primordial for the different partitions to meet with the formal characterisation of hybrid nature.

5.12 Hybrids and NBE

As we have commented at length in Section 5.9, the NBE-style artefacts could be inter-derived by using 2CPS or by using single-layer CPS and control delimiters (Munk, 2008;

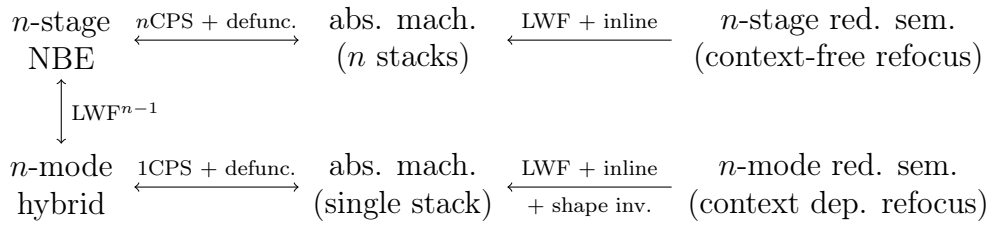
Danvy *et al.*, 2013). Our solution in Chapter 5 provides an alternative path to (Munk, 2008; Danvy *et al.*, 2013) by considering the hybrid counterparts of the NBE-style artefacts. In our solution all the intermediate artefacts use single-layer CPS, there is no need for control delimiters, and the abstract machine has a single control stack.

The works (Munk, 2008; Danvy *et al.*, 2013) do not include the detailed inter-derivation, and the code is not publicly available. We have followed the indications in (Munk, 2008; Danvy *et al.*, 2013) and have adapted their solution to the normal order strategy in a plain (*i.e.*, without closures) lambda calculus. Their solution does not connect the SOS with the (context-based) reduction semantics, and the latter has to be contrived. We remedy this for normal order by providing the derivation of the reduction semantics from a search function characteristic of the SOS in NBE-style. The code can be found on line.[11] To the best of our knowledge, the 2CPS approach has not been explored enough as to connect the SOS formalism, and the normal order strategy in plain lambda calculus has not been considered before.

Connecting the search function with the reduction semantics requires some ingenuity. For the simplification step (see Section 5.4.4) both spaces of continuations have to be simplified. However, this simplification cannot be performed in cascade (*i.e.*, the intermediate artefact with only one simplified continuation space does not type in ML) and hence the two simplification steps have to be tackled together, which obscures the equivalence of the resulting artefact. The defunctionalisation step is also not completely straightforward. In the apply function for the continuation proper (*i.e.*, not for the meta-continuation) the init case glues together the eval stage with the readback stage in the NBE approach. It is not quite direct to come up with the code for the defunctionalised artefact, which has to instrument a recursive invocation of the whole search/decomposition process.

To exemplify the applicability of our technique, and to outline the similitudes and differences with respect to NBE, we have applied the hybrid and the NBE approach to yet another hybrid semantics, the outermost strategy for arithmetic expressions in (Danvy & Johannsen, 2013). Section 5.8 presents the context-based reduction semantics of this strategy. The code with both transformations can be found on line.[12] [13]

The following diagram illustrates the two alternative paths:



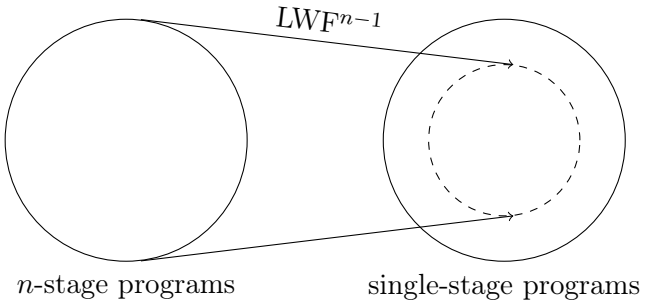
Light-weight fusion by fixed-point promotion (LWF) (Ohori & Sasano, 2007) connects the

[11] `http://babel.ls.fi.upm.es/~agarcia/papers/SCICO-PEPM/normal-order-2cps.sml`
[12] `http://babel.ls.fi.upm.es/~agarcia/phd-thesis/outermost-hybrid.sml`
[13] `http://babel.ls.fi.upm.es/~agarcia/phd-thesis/outermost-2cps.sml`

NBE artefacts with the (single-function) hybrid artefacts. The LWF has to be applied $n-1$ times to fuse a n-stage artefact.

In the upper part of the diagram the refocus function is context-free, but the intermediate artefacts have nested continuation spaces and the number of control stacks in the abstract machine grows with the number of stages. In the lower part of the diagram the shape invariant of the control stack allows to remove the dependency introduced by refocusing, the intermediate artefacts have a single space of continuations and the abstract machine has a single stack. Our solution scales smoothly to any number of (inter-)dependent modes.

Danvy & Millikin (2009) comment about Dijkstra's case against the GOTO statement (Dijkstra, 1968). The implicit message is not that 'GOTO statements should be considered harmful', but that 'one should be mindful about staying in or straying from the image of the compiler' when compiling a structured program without GOTO into some unstructured program where the GOTO is used to implement the control structures of the former. As Danvy & Millikin (2009) argue, 'straying with good reasons is a clear indication that a useful control structure is missing'. This comment applies to other program transformations, like CPS transformation or defunctionalisation. Here we put the emphasis on the LWF. Consider the following diagram:



Programs with n stages are fused into single-stage programs. The inter-derivation in (Munk, 2008; Danvy *et al.*, 2013) departs from the programs in the left of the diagram. Our inter-derivation departs from the image of LWF in the right part of the diagram, where certain transformation steps are simpler and more amenable to automation. We aim at staying in the image of LWF. The shape invariant of the control stack reflects the 'staging' control structure in the left part.

Danvy & Johannsen (2013) also explore the refocusing step in the right part of the diagram above. They remove the dependency on the context by inserting, prior to refocus, a backtracking operation that blindly reverts the current decomposition a fixed number of steps. Although effective, their solution deviates from the standard formalisms for operational semantics. The backtracking operation requires, by design, a grammar of reduction contexts that does not reflect a reduction semantics proper (Felleisen, 1987), *i.e.*, a grammar that lacks unique decomposition. Starting with a reduction semantics with unique decomposition forces the refocus function to be context dependent, and duly supersedes

the solution in (Danvy & Johannsen, 2013) since the backtracking step is implicitly performed by the decomposition function.

6

From Normal Order to the Full-Reducing Krivine Machine by Program Transformation

Todos los buenos soldados
Que asentaren a esta guerra
No quieran nada en la Tierra
Si quieren ir descansados.

Si salieren con victoria
La paga que les darán
Será que siempre ternán
En el Cielo eterna Gloria.[1]

(*La Guerra*, Mateo Flecha el Viejo)

We derive (an implementation of) Pierre Crégut's full-reducing Krivine machine KN by program transformation from search functions implementing the compatibility rules of the structural operational semantics of the normal order reduction strategy in a pure lambda calculus of closures. We arrive at a slightly optimised KN that can work with open terms. We prove the stepwise correspondence of normal order reduction in the calculus of closures and in the pure lambda calculus. We introduce a preponing step before shortcutting ephemeral expansion, and remove explicit control using the shape invariant of the grammar

[1] All the good soldiers / That attended to this battle / Should not long for anything while in Earth / If they want to stay restful. / If they come out victorious / The salary they shall receive / Is to have / Everlasting Glory in Heaven. (Translation by the author.)

of continuation stacks. Thanks to the single-function nature of our semantics we use single-layer CPS without control delimiters, as opposed to a two-layer CPS or a single-layer CPS with control delimiters.

6.1 Introduction

Languages and calculi constantly spring up. Their semantics (small-step or big-step operational, abstract machines, denotational, etc.) are typically specified on paper and their correspondences are either obviated, conjectured, or proven mathematically. In contrast, semantics can be implemented and their correspondences can be often established by program transformation (*e.g.*, (Ager *et al.*, 2003b; Danvy, 2005; Biernacka & Danvy, 2007; Danvy, 2008a; Danvy *et al.*, 2011)). A recent case in point is (Siek & Garcia, 2012) where several definitional interpreters implement the various denotational semantics of the core language of the gradually-typed lambda calculus. The correspondences with the small-step operational semantics are left as a conjecture, but they can be established by program transformation (García-Pérez *et al.*, 2014).

Another case in point, and the topic of this chapter, is the correspondence between the normal order reduction strategy (aka leftmost reduction) (Barendregt, 1984) and the full-reducing Krivine machine KN (Crégut, 2007). Normal order is the 'full-reducing' (reduces terms fully to normal form) and 'complete' (does not fail to deliver the normal form when it exists) reduction strategy of the pure lambda calculus. (A reduction strategy specifies an operational semantics, *i.e.*, the order in which reducible subterms or 'redices' (singular, 'redex') are to be reduced.)

Pierre Crégut's KN is a well-known machine that fully reduces pure lambda calculus terms. More precisely, KN is a push/enter (we borrow the terminology from (Biernacka & Danvy, 2007)) environment-based closed-terms machine that uses de Bruijn indices and levels for representing terms. Pierre Crégut proved mathematically that KN finds the normal form of a *closed* term when the normal form exists (Crégut, 2007), but the actual correspondence between KN and normal order, *i.e.*, that KN *realises* normal order, has remained unproven.

In this chapter we derive (an implementation of) KN from search functions that implement the compatibility rules (the rules that express how to navigate a term to locate a redex) of the structural operational semantics of normal order in a pure lambda calculus of closures. Thus, we prove by means of program-transformation the correspondence between the strategy and the machine. Actually, what we obtain is a slightly optimised KN that can also work with *open* terms, and is therefore suitable for use in implementations of proof assistants (Crégut, 1990; Kesner, 2007; Grégoire & Leroy, 2002).

Figure 6.1 illustrates the derivation path. The start and end points are shown in boldface. In the figure the starting point is the reduction-based normaliser obtained from the search functions. The figure extends the derivational taxonomy of (Biernacka & Danvy, 2007, p.24) and summarises the contents of Sections 6.8.2 to 6.10.4 of this chapter.

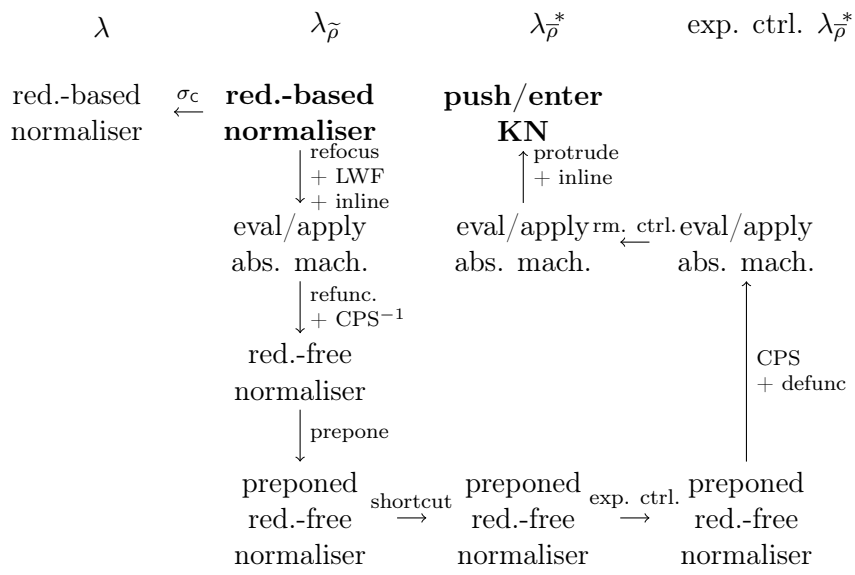


Figure 6.1: Derivation path of KN

Here is a more detailed list of the contributions:

- Our operational semantics are single-function *i.e.*, they define a single but hybrid normal order strategy that relies on a subsidiary call-by-name strategy (Section 6.4). Consequently, we can use *single-layer* CPS without control delimiters, as opposed to a two-layer CPS or a single-layer CPS with control delimiters, as found in other works (Section 6.11).

- We introduce the $\lambda_{\widetilde{\rho}}$ calculus which naturally extends the $\lambda_{\widehat{\rho}}$ calculus of Biernacka and Danvy (Biernacka & Danvy, 2007) with de Bruijn levels (present in KN), closure abstractions, and absolute indices. The latter two are required for full-reduction. Closure abstractions are required to represent closures where the redex may occur under lambda, and absolute indices are required to represent 'neutral closures', *i.e.*, non-redex closure applications. We define the substitution function σ_{c} that simulates capture-avoiding substitution in the pure lambda calculus, define the structural operational semantics of normal order in $\lambda_{\widetilde{\rho}}$, and prove that normal order reduction in $\lambda_{\widetilde{\rho}}$ mirrors *stepwise* normal order reduction in the pure lambda calculus.

- We define the small-step structural operational semantics of normal order in $\lambda_{\widetilde{\rho}}$ and derive from the search functions that implement the compatibility rules the reduction-based normaliser, and from it the reduction-free normalisers. In other words, we derive the context-based and the big-step natural semantics of normal order in $\lambda_{\widetilde{\rho}}$.

- We use the *refined* inlining-of-iterate-function step (García-Pérez & Nogueira, 2013, 2014a) that exploits the shape invariant of the continuation stack to recover shallow inspection of the environment-based eval/apply abstract machine derived from the reduction-based normaliser.

- We introduce a non-standard but straightforward 'preponing' step that is needed for shortcut optimisation. We prove that preponing is equivalence-preserving (Section 6.9.2).

- After shortcut optimisation, which takes us to a version of $\lambda_{\widetilde{\rho}}$ without ephemerals[2] that we call $\lambda_{\widetilde{\rho}}^{*}$, we have to introduce explicit control to combine two reduction-free normalisers into one. We remove explicit control by constructing the grammar of the machine's well-formed stacks and then observing the correlation between the control characters and the occurrence of certain constructors on the top of the well-formed stack. (This is another application of constructing grammars of continuation stacks.)

- We obtain is a slightly optimised KN that can also work with *open* terms and that does not need to carry lambda levels in ground terms.

We have written all the code of the derivation in Standard ML, the traditional programming language of derivation papers. Though semi-automatically obtained, the code is rather long (we include all steps in detail) and language-specific. Therefore we present the semantic artefacts in mathematical notation and simply name in the chapter the functions implementing the artefacts in the code, which is available on line.[3] We have tested the code. We have not verified the transformations using a proof assistant for several reasons. First, most transformation steps are standard and easy to check by readers familiar with derivation papers in Standard ML, but hard to prove using a proof assistant. Second, involving a proof assistant or a dependently-typed language would result in a different text for a different readership. We would have to explain additional proof techniques and the particulars of the assistant. See for instance (Swierstra, 2012) where several techniques (logical relations, etc) have to be introduced to obtain the weak-reducing KAM machine for the simply typed lambda calculus.

6.2 Structure of the chapter

- In Section 6.3 we refresh terminology and definitions of the lambda calculus and of derivation by program transformation of semantic artefacts.

- In Section 6.4 we define the operational semantics (structural, natural, context-based, and eval-readback) of normal order in the pure lambda calculus.

[2]The 'ephemeral' terminology is introduced in Section 6.5.

[3]`http://babel.ls.fi.upm.es/~agarcia/papers/SCICO-PPDP`

- In Section 6.5 we describe the closure calculus $\lambda_{\widehat{\rho}}$ of Biernacka and Danvy, the small-step operational semantics of call-by-name in $\lambda_{\widehat{\rho}}$, and the substitution function σ connecting values in $\lambda_{\widehat{\rho}}$ with values in the pure lambda calculus (Biernacka & Danvy, 2007).

- In Section 6.6 we describe the full-reducing Krivine machine KN of (Crégut, 2007).

- In Section 6.7 we introduce our $\lambda_{\widetilde{\rho}}$ calculus, define the substitution function σ_{C} that simulates capture-avoiding substitution in the pure lambda calculus, define the structural operational semantics of normal order in $\lambda_{\widetilde{\rho}}$, and prove that normal order reduction in $\lambda_{\widetilde{\rho}}$ mirrors *stepwise* normal order reduction in the pure lambda calculus.

- In Section 6.8.1 we derive a reduction-based normaliser from the search functions that implement the compatibility rules of the structural operational semantics of normal order in $\lambda_{\widetilde{\rho}}$. We show the reduction semantics implemented by the reduction-based normaliser and prove that it satisfies unique-decomposition.

- In Section 6.8.2 we transform the reduction-based normaliser to an environment-based eval/apply abstract machine with *shallow inspection property* (the machine's dispatcher does not inspect the current continuation argument and therefore can be refunctionalised). In the transformation we employ our *refined* inlining-of-iterate-function step (García-Pérez & Nogueira, 2013, 2014a). This step exploits the shape invariant of the continuation stack to recover shallow inspection.

- In Section 6.8.3 we apply refunctionalisation and inverse CPS to the previous machine to obtain the reduction-free normalisers that implement the natural semantics of normal order in $\lambda_{\widetilde{\rho}}$.

- In Section 6.9 we shortcut ephemeral expansion. Previously we introduce a required preponing step that prepones some normal order steps to call-by-name. We explain why preponing is needed and present a proof that it is equivalence-preserving.

- In Section 6.10.1 we transform the shortcut reduction-free normalisers into a single reduction-free normaliser by introducing explicit control which encodes the different treatment of abstractions. We then transform in Section 6.10.2 the normaliser by defunctionalisation and CPS transformation to an environment-based eval/apply machine with explicit control.

- In Section 6.10.3 we remove explicit control by constructing the grammar of the machine's well-formed stacks and then observing the correlation between the control characters and the occurrence of certain constructors on the top of the well-formed stack.

- In Section 6.10.4 we transform the eval/apply machine to a push/enter optimised KN, and discuss and remove some minor visual differences with the original.

$$
\begin{array}{rcl}
\uparrow_n^k m & = & \begin{cases} m+n & \text{if } m \geq k \\ m & \text{otherwise} \end{cases} \\
\uparrow_n^k (\lambda.B) & = & \lambda.\uparrow_n^{k+1} B \\
\uparrow_n^k (M\,N) & = & (\uparrow_n^k M)(\uparrow_n^k N)
\end{array}
$$

$$
\begin{array}{rcl}
[T/n]m & = & \begin{cases} \uparrow_n^0 T & \text{if } m = n \\ m-1 & \text{if } m > n \\ m & \text{otherwise} \end{cases} \\
[T/n](\lambda.B) & = & \lambda.[T/n+1]B \\
[T/n](M\,N) & = & ([T/n]M)([T/n]N)
\end{array}
$$

Figure 6.2: Capture-avoiding substitution function for de Bruijn indices

6.3 Preliminaries

We consider the pure untyped lambda calculus with de Bruijn indices (Barendregt, 1984), hereafter λ, whose terms are defined by the grammar $\Lambda ::= n \mid (\lambda.\Lambda) \mid (\Lambda\,\Lambda)$. A natural number n represents a variable bound to the nth lambda starting from 0, or to a free variable when n is greater than or equal to the nesting level. For example, the abstraction $\lambda.0$ is the identity function whereas $\lambda.1$ is a constant function delivering free 0 when the function is applied to an operand. Uppercase, often primed, letters M, M', N, B, etc, will range over elements of Λ. We use the standard precedence and association convention: applications associate to the left and abstraction binds tighter than application. The definition of capture-avoiding substitution for de Bruijn indices, $[T/m]M$, is shown in Figure 6.2. Function $[T/m]M$ finds occurrences of the variable to substitute for in M. For each substitution, function $\uparrow_n^k T$ shifts the indices of the variables in the subject T that require it (Crégut, 2007). The reader must be familiar with the usual notions of bound and free variables, redices $(\lambda.B)M$, syntactic equivalence $\equiv$, β-contraction, and relations $\rightarrow_\beta$, $\rightarrow_\beta^*$, and $=_\beta$. A reduction strategy s of λ is a (partial) function that is a sub-relation of $\rightarrow_\beta^*$. We write $\rightarrow_s$ and $\Downarrow_s$ for the small-step and big-step definitions of s. We use relational $M \Downarrow_s N$ and functional $\Downarrow_s(M) = N$ notation interchangeably, and use function composition when appropriate, *e.g.*, $(\Downarrow_t \circ \Downarrow_s)(M) = \Downarrow_t(\Downarrow_s(M))$.

We make extensive use of grammars in Extended Backus-Naur Form with regular expressions for optionals and sequences. For example, $\mathsf{NF} ::= \lambda.\mathsf{NF} \mid n\,\{\mathsf{NF}\}^*$ defines the set of normal forms. Some sentential forms of the second production are n, $n\,\mathsf{NF}$, $n\,\mathsf{NF}\,\mathsf{NF}$, etc., which respectively associate as n, $(n\,\mathsf{NF})$, $((n\,\mathsf{NF})\,\mathsf{NF})$, etc, according to the convention. The context grammar $\mathbf{C}[\,] ::= [\,]\{\Lambda\}^?$ derives the contexts $[\,]$ (the empty context or hole) and $[\,]\,\Lambda$ (an application with a hole in operator position and an arbitrary term in operand position).

In the text we use 'whnf' and 'nf' to abbreviate 'weak head normal form' and 'normal form' respectively. The set of whnfs and nfs is defined in Figure 6.3. Notice that a term

$$\frac{}{(\lambda.B)N \to_{no} [N/0]B}\ (\beta) \qquad \frac{M \notin \mathsf{WHNF} \quad M \to_{no} M'}{M\,N \to_{no} M'\,N}\ (\mu 1)$$

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

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

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{WHNF} & ::= & \lambda.\Lambda & \mathsf{NF} & ::= & \lambda.\mathsf{NF} \\ & | & n\{\Lambda\}^* & & | & n\{\mathsf{NF}\}^* \end{array}$$

Figure 6.3: Structural operational semantics of normal order (García-Pérez & Nogueira, 2013).

in nf is also in whnf.

6.4 Normal order in all substitution-based styles

Normal order is a full-reducing and complete strategy of the pure lambda calculus. It is defined by the slogan 'contract the leftmost redex first' understanding 'leftmost' as in (Curry & Feys, 1958) or 'leftmost-outermost' when referring to the redex's position in the abstract syntax tree of the term. Normal order is a *hybrid* strategy (Sestoft, 2002; García-Pérez & Nogueira, 2013, 2014a), *i.e.*, it relies on a subsidiary strategy, call-by-name, to reduce particular subterms. More precisely, given an abstraction $\lambda.B$, normal order 'goes under lambda' and reduces B fully to nf. However, given an application MN, if M is β-reducible in an arbitrary number of steps to an arbitrary abstraction $\lambda.B$, at that point the leftmost-outermost redex is $(\lambda.B)N$ and normal order must reduce that redex and not $\lambda.B$. Since normal order reduces abstractions fully, it must reduce operators in applications using a less reducing strategy, one that does not reduce abstractions and that, like normal order, is 'non-strict', *i.e.*, does not reduce operands in redices before substitution. In other words, it must rely on call-by-name. By choosing the leftmost-outermost redex, normal order discards unneeded potentially divergent subterms. For example, given the term $(\lambda.0\,\Omega)(\lambda.1)$ where Ω is a divergent subterm, normal order reduces that leftmost-outermost redex to $(\lambda.1)\Omega$, and this term to 0, discarding Ω.

Figures 6.3 and 6.4 show the structural and natural operational semantics of normal order respectively. In the structural small-step there is no rule for variables because these are in nf. There are four rules for applications. The first (β) is well-known. It applies

$$\frac{}{n \Downarrow_{bn} n}\ (\textsc{Var}_{bn}) \qquad \frac{}{\lambda.B \Downarrow_{bn} \lambda.B}\ (\textsc{Lam}_{bn})$$

$$\frac{M \Downarrow_{bn} M' \quad M' \equiv \lambda.B \quad [N/0]B \Downarrow_{bn} B'}{M\ N \Downarrow_{bn} B'}\ (\textsc{Red}_{bn})$$

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

$$\frac{}{n \Downarrow_{no} n}\ (\textsc{Var}_{no}) \qquad \frac{B \Downarrow_{no} B'}{\lambda.B \Downarrow_{no} \lambda.B'}\ (\textsc{Lam}_{no})$$

$$\frac{M \Downarrow_{bn} M' \quad M' \equiv \lambda.B \quad [N/0]B \Downarrow_{no} B'}{M\ N \Downarrow_{no} B'}\ (\textsc{Red}_{no})$$

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

Figure 6.4: Natural operational semantics of normal order (García-Pérez & Nogueira, 2013).

when the operator is an abstraction. The second ($\mu 1$) says the redex must be searched for in the operator if the operator is not in whnf. The third rule ($\mu 2$) says the redex must be searched for in the operator if it is a whnf but not a nf nor an abstraction (if an abstraction then (β) is applicable). Finally, (ν) says the redex must be searched for in the operand if the operator is a nf but not an abstraction. The outermost application does not reduce to a redex and so the redex must be searched for in the operand. Although a nf is also a whnf, rules ($\mu 2$) and (ν) are non-overlapping because the third premiss in ($\mu 2$) is not the case when $M \in \mathsf{NF}$. Hereafter we shall refer to variables and non-redex applications as *neutral terms*, defined by the regular expression $n\,\{\Lambda\}^*$. Last, rule (ξ) provides structural compatibility with abstractions, that is, 'go under lambda'.

In the structural operational semantics of Figure 6.3 the dependency of normal order on call-by-name is implicit as a sub-relation: rules (β) and ($\mu 1$) taken in isolation make up call-by-name. The dependency can be made explicit by unearthing the sub-relation, *i.e.*, by breaking up (β) and ($\mu 1$) into call-by-name and normal order rules, with the dependency

on call-by-name now explicit in rule ($\mu 1_{no}$):

$$\frac{}{(\lambda.B)N \to_{bn} [N/0]B}\ (\beta_{bn}) \qquad \frac{}{(\lambda.B)N \to_{no} [N/0]B}\ (\beta_{no})$$

$$\frac{M \notin \mathsf{WHNF} \quad M \to_{bn} M'}{M\,N \to_{bn} M'\,N}\ (\mu 1_{bn}) \qquad \frac{M \notin \mathsf{WHNF} \quad M \to_{bn} M'}{M\,N \to_{no} M'\,N}\ (\mu 1_{no})$$

The dependency on call-by-name is explicit in the big-step natural semantics where normal order $\Downarrow_{no}$ relies on call-by-name $\Downarrow_{bn}$ to reduce operators to whnf (first premiss of rules $\textsc{Red}_{no}$ and $\textsc{Neu}_{no}$), and then fully reduces the resulting redex (rule $\textsc{Red}_{no}$) or the resulting neutral (rule $\textsc{Neu}_{no}$). Finally, $\textsc{Lam}_{no}$ says that normal order goes under lambda and $\textsc{Var}_{no}$ says normal order is an identity on variables. In contrast, call-by-name does not go under lambda and does not reduce operands in neutral terms.[4]

The structural and natural semantics in Figures 6.3 and 6.4 are *single-function*, that is, $\to_{no}$, $\Downarrow_{no}$, and $\Downarrow_{bn}$ are partial functions. There is an alternative two-function *eval-readback* (Grégoire & Leroy, 2002) approach that defines reduction as the composition of two single functions, namely, an 'eval' function that delivers intermediate results, and a 'readback' function that distributes reduction over the subterms of the intermediate result. The eval-readback approach is a degenerate case of normalisation-by-evaluation (Aehlig & Joachimski, 2004) in which the value domain is the set of terms, and readback is 'reify' without the translation from domain values to terms. (The two-function nature of eval-readback definitions is also present in their corresponding small-step semantics, where a reduction sequence consists of nested concatenations of eval and readback sequences.) Normal order is defined in eval-readback style as the composition $\Downarrow_{rn} \circ \Downarrow_{bn}$ where $\Downarrow_{bn}$ is eval and $\Downarrow_{rn}$ is readback:

$$\frac{}{n \Downarrow_{rn} n}\ \textsc{Var}_{rn} \qquad \frac{B \Downarrow_{bn} B' \quad B' \Downarrow_{rn} B''}{\lambda.B \Downarrow_{rn} \lambda.B''}\ \textsc{Lam}_{rn}$$

$$\frac{M \Downarrow_{rn} M' \quad N \Downarrow_{bn} N' \quad N' \Downarrow_{rn} N''}{M\,N \Downarrow_{rn} M'\,N''}\ \textsc{App}_{rn}$$

Readback takes input terms in whnf (no redex at the outermost level) which explains the lack of a contraction rule for it. The equivalence between single-function and eval-readback, namely $\Downarrow_{no} = \Downarrow_{rn} \circ \Downarrow_{bn}$, can be proven by induction on derivations, or by program transformation using lightweight fusion by fixed-point promotion (Ohori & Sasano, 2007) (Section 6.9.2).

Now to the context-based reduction semantics. In addition to the grammar of terms and normal forms there is a grammar for reduction contexts and a contraction rule that

[4]Call-by-name in the pure untyped lambda calculus differs from call-by-name in the applied and implicitly typed calculus of (Plotkin, 1975) (which also assumes closed input terms) precisely in its treatment of neutral terms (Sestoft, 2002, p.421).

applies (β) within the context hole.

$$\begin{array}{lll} \text{Red. context:} & \mathbf{C}_{no}[\,] ::= [\,] \mid \mathbf{C}_{bn}[\,]\,\Lambda \mid \lambda.\mathbf{C}_{no}[\,] \mid \mathbf{C}_{ne}[\,] \\ & \mathbf{C}_{bn}[\,] ::= [\,] \mid \mathbf{C}_{bn}[\,]\,\Lambda \\ & \mathbf{C}_{ne}[\,] ::= n\{\mathsf{NF}\}^*\,\mathbf{C}_{no}[\,] \mid \mathbf{C}_{ne}[\,]\,\Lambda \\ \text{Contraction:} & \mathbf{C}_{no}[(\lambda.B)N] \to_{no} \mathbf{C}_{no}[[N/0]B] \end{array}$$

Given a term M, it is either in nf or is uniquely decomposed into a context, derived from non-terminal $\mathbf{C}_{no}[\,]$, and a redex within the hole. For example, the term $\lambda.(\lambda.0)0$ is decomposed into $\lambda.[(\lambda.0)0]$ with the context $\lambda.[\,]$ grammatically derived as follows: $\mathbf{C}_{no}[\,] \Rightarrow \lambda.\mathbf{C}_{no}[\,] \Rightarrow \lambda.[\,]$. The unique decomposition of $\mathbf{C}_{no}[\,]$ is proven by structural induction on the input term (see Theorem 6.8.1 in Section 6.8.1). The dependency of normal order on call-by-name is indicated by the presence of call-by-name sub-contexts $\mathbf{C}_{bn}[\,]$.

6.5 Closures and environment machines

The operational semantics defined in Section 6.4 are all substitution-based, *i.e.*, rely on the traditional meta-level substitution function $[N/n]B$. But KN is an environment-based machine that works with *closures* $M[\rho]$ consisting of a term M with an environment ρ that maps M's variables to corresponding bindings. Usually, the environment is a list of closures, such that de Bruijn indices act as lexical offsets (starting with 0) that point to the appropriate binding in the environment.

The $\lambda_{\widehat{\rho}}$ calculus of Biernacka and Danvy (Biernacka & Danvy, 2007) extends the pure lambda calculus with definitions for closures C and environments ρ. This calculus is itself an extension of Curien's calculus of closures λ_ρ. Here is their respective syntax for terms, adapted from (Biernacka & Danvy, 2007) to our own notation explained below.

$$\lambda_\rho \quad \begin{array}{lll} \mathsf{C} & ::= & \Lambda[\rho] \\ \rho & ::= & \epsilon \mid \mathsf{C} : \rho \end{array} \qquad\qquad \lambda_{\widehat{\rho}} \quad \begin{array}{lll} \mathsf{C} & ::= & \Lambda[\rho] \mid \mathsf{C} \cdot \mathsf{C} \\ \rho & ::= & \epsilon \mid \mathsf{C} : \rho \end{array}$$

In the $\lambda_{\widehat{\rho}}$ calculus we have *proper closures* $\Lambda[\rho]$ and *closure applications* $\mathsf{C}\cdot\mathsf{C}$. We use a left-associative explicit closure application operator $\cdot$ which is elided in (Biernacka & Danvy, 2007). In both calculi, an environment ρ is either empty, which we denote by ϵ, or a list of colon-separated closures. In Curien's calculus the β-rule is $((\lambda.B)N)[\rho] \to B[N[\rho] : \rho]$ which pushes the operand as a closure on the environment. A rule for variables is introduced $n[\rho] \to n^{\text{th}}(\rho)$ to deliver the nth binding in the environment.

Both λ_ρ and $\lambda_{\widehat{\rho}}$ assume *closures without free variables*, *i.e.*, the term is closed by the environment and a binding is always found. (Input terms are closed and reduction delivers *values*: variables or abstractions.) As noted in (Biernacka & Danvy, 2007), small-step reduction relations cannot be expressed in λ_ρ and a natural solution is to extend λ_ρ with closure application together with an *ephemeral expansion* rule $(M\,N)[\rho] \to M[\rho] \cdot N[\rho]$ that

$$\frac{}{(\lambda.B)[\rho]\cdot \mathsf{N} \to_{\widehat{bn}} B[\mathsf{N}:\rho]}\ (\beta_{\widehat{\rho}})$$

$$\frac{\mathsf{M} \to_{\widehat{bn}} \mathsf{M}'}{\mathsf{M}\cdot\mathsf{N} \to_{\widehat{bn}} \mathsf{M}'\cdot\mathsf{N}}\ (\mu_{\widehat{\rho}})$$

$$\frac{}{n[\rho] \to_{\widehat{bn}} n^{\text{th}}(\rho)}\ (\textsc{Var}_{\widehat{\rho}})$$

$$\frac{}{(M\,N)[\rho] \to_{\widehat{bn}} M[\rho]\cdot N[\rho]}\ (\textsc{App}_{\widehat{\rho}})$$

$$\begin{array}{ll} \mathbf{C}_{\widehat{bn}}[\,] & ::= \quad [\,] \mid \mathbf{C}_{\widehat{bn}}[\,]\cdot \mathsf{C} \\ \mathbf{C}_{\widehat{bn}}[(M\,N)[\rho]] & \to_{\widehat{bn}}\ \mathbf{C}_{\widehat{bn}}[M[\rho]\cdot N[\rho]] \\ \mathbf{C}_{\widehat{bn}}[n[\rho]] & \to_{\widehat{bn}}\ \mathbf{C}_{\widehat{bn}}[n^{\text{th}}(\rho)] \\ \mathbf{C}_{\widehat{bn}}[(\lambda.B)[\rho]\cdot \mathsf{N}] & \to_{\widehat{bn}}\ \mathbf{C}_{\widehat{bn}}[B[\mathsf{N}:\rho]] \end{array}$$

Figure 6.5: Structural (left) and reduction semantics (right) of call-by-name in $\lambda_{\widehat{\rho}}$ (adapted from (Biernacka & Danvy, 2007)).

distributes the environment by constructing an ephemeral closure application.[5] The β-rule now operates on closure applications: $(\lambda.B)[\rho]\cdot \mathsf{N} \to B[\mathsf{N}:\rho]$. (We use uppercase sans-serif letters M, N, etc, for closure terms to save us from contriving environment symbols. For instance, N above stands for some closure $N[\rho']$ with some environment ρ'.)

To illustrate, consider a closed term $(\lambda.0)N$. It is initialised to a closure $((\lambda.0)N)[\epsilon]$ and reduced as follows:

$$((\lambda.0)N)[\epsilon] \to (\lambda.0)[\epsilon]\cdot N[\epsilon] \to 0[N[\epsilon]:\epsilon] \to 0^{\text{th}}([N[\epsilon]:\epsilon]) \equiv N[\epsilon] \to \ldots$$

The simulation of λ_ρ reductions in $\lambda_{\widehat{\rho}}$ is proven in (Biernacka & Danvy, 2007).

6.5.1 Call-by-name semantics and environment-based machine

The structural and context-based small-step operational semantics of call-by-name in $\lambda_{\widehat{\rho}}$ are shown in Figure 6.5 (adapted from (Biernacka & Danvy, 2007) to our notation). Observe in the reduction semantics that closure application enables the definition of reduction contexts for closures. The redex can now occur in the operator side of a closure application. In (Biernacka & Danvy, 2007), a reduction-based normaliser for this reduction semantics is implemented, and an environment-machine derived. The ephemeral expansion step is shortcut, getting rid of the closure application. The machine obtained is the call-by-name KAM machine (Crégut, 1990).

In (Biernacka & Danvy, 2007, p.9) a substitution function σ is defined that relates values in $\lambda_{\widehat{\rho}}$ with values in λ by forcing all the delayed substitutions. The function is depicted in Figure 6.6. However, the stepwise connection between $\lambda_{\widehat{\rho}}$ and λ via σ is not proven. The function carries a *lexical adjustment* parameter k that is incremented when

[5] 'Ephemeral' in the sense that closure applications are shortcut when deriving big-step artefacts (Biernacka & Danvy, 2007).

$$
\begin{array}{rcl}
\sigma(\mathsf{C}, \mathbb{N}) & \rightarrow & \Lambda \\
\sigma(n[\rho], k) & = & \begin{cases} n & \text{if } n \leq k \\ \sigma((n-k)^{\text{th}}(\rho), 0) & \text{if } n > k \end{cases} \\
\sigma((\lambda.B)[\rho], k) & = & \lambda.\sigma(B[\rho], k+1) \\
\sigma((M\ N)[\rho], k) & = & (\sigma(M[\rho], k))(\sigma(N[\rho], k)) \\
\sigma(\mathsf{M} \cdot \mathsf{N}, k) & = & (\sigma(\mathsf{M}, k))(\sigma(\mathsf{N}, k))
\end{array}
$$

Figure 6.6: Substitution function in (Biernacka & Danvy, 2007)

going under lambda (second clause). Integers $n \leq k$ stand for occurrences of formal parameters of abstractions that *have not* been applied to an operand. Integers $n > k$ are occurrences of formal parameters of abstractions that *have* been applied to an operand and thus have a binding in the environment (recall $\lambda_{\widehat{\rho}}$ assumes closures without free variables). For these variables the index is adjusted to $n-k$, and substitution is applied on the binding with the lexical adjustment reset to zero. The environment and the lexical adjustment are duplicated for application closures and closure applications (third and fourth clauses). The lexical adjustment discipline faithfully implements substitution for closures without free variables.

6.6 Crégut's full-reducing Krivine machine

The full-reducing machine KN (adapted from (Crégut, 2007) to our notation), is the target of our derivation. KN is a first-order transition function which operates on a triple consisting of a closure C, a continuation stack S, and a lambda level l that keeps track of the current nesting level at which reduction is taking place.

$$
\lambda_{\overline{\rho}} \quad \begin{array}{rcl} \mathsf{C} & ::= & \Lambda[\rho] \mid \overline{n} \mid \lfloor \Lambda, l \rfloor \\ \rho & ::= & \epsilon \mid \mathsf{C} : \rho \end{array}
$$

	$\lambda.0((\lambda.0)0)$	(1) init
$\to$	$((\lambda.0((\lambda.0)0))[\epsilon], \epsilon, 0)$	(6) unapplied abstraction
$\to$	$(0((\lambda.0)0)[\overline{1} : \epsilon], \lambda : \epsilon, 1)$	(4) application
$\to$	$(0[\overline{1} : \epsilon], ((\lambda.0)0)[\overline{1} : \epsilon] : \lambda : \epsilon, 1)$	(3) retrieve top binding
$\to$	$(\overline{1}, ((\lambda.0)0)[\overline{1} : \epsilon] : \lambda : \epsilon, 1)$	(7) embed index into ground
$\to$	$(\lfloor 0, 1 \rfloor, ((\lambda.0)0)[\overline{1} : \epsilon] : \lambda : \epsilon, 1)$	(8) retrieve operand from stack
$\to$	$(((\lambda.0)0)[\overline{1} : \epsilon], \lfloor 0, 1 \rfloor : \lambda : \epsilon, 1)$	(4) application
$\to$	$((\lambda.0)[\overline{1} : \epsilon], 0[\overline{1} : \epsilon] : \lfloor 0, 1 \rfloor : \lambda : \epsilon, 1)$	(5) β-redex
$\to$	$(0[0[\overline{1} : \epsilon] : \overline{1} : \epsilon], \lfloor 0, 1 \rfloor : \lambda : \epsilon, 1)$	(3) retrieve top binding
$\to$	$(0[\overline{1} : \epsilon], \lfloor 0, 1 \rfloor : \lambda : \epsilon, 1)$	(3) retrieve top binding
$\to$	$(\overline{1}, \lfloor 0, 1 \rfloor : \lambda : \epsilon, 1)$	(7) embed index into ground
$\to$	$(\lfloor 0, 1 \rfloor, \lfloor 0, 1 \rfloor : \lambda : \epsilon, 1)$	(10) accumulate neutral in nf
$\to$	$(\lfloor 0\,0, 1 \rfloor, \lambda : \epsilon, 1)$	(9) out-of-lambda
$\to$	$(\lfloor \lambda.0\,0, 1 \rfloor, \epsilon, 1)$	(11) continuation stack is empty
$\to$	$\lambda.0\,0$	done

Figure 6.7: Execution example of KN on term $\lambda.0((\lambda.0)0)$

$$S \quad ::= \quad \epsilon \mid \Lambda[\rho] : S \mid \lambda : S \mid \lfloor \Lambda, l \rfloor$$

(1)	T	$\to$	$(T[\epsilon], \epsilon, 0)$
(2)	$((n+1)[\mathsf{C} : \rho], S, l)$	$\to$	$(n[\rho], S, l)$
(3)	$(0[\mathsf{C} : \rho], S, l)$	$\to$	(C, S, l)
(4)	$((M\,N)[\rho], S, l)$	$\to$	$(M[\rho], N[\rho] : S, l)$
(5)	$((\lambda.B)[\rho], N[\rho'] : S, l)$	$\to$	$(B[N[\rho'] : \rho], S, l)$
(6)	$((\lambda.B)[\rho], S, l)$	$\to$	$(B[\overline{l+1} : \rho], \lambda : S, l+1)$
(7)	$(\overline{n}, S, l)$	$\to$	$(\lfloor l - n, l \rfloor, S, l)$
(8)	$(\lfloor M, l \rfloor, N[\rho] : S, l')$	$\to$	$(N[\rho], \lfloor M, l \rfloor : S, l)$
(9)	$(\lfloor B, l \rfloor, \lambda : S, l')$	$\to$	$(\lfloor \lambda.B, l \rfloor, S, l')$
(10)	$(\lfloor N, l \rfloor, \lfloor M, l' \rfloor : S, l'')$	$\to$	$(\lfloor M\,N, l' \rfloor, S, l'')$
(11)	$(\lfloor T, l \rfloor, \epsilon, l')$	$\to$	T

Closures C now include de Bruijn indices (n coming from Λ) and de Bruijn levels (written $\overline{n}$) for encoding the nesting of formal parameters that are pushed on the environment. The de Bruijn levels realise what we shall refer to as the *parameters-as-levels* technique. Closures also include an embedding of ground terms with a lambda level $\lfloor \Lambda, l \rfloor$ whose meaning is explained below. The syntax suggests an implicit calculus which we name $\lambda_{\overline{\rho}}$. The continuation stack S can be empty (same symbol as empty environments), store operands, store the control character λ which indicates that the current scope is under an

abstraction, or store embedded ground terms.

The execution example in Figure 6.7 shows the rules of KN at work, where each step is labelled with the rule that is applied next and with a short description of what the rule does. We explain each rule in detail. The init rule (1) constructs a triple for a *closed* term T. Rules (2) and (3) are for looking up variables by peeling off the environment while decrementing indices. The binding at the top of the environment is delivered when the index is 0. Rule (4) pushes on the stack the operand in closure form. Rule (5) embodies a contraction: the operand closure is retrieved from the stack and pushed on the abstraction body's environment. Rule (6) is for unapplied abstractions (there is no closure operand on the top of the stack). The control character λ is pushed on the stack to signal that the machine is going under lambda, and the level l is incremented and also pushed on the abstraction body's environment. The level pushed on the environment $\overline{l+1}$ encodes the nesting of the abstraction's formal parameter. In rule (7), the appropriate de Bruijn index is computed by subtracting $\overline{n}$ from the level in the current scope, and the computed index is embedded in a ground term with the current level. The subtraction is reminiscent of the lexical adjustment technique in σ (Section 6.5) although in KN level l is not reset to zero and no adjustment is needed when looking up in the environment, for it grows as formal parameters are pushed onto it. This guarantees an *index alignment* property, *i.e.*, every index points to a binding on the environment.

The remaining rules are for neutral terms and illuminate the reason for embedded ground terms with levels. We shall explain them with an example. Consider the abstraction $\lambda.0(\lambda.M)N$ which has a neutral term as body. Subterm N has to be reduced with the same level as the head variable 0. The head variable is embedded in a ground term with its level (rule (7), already explained), and the embedding pushed on the stack by rule (8). The machine increments the level when going under lambda in $\lambda.M$ (rule (6), already explained), but it does not decrement the level when scoping out of it in rule (9). However, the appropriate level for N is recovered by rule (10) from the ground term on the top of the stack. Rule (11) ends the execution when the control stack is empty.

6.7 Introducing the calculus of closures $\lambda_{\widetilde{\rho}}$

We introduce the $\lambda_{\widetilde{\rho}}$ calculus as the natural extension of $\lambda_{\widehat{\rho}}$ that subsumes $\lambda_{\overline{\rho}}$:

$$\lambda_{\widetilde{\rho}} \quad \begin{array}{lcl} \mathsf{C} & ::= & \Lambda[\rho] \mid \overline{n} \mid \lfloor n \rfloor \mid \lambda\!\!\lambda.\mathsf{C} \mid \mathsf{C} \cdot \mathsf{C} \\ \rho & ::= & \epsilon \mid \mathsf{C} : \rho \end{array}$$

The calculus only adds two ephemeral constructors which are required for full-reduction, namely, *absolute indices* $\lfloor n \rfloor$ (not to be confused with KN's level-carrying ground terms $\lfloor \Lambda, l \rfloor$) and *closure abstractions* $\lambda\!\!\lambda.\mathsf{C}$. Absolute indices are de Bruijn indices that are not relative to an environment. Absolute indices are different from closures $n[\epsilon]$. The latter stand for free variables (as well as $n[\rho]$ with $n > |\rho|$) and, as we will see below, they trigger index calculations. The reader can deduce from the previous sentence that $\lambda_{\widetilde{\rho}}$ *assumes closures with free variables* (open terms). Absolute indices are required to represent

neutral closures which are closure applications of an absolute index to other closures (for an advance, see the irreducible forms at the bottom of Figure 6.8). Closure abstractions are required to represent closures where the redex may occur under lambda. There is an obvious isomorphism between Λ and all the ephemeral closure constructions (hereafter 'ephemeral closures'), gathered in $\mathsf{E} ::= \lfloor n \rfloor \mid \lambda\!\!\lambda.\mathsf{E} \mid \mathsf{E} \cdot \mathsf{E}$. As was the case with $\lambda_{\widehat{\rho}}$ (Section 6.5), the ephemeral closures of $\lambda_{\widetilde{\rho}}$ are required to define reduction contexts for closures (Section 6.7.1).

The connection between $\lambda_{\widetilde{\rho}}$ and λ is established by substitution σ_{c}:

$$\begin{array}{rcl}
\sigma_{\mathsf{c}}(\mathsf{C}, \mathbb{N}) & \to & \mathsf{E} \\
\sigma_{\mathsf{c}}(n[\rho], l) & = & \begin{cases} \sigma_{\mathsf{c}}(n^{\text{th}}(\rho), l) & \text{if } n < |\rho| \\ \lfloor n - (|\rho| - l) \rfloor & \text{if } n \geq |\rho| \end{cases} \\
\sigma_{\mathsf{c}}((\lambda.B)[\rho], l) & = & \sigma_{\mathsf{c}}(\lambda\!\!\lambda.B[\overline{l+1} : \rho], l) \\
\sigma_{\mathsf{c}}(\lambda\!\!\lambda.\mathsf{B}, l) & = & \lambda\!\!\lambda.\sigma_{\mathsf{c}}(\mathsf{B}, l+1) \\
\sigma_{\mathsf{c}}(\overline{n}, l) & = & \lfloor l - n \rfloor \\
\sigma_{\mathsf{c}}((M\,N)[\rho], l) & = & \sigma_{\mathsf{c}}(M[\rho] \cdot N[\rho], l) \\
\sigma_{\mathsf{c}}(\mathsf{M} \cdot \mathsf{N}, l) & = & \sigma_{\mathsf{c}}(\mathsf{M}, l) \cdot \sigma_{\mathsf{c}}(\mathsf{N}, l) \\
\sigma_{\mathsf{c}}(\lfloor n \rfloor, l) & = & \lfloor n \rfloor
\end{array}$$

Function σ_{c} is the analogous of function σ in $\lambda_{\widehat{\rho}}$ and simulates capture-avoiding substitution in λ, as proven by Lemma 6.7.3 below. Function σ_{c} now carries a lambda level parameter l and enforces index alignment like KN (Section 6.6). Observe that in the 3rd clause, σ_{c} increments the level encoding the nesting of the formal parameter that is pushed on the environment as a de Bruijn level, namely $\overline{l+1}$, but does not increment the lambda level l. It is in the 4th clause, when going under closure abstraction, that the lambda level l is incremented but the environment is not touched. The remaining clauses are unsurprising. Absolute indices are simply returned (1st clause), bound variables are looked up in the environment (2nd clause, case $n < |\rho|$), free variables are given their absolute indices (2nd clause, case $n \geq |\rho|$) which are calculated by subtracting to the current index n the number of *proper bindings* in the environment, *i.e.*, bindings other than de Bruijn levels. (By the invariant on closures that we introduce in Section 6.7.1 the proper bindings in the environment are actually proper closures.) This number coincides with the length of the environment $|\rho|$ minus the current lambda level l. Finally, σ_{c} calculates the absolute index of formal parameters retrieved from the environment (5th clause), lifts application closures to closure applications (6th clause), and distributes over closure applications (7th clause).

Now we show that σ_{c} simulates the capture avoiding substitution function in λ.

Definition 6.7.1 (Well-formed proper closures)**.** *A pair $\langle T[\rho], n\rangle$ is* well-formed*, written $wf(T[\rho], n)$, if the formal parameters in ρ form a descending sequence $\overline{n} : \ldots : \overline{1}$ (an empty sequence if $n = 0$) and every closure interspersed in the sequence is a proper closure $T'[\rho']$ such that $wf(T'[\rho'], m)$ where m is the closest de Bruijn level occurring in ρ from the right, that is, if $m \neq 0$ then $T[\ldots : T'[\rho'] : \ldots : \overline{m} : \ldots]$ and between $T'[\rho']$ and $\overline{m}$ there are only proper closures.*

Trivially, $wf(T[\rho], n)$ implies $n \leq |\rho|$.

Notation 6.7.1.1. *We write μ_m^n for an environment consisting uniquely of formal parameters*

$$[\overline{m+n} : \ldots : \overline{m+1}]$$

where $n, m \geq 0$.

Lemma 6.7.2 (Shifting preserves σ_{C})**.** *For any $l, m \geq 0$ and $T[\rho]$ a proper closure such that $wf(T[\rho], l)$*

$$\sigma_{\mathsf{C}}(T[\rho], l+m) \equiv \uparrow_m^0 (\sigma_{\mathsf{C}}(T[\rho], l))$$

In words, flattening a closure with a level $l+m$ yields the same result than flattening the closure with level l and then shifting it by m with a threshold equals to zero. Notice that the isomorphism $\mathsf{E} \cong \Lambda$ is used extensively.

Proof. In order to facilitate a proof by induction, the lemma has to be generalised by considering how the environment ρ grows with new formal parameters $\overline{n}$ at its left, *i.e.*, when $T \equiv \{\lambda.\}^* B$. The shifting function $\uparrow_m^0$ 'crosses' the lambda symbols in T, and for that some appropriate ranges for the trailing formal parameters at the left of ρ have to be provided. The generalised lemma reads as follows. For any $l_0, l, k_0, k, m \geq 0$ if $wf(T[\rho], l_0)$ and $l \geq l_0$ then

$$\sigma_{\mathsf{C}}(T[\mu_{l+k_0+m}^k : \rho], l+k_0+k+m) \equiv \uparrow_m^{k_0+k} (\sigma_{\mathsf{C}}(T[\mu_{l+k_0}^k : \rho], l+k_0+k))$$

The proof is by structural induction on $T[\rho]$. Thanks to the assumption $wf(T[\rho], l_0)$ the induction hypothesis is only called over the sub-closures of $T[\rho]$ that are of the form that we analyse.

Case $T \equiv n$: We distinguish the sub-cases:

Case $n < k$: Then $n^{\text{th}}(\mu_{l+k_0+m}^k : \rho) = \overline{l+k_0+k+m-n}$ and $n^{\text{th}}(\mu_{l+k_0}^k : \rho) = \overline{l+k_0+k-n}$. We need

$$\sigma_{\mathsf{C}}(\overline{l+k_0+k+m-n}, l+k_0+k+m) \equiv \uparrow_m^{k_0+k} (\sigma_{\mathsf{C}}(\overline{l+k_0+k-n}, l+k_0+k))$$

which simplifies to $n \equiv \uparrow_m^{k_0+k} n$. The latter trivially holds since $n < k_0 + k$.

Case $k \leq n < k + |\rho|$: Then $n^{\text{th}}(\mu_{l+k_0+m}^k : \rho) = n^{\text{th}}(\mu_{l+k_0}^k : \rho) = \mathsf{C}$. We distinguish the sub-cases:

Case $\mathsf{C} \equiv \overline{p}$: We know $p \leq l_0 \leq l$ from the assumption $wf(T[\rho], l_0)$. We need

$$\sigma_{\mathsf{C}}(\overline{p}, l+k_0+k+m) \equiv \uparrow_m^{k_0+k} (\sigma_{\mathsf{C}}(\overline{p}, l+k_0+k))$$

We let $q = l - p \geq 0$. The goal simplifies to

$$q+k_0+k+m \equiv \uparrow_m^{k_0+k} (q+k_0+k)$$

which holds by definition of $\uparrow_m^{k_0+k}$ since $q+k_0+k \geq k_0+k$.

Case $\mathsf{C} \equiv N[\rho']$**:** We need

$$\sigma_{\mathsf{c}}(N[\rho'], l + k_0 + k + m) \equiv \uparrow_m^{k_0+k} (\sigma_{\mathsf{c}}(N[\rho'], l + k_0 + k))$$

Let $k' = k_0 + k$, then the goal is

$$\sigma_{\mathsf{c}}(N[\rho'], l + k' + m) \equiv \uparrow_m^{k'} (\sigma_{\mathsf{c}}(N[\rho'], l + k'))$$

From the assumption $wf(T[\rho], l_0)$ we know $wf(N[\rho'], l')$ and $l \geq l_0 \geq l'$. The goal holds by the induction hypothesis replacing k_0 by k', and letting $k = 0$.

Case $n \geq k + |\rho|$**:** We need

$$n - (|\mu^k_{l+k_0+m} : \rho| - (l + k_0 + k + m)) \equiv \uparrow_m^{k_0+k} (n - (|\mu^k_{l+k_0} : \rho| - (l + k_0 + k)))$$

Let $q = n - (k + |\rho|) \geq 0$. The goal simplifies to

$$q + l + k_0 + k + m \equiv \uparrow_m^{k_0+k} (q + l + k_0 + k)$$

which holds by definition of $\uparrow_m^{k_0+k}$ since $q + l + k_0 + k \geq k_0 + k$.

Case $T \equiv \lambda.B$**:** Both σ_{c} and $\uparrow_m^{k_0+k}$ 'cross' the lambda, and we need

$$\begin{array}{l} \lambda.\sigma_{\mathsf{c}}(B[\mu^{k_0+k+1}_{l+k_0+m} : \rho], l + k_0 + k + 1 + m) \equiv \\ \lambda.\ \uparrow_m^{k_0+k+1} (\sigma_{\mathsf{c}}(B[\mu^{k_0+k+1}_{l+k_0} : \rho], l + k_0 + k + 1)) \end{array}$$

which holds by the induction hypothesis.

Case $T \equiv M\,N$**:** By the induction hypothesis. □

Lemma 6.7.3 (Function σ_{c} simulates $[\_/\_]\_$)**.** *For any $l \geq 0$ and $B[N[\rho] : \rho]$ a proper closure such that $wf(B[N[\rho] : \rho], l)$*

$$\sigma_{\mathsf{c}}(B[N[\rho] : \rho], l) \ \equiv\ [\sigma_{\mathsf{c}}(N[\rho], l)/0](\sigma_{\mathsf{c}}(B[\overline{l+1} : \rho], l + 1))$$

*In words, for any lambda level l, flattening a closure $B[N[\rho] : \rho]$ (*i.e.*, the body of a closure abstraction $\lambda\!\!\lambda.B[\overline{l+1} : \rho]$ where a closure subject $N[\rho]$ is pushed in the position pointed by index 0) yields the same term than the substitution of flattened subject $\sigma_{\mathsf{c}}(N[\rho], l)$ for 0 in flattened body $\sigma_{\mathsf{c}}(B[\overline{l+1} : \rho], l+1)$. Notice that the isomorphism $\mathsf{E} \cong \Lambda$ is used extensively.*

Proof. In order to facilitate a proof by induction, the lemma has to be generalised by considering how the environment $[N[\rho] : \rho]$ grows with new formal parameters $\overline{n}$ at its left, *i.e.*, when $B \equiv \{\lambda.\}^* M$. The capture-avoiding substitution function $[\_/\_]\_$ 'crosses' the lambda symbols in B. The new formal parameters on the left of $[N[\rho] : \rho]$ have to be incremented by one in the right part of the lemma. The generalised lemma reads as follows. For any $l_0, l, m \geq 0$, if $wf(B[N[\rho] : \rho], l_0)$ and $l \geq l_0$ then

$$\sigma_{\mathsf{c}}(B[\mu^m_l : N[\rho] : \rho], l + m) \equiv [\sigma_{\mathsf{c}}(N[\rho], l)/m](\sigma_{\mathsf{c}}(B[\mu^{m+1}_l : \rho], l + m + 1))$$

The proof is by structural induction on $B[\mu_l^m : N[\rho] : \rho]$. Thanks to the assumption $wf(B[N[\rho] : \rho], l_0)$ the induction hypothesis is only called over the sub-closures of $B[N[\rho] : \rho]$ that are of the form that we analyse.

Case $B \equiv n$**:** We distinguish the sub-cases:

Case $n = m$**:** Then $n^{\text{th}}(\mu_l^m : N[\rho] : \rho) = N[\rho]$ and by definition of $[\_/\_]\_$ we need $\sigma_{\mathsf{c}}(N[\rho], l + m) \equiv \uparrow_m^0 \sigma_{\mathsf{c}}(N[\rho], l)$ which holds by Lemma 6.7.2 where $k_0, k = 0$.

Case $n < m$**:** Then $n^{\text{th}}(\mu_l^m : N[\rho] : \rho) = \overline{l + (m - n)}$ and $n^{\text{th}}(\mu_{l+1}^{m+1} : \overline{l+1} : \rho) = \overline{l + (m - n) + 1}$. We need

$$\sigma_{\mathsf{c}}(\overline{l + (m - n)}, l + m) \equiv [\sigma_{\mathsf{c}}(N[\rho], l)/m](\sigma_{\mathsf{c}}(\overline{l + (m - n) + 1}, l + m + 1))$$

which simplifies to $n \equiv [\sigma_{\mathsf{c}}(N[\rho], l)/m]n$. The lemma holds by definition of $[\_/\_]\_$ because $n < m$.

Case $n > m$**:** Then $n^{\text{th}}(\mu_l^m : N[\rho] : \rho) = n^{\text{th}}(\mu_l^{m+1} : \rho) = \mathsf{C}$, where C is either a formal parameter $\overline{p}$ or a proper closure $T[\rho']$. We distinguish the sub-cases:

Case $\mathsf{C} \equiv \overline{p}$**:** From the assumption $wf(B[N[\rho] : \rho], l_0)$ we know that $p \leq l_0 \leq l$. We need

$$\sigma_{\mathsf{c}}(\overline{p}, l + m) \equiv [\sigma_{\mathsf{c}}(N[\rho], l)/m](\sigma_{\mathsf{c}}(\overline{p}, l + m + 1))$$

which simplifies to $q \equiv [\sigma_{\mathsf{c}}(N[\rho], l)/m](q + 1)$, where $q \geq m$. The lemma holds because $[\_/\_]\_$ decrements by one every index which is greater than m.

Case $\mathsf{C} \equiv T[\rho']$**:** We need

$$\sigma_{\mathsf{c}}(T[\rho'], l + m) \equiv [\sigma_{\mathsf{c}}(N[\rho], l)/m](\sigma_{\mathsf{c}}(T[\rho'], l + m + 1))$$

where $wf(T[\rho'], l')$. The lemma holds by the induction hypothesis because $l \geq l_0 \geq l'$.

Case $B \equiv \lambda.M$**:** Both σ_{c} and $[\_/\_]\_$ 'cross' the lambda, and we need

$$\lambda.\sigma_{\mathsf{c}}(M[\mu_l^{m+1} : N[\rho] : \rho], l+m+1) \equiv \lambda.[\sigma_{\mathsf{c}}(N[\rho], l)/m+1](\sigma_{\mathsf{c}}(M[\mu_l^{m+2} : \rho], l+m+2))$$

which holds by the induction hypothesis.

Case $B \equiv M\,N$**:** By the induction hypothesis. □

6.7.1 Structural operational semantics of normal order in $\lambda_{\widetilde{\rho}}$

Figure 6.8 shows the structural operational semantics of normal order in $\lambda_{\widetilde{\rho}}$. It is a straightforward adaptation of the structural operational semantics of normal order in λ (Figure 6.3)

$$\frac{n < |\rho|}{\langle n[\rho], l\rangle \to_{\widetilde{no}} \langle n^{\text{th}}(\rho), l\rangle}\ (\textsc{Var}_{\widetilde{\rho}})$$

$$\frac{}{\langle (M\ N)[\rho], l\rangle \to_{\widetilde{no}} \langle M[\rho] \cdot N[\rho], l\rangle}\ (\textsc{App}_{\widetilde{\rho}})$$

$$\frac{n \geq |\rho|}{\langle n[\rho], l\rangle \to_{\widetilde{no}} \langle \lfloor n - (|\rho| - l)\rfloor, l\rangle}\ (\textsc{Fre}_{\widetilde{\rho}})$$

$$\frac{}{\langle (\lambda.B)[\rho], l\rangle \to_{\widetilde{no}} \langle \lambda\!\!\lambda.B[\overline{l+1} : \rho], l\rangle}\ (\textsc{Lam}_{\widetilde{\rho}})$$

$$\frac{}{\langle \overline{n}, l\rangle \to_{\widetilde{no}} \langle \lfloor l - n\rfloor, l\rangle}\ (\textsc{Par}_{\widetilde{\rho}})$$

$$\frac{}{\langle (\lambda\!\!\lambda.B[\overline{l+1} : \rho]) \cdot N[\rho], l\rangle \to_{\widetilde{no}} \langle B[N[\rho] : \rho], l\rangle}\ (\beta_{\widetilde{\rho}})$$

$$\frac{\mathsf{M} \notin \mathsf{WHNF_C} \quad \langle \mathsf{M}, l\rangle \to_{\widetilde{no}} \langle \mathsf{M}', l\rangle}{\langle \mathsf{M} \cdot \mathsf{N}, l\rangle \to_{\widetilde{no}} \langle \mathsf{M}' \cdot \mathsf{N}, l\rangle}\ (\mu 1_{\widetilde{\rho}})$$

$$\frac{\mathsf{M} \in \mathsf{WHNF_C} \quad \mathsf{M} \not\equiv \lambda\!\!\lambda.\mathsf{B} \quad \langle \mathsf{M}, l\rangle \to_{\widetilde{no}} \langle \mathsf{M}', l\rangle}{\langle \mathsf{M} \cdot \mathsf{N}, l\rangle \to_{\widetilde{no}} \langle \mathsf{M}' \cdot \mathsf{N}, l\rangle}\ (\mu 2_{\widetilde{\rho}})$$

$$\frac{\mathsf{M} \in \mathsf{NF_C} \quad \mathsf{M} \not\equiv \lambda\!\!\lambda.\mathsf{B} \quad \langle \mathsf{N}, l\rangle \to_{\widetilde{no}} \langle \mathsf{N}', l\rangle}{\langle \mathsf{M} \cdot \mathsf{N}, l\rangle \to_{\widetilde{no}} \langle \mathsf{M} \cdot \mathsf{N}', l\rangle}\ (\nu_{\widetilde{\rho}})$$

$$\frac{\langle \mathsf{B}, l+1\rangle \to_{\widetilde{no}} \langle \mathsf{B}', l+1\rangle}{\langle \lambda\!\!\lambda.\mathsf{B}, l\rangle \to_{\widetilde{no}} \langle \lambda\!\!\lambda.\mathsf{B}', l\rangle}\ (\xi_{\widetilde{\rho}})$$

$$\begin{array}{rcl} \mathsf{WHNF_C} & ::= & \lambda\!\!\lambda.\Lambda[\rho] \mid \lfloor n\rfloor\{\cdot\,\mathsf{C}\}^* \\ \mathsf{NF_C} & ::= & \lambda\!\!\lambda.\mathsf{NF_C} \mid \mathsf{NNF_C} \\ \mathsf{NNF_C} & ::= & \lfloor n\rfloor\{\cdot\,\mathsf{NF_C}\}^* \end{array}$$

Figure 6.8: Parameters-as-levels and closure-converted structural operational semantics of normal order in $\lambda_{\widetilde{\rho}}$.

to which we have added rules that consider ephemerals and KN's parameters-as-levels (Section 6.6). The lambda level l has to be carried along and thus $\to_{\widetilde{no}}$ operates on pairs $\langle \mathsf{C}, \mathbb{N}\rangle$ rather than just closures. The rules on the top of Figure 6.8 are notions of reduction for the new constructors and come naturally from σ_{C}. $\textsc{Var}_{\widetilde{\rho}}$ is the rule for bound variables,

$\textsc{App}_{\widetilde{\rho}}$ for lifting to closure application, $\textsc{Fre}_{\widetilde{\rho}}$ for free variables, $\textsc{Lam}_{\widetilde{\rho}}$ for lifting to closure abstraction where the formal parameter (the incremented lambda level) is pushed on the environment, and $\textsc{Par}_{\widetilde{\rho}}$ for formal parameters. The first rule on the bottom ($\beta_{\widetilde{\rho}}$) contracts $\beta_{\widetilde{\rho}}$-redices $(\lambda\!\!\lambda.B[\overline{l+1} : \rho]) \cdot N[\rho]$, where the formal parameter $\overline{l+1}$ that was pushed on the top of the environment by an immediately preceding ephemeral expansion $\textsc{Lam}_{\widetilde{\rho}}$ is discarded and replaced by the operand $N[\rho]$. The other compatibility rules ($\mu 1_{\widetilde{\rho}}$), ($\mu 2_{\widetilde{\rho}}$), ($\nu_{\widetilde{\rho}}$), and ($\xi_{\widetilde{\rho}}$) are obtained by adapting to closure-level pairs the corresponding rules in Figure 6.3. A pair's lambda level is incremented in ($\xi_{\widetilde{\rho}}$), for it 'goes under closure abstraction', leaving B's environment untouched.

The dependency on call-by-name can be observed by looking at rules $\textsc{Var}_{\widetilde{\rho}}$, $\textsc{App}_{\widetilde{\rho}}$, $\textsc{Fre}_{\widetilde{\rho}}$, $\textsc{Lam}_{\widetilde{\rho}}$, ($\beta_{\widetilde{\rho}}$) and ($\mu 1_{\widetilde{\rho}}$) which taken in isolation define call-by-name in $\lambda_{\widetilde{\rho}}$. If we compare with call-by-name in $\lambda_{\widehat{\rho}}$ (Section 6.5.1, Figure 6.5) we find the addition of $\textsc{Fre}_{\widetilde{\rho}}$ and $\textsc{Lam}_{\widetilde{\rho}}$, and the omission in ($\mu_{\widehat{\rho}}$) of the premiss $\mathsf{M} \notin \mathsf{WHNF_C}$ which is present in ($\mu 1_{\widetilde{\rho}}$). These minor differences are easily justified: $\textsc{Fre}_{\widetilde{\rho}}$ is required for free variables, and $\textsc{Lam}_{\widetilde{\rho}}$ is the immediately preceding ephemeral expansion required for ($\beta_{\widetilde{\rho}}$). Finally, the premiss $\mathsf{M} \notin \mathsf{WHNF_C}$ is not present in ($\mu_{\widehat{\rho}}$) because that rule applies only when M is not in whnf. The premiss is required in ($\mu 1_{\widetilde{\rho}}$) to control the rule's applicability as part of a larger set of rules that specify normal order.

Observe that derivations are *balanced*, *i.e.*, a pair's lambda level remains constant in the left- and right-hand sides of judgements in derivation trees. This makes reasoning by structural induction easier. Since the lambda level of a binding $N[\rho]$ depends on its position in the environment (see Lemma 6.7.4 in this section) the lambda levels need not be carried along with closures in environments and, unlike KN, levels need not be recovered from the environment when reducing operands of neutral closures. This suggest an optimisation of KN that we discuss further in Section 6.10.4.

The syntax for closure whnfs (hereafter whnf_{C}) and closure nfs (hereafter nf_{C}) is shown at the bottom of Figure 6.8. A nf_{C} consists of a closure abstraction with a body in nf_{C}, or of a neutral in normal form (hereafter a nnf_{C}). The nf_{C}s are included in ephemeral closures E but are not included in whnf_{C}s because abstraction bodies in whnf_{C}s are proper closures with delayed substitutions in their environments. These environments may be enlarged by the combination of $\textsc{Lam}_{\widetilde{\rho}}$ and ($\beta_{\widetilde{\rho}}$), and their closures can only be removed when demanded by $\textsc{Var}_{\widetilde{\rho}}$.

Lemma 6.7.4 (Invariant on closures)**.** *(i) $wf(T[\rho], l)$ holds for every pair $\langle T[\rho], l\rangle$ in any derivation of the inference system in Figure 6.8.*

(ii) Every $\beta_{\widetilde{\rho}}$-redex in a $\rightarrow_{\widetilde{no}}$ reduction sequence is of the form $(\lambda\!\!\lambda.B[\overline{l+1} : \rho]) \cdot N[\rho]$ (same ρ in body and operand) and both $wf(B[\overline{l+1} : \rho], l+1)$ and $wf(N[\rho], l)$ hold with l the current lambda level.

Proof. (i) By considering rules ($\beta_{\widetilde{\rho}}$), $\textsc{Lam}_{\widetilde{\rho}}$, and ($\xi_{\widetilde{\rho}}$) in Figure 6.8. The rest of the rules do not modify the environment nor the level. Rule ($\beta_{\widetilde{\rho}}$) trivially preserves the invariant. The combination of $\textsc{Lam}_{\widetilde{\rho}}$ and ($\xi_{\widetilde{\rho}}$) also preserves the invariant. The

environment in the body is enlarged by $\textsc{Lam}_{\widetilde{\rho}}$ and the level in the pair is incremented by $(\xi_{\widetilde{\rho}})$.

(ii) By considering all the rules in Figure 6.8. Reduction always starts at a proper closure, *i.e.*, a term injected into a closure by adding an empty environment. By the compatibility rules, no operand will be expanded or reduced before the operator in an application, and the operators are reduced only to closure abstractions (the strategy is leftmost-outermost). Rule $\textsc{App}_{\widetilde{\rho}}$ duplicates the environment at both sides of a closure application, and rule $\textsc{Lam}_{\widetilde{\rho}}$ enlarges the environment in the body of the operator when the operator is an abstraction. □

6.7.2 Stepwise connection between $\to_{no}$ and $\to_{\widetilde{no}}$

In Section 6.7 we said that substitution function σ_{C} connects $\lambda_{\widetilde{\rho}}$ with λ. Moreover, the stepwise connection can be established between $\to_{\widetilde{no}}$ in $\lambda_{\widetilde{\rho}}$ and $\to_{no}$ in λ, as we prove next. We refer to the union of the rules in the upper part of Figure 6.8 as $(\widetilde{\rho})$, which consists of all the notions of reduction of $\to_{\widetilde{no}}$ but $(\beta_{\widetilde{\rho}})$. We partition the reduction relation $\to_{\widetilde{no}}$ into ephemeral expansion and single-step $\beta_{\widetilde{\rho}}$-reduction:

Definition 6.7.5 (Epehemeral expansion)**.** *The* ephemeral expansion $\to_{\widetilde{\rho}}$ *is the union of the compatibility rules in Figure 6.8 and* $(\widetilde{\rho})$*,* i.e.*, all the rules in the figure but* $(\beta_{\widetilde{\rho}})$*. Iterating ephemeral expansion yields a* nf_{C} *if the input does not reduce to a closure containing any* $\beta_{\widetilde{\rho}}$*-redex, or an* expanded closure*, which unravels its leftmost-outermost* $\beta_{\widetilde{\rho}}$*-redex, otherwise. The expanded closures* X *(do not mistake them with the ephemeral closures* E*) are defined by the EBNF grammar* $\mathsf{X} ::= (\lambda\!\!\lambda.\Lambda[\overline{n} : \rho]) \cdot \mathsf{C} \mid \lambda\!\!\lambda.\mathsf{X} \mid \mathsf{NNF}_{\mathsf{C}} \cdot \mathsf{X}\{\cdot \mathsf{C}\}^*$.

Definition 6.7.6 (Single-step $\beta_{\widetilde{\rho}}$-reduction)**.** *The* single-step $\beta_{\widetilde{\rho}}$-reduction $\to_{\beta_{\widetilde{\rho}}}$ *is the union of the compatibility rules in Figure 6.8 and* $(\beta_{\widetilde{\rho}})$*,* i.e.*, all the rules in the bottom part of the figure. Single-step* $\beta_{\widetilde{\rho}}$*-reduction contracts the leftmost-outermost* $\beta_{\widetilde{\rho}}$*-redex (of the form* $(\lambda\!\!\lambda.B[\overline{l+1} : \rho]) \cdot N[\rho]$*) of an expanded closure.*

Relations $\to_{\widetilde{\rho}}$ and $\to_{\beta_{\widetilde{\rho}}}$ share the same compatibility rules. Since the notions of reduction of $\to_{\widetilde{no}}$ are partitioned into $(\widetilde{\rho})$ and $(\beta_{\widetilde{\rho}})$, it is clear that $\to_{\widetilde{no}} = \to_{\widetilde{\rho}} \bigcup \to_{\beta_{\widetilde{\rho}}}$.

We turn the σ_{C} function into a substitution relation:

Definition 6.7.7 (Substitution relation)**.** *The* substitution relation $\to_{\sigma}$ *is the compatible closure of* $(\widetilde{\rho})$ *minus the rules dealing with bindings in environments.*

$$\frac{\langle \mathsf{M}, l\rangle \to_{\sigma} \langle \mathsf{M}', l\rangle}{\langle \mathsf{M} \cdot \mathsf{N}, l\rangle \to_{\sigma} \langle \mathsf{M}' \cdot \mathsf{N}, l\rangle}\ (\mu_{\sigma}) \qquad \frac{\langle \mathsf{N}, l\rangle \to_{\sigma} \langle \mathsf{N}', l\rangle}{\langle \mathsf{M} \cdot \mathsf{N}, l\rangle \to_{\sigma} \langle \mathsf{M} \cdot \mathsf{N}', l\rangle}\ (\nu_{\sigma})$$

$$\frac{\langle \mathsf{B}, l+1\rangle \to_{\sigma} \langle \mathsf{B}', l+1\rangle}{\langle \lambda\!\!\lambda.\mathsf{B}, l\rangle \to_{\sigma} \langle \lambda\!\!\lambda.\mathsf{B}', l\rangle}\ (\xi_{\sigma})$$

The compatibility rules above are the same as those of $\to_{\widetilde{no}}$ *(Figure 6.8) but unrestricted and simplified by removing the side conditions. Clearly,* $\langle \mathsf{M}, l\rangle \to_{\widetilde{\rho}} \langle \mathsf{M}', l\rangle$ *implies* $\langle \mathsf{M}, l\rangle \to_{\sigma}$

$\langle \mathsf{M}', l\rangle$, *but not the opposite. Iterating* $\rightarrow_\sigma$ *yields an ephemeral closure where all the delayed substitutions have been flattened. This ephemeral closure coincides with the result of applying* σ_{C} *to the input closure,* i.e., $\langle \mathsf{C}, l\rangle \rightarrow_\sigma^* \langle \mathsf{E}, l\rangle$ *iff* $\sigma_{\mathsf{C}}(\mathsf{C}, l) = \mathsf{E}$.

Definition 6.7.8 (Height of a closure)**.** *The* height of a closure *is calculated by function* h *below.*

$$\begin{array}{rcl} h(\mathsf{C}) & \rightarrow & \mathbb{N} \\ h(n[\rho]) & = & \begin{cases} 1 + h(n^{\text{th}}(\rho)) & \text{if } n < |\rho| \\ 0 & \text{if } n \geq |\rho| \end{cases} \\ h((\lambda.B)[\rho]) & = & 1 + h(\lambda\!\!\lambda.B[\overline{n} : \rho]) \\ h((M\ N)[\rho]) & = & 1 + h(M[\rho] \cdot N[\rho]) \\ h(\lambda\!\!\lambda.\mathsf{B}) & = & 1 + h(\mathsf{B}) \\ h(\overline{n}) & = & 0 \\ h(\mathsf{M} \cdot \mathsf{N}) & = & 1 + \max\{h(\mathsf{M}), h(\mathsf{N})\} \\ h(\lfloor n \rfloor) & = & 0 \end{array}$$

The first three clauses calculate the height of a proper closure $T[\rho]$*. In the first clause* T *is a de Bruijn index* n*. If* n *has a binding in the environment (*$n < |\rho|$*) the height is incremented by one and the function resumes over the binding retrieved from* ρ*. If* n *codifies a free variable (*$n \geq |\rho|$*) the height is zero. The second and third clauses just increment the height by one and resume over the expanded closure. In the fourth clause, the height is incremented by one when 'crossing' a closure abstraction and the function resumes over the body. In the sixth clause, the height is incremented by one and the function resumes over the highest branch. The height is zero for both de Bruijn levels codifying a formal parameter (fifth clause) and for ground indices (seventh clause).*

Theorem 6.7.9 (Normal order commutes with substitution)**.** *Let* $l \geq 0$*,* M_n *be closures,* X_n *be expanded closures,* E_n *be ephemeral closures, and* M_n *be terms. The following diagram commutes:*

$$\begin{array}{ccc} \langle \mathsf{M}_i, l\rangle & \xrightarrow[\widetilde{\rho}]{}^{*} \langle \mathsf{X}_i, l\rangle & \underset{\beta_{\widetilde{\rho}}}{\longrightarrow} \langle \mathsf{M}_{i+1}, l\rangle \\ \sigma \downarrow_* & & \sigma \downarrow_* \\ \langle \mathsf{E}_i, l\rangle & & \langle \mathsf{E}_{i+1}, l\rangle \\ \cong \updownarrow & & \cong \updownarrow \\ M_i & \underset{no}{\longrightarrow} & M_{i+1} \end{array}$$

Proof. By induction on the height of M_i. The closure M_i has an expanded closure X_i which contains a $\beta_{\widetilde{\rho}}$-redex. Let us first analyse the base cases, which are the $\beta_{\widetilde{\rho}}$-redices where the body is a proper closure with an index term n:

Case $\mathsf{M}_i \equiv (\lambda\!\!\lambda.n[\overline{l+1} : \rho]) \cdot N[\rho]$**:** Then $M_i \equiv (\lambda.\sigma_{\mathsf{C}}(n[\overline{l+1}], l+1))(\sigma_{\mathsf{C}}(N[\rho], l))$, $\mathsf{X}_i \equiv \mathsf{M}_i$, and $\mathsf{M}_{i+1} \equiv n[N[\rho] : \rho]$. The commuting condition

$$\sigma_{\mathsf{C}}(n[N[\rho] : \rho], l) \equiv [\sigma_{\mathsf{C}}(N[\rho], l)/0](\sigma_{\mathsf{C}}(n[\overline{l+1} : \rho], l+1))$$

holds by Lemma 6.7.3 and by the invariant on closures. If $n \neq 0$, then M_i has height $h(N[\rho])$. Otherwise, M_i has height $1 + h(N[\rho])$.

By the invariant on closures, the operand of the redex is always a proper closure $N[\rho]$ so we need not consider whether the operand is an arbitrary closure. Now we analyse the general cases:

Case $\mathsf{M}_i \equiv (\lambda\!\!\lambda.B[\overline{l+1} : \rho]) \cdot N[\rho]$**:** Then $M_i \equiv (\lambda.\sigma_{\mathsf{C}}(B[\overline{l+1}], 1))(\sigma_{\mathsf{C}}(N[\rho], l))$, $\mathsf{X}_i \equiv \mathsf{M}_i$, and $\mathsf{M}_{i+1} \equiv B[N[\rho] : \rho]$. The commuting condition

$$\sigma_{\mathsf{C}}(B[N[\rho] : \rho], l) \equiv [\sigma_{\mathsf{C}}(N[\rho], l)/0](\sigma_{\mathsf{C}}(B[\overline{l+1} : \rho], l+1))$$

holds by Lemma 6.7.3 and by the invariant on closures. (We need not consider whether the operand is an arbitrary closure.)

Case $\mathsf{M}_i \equiv T[\rho]$**:** Then $T[\rho]$ expands in one step to some M'_i with $h(\mathsf{M}'_i) < h(T[\rho])$. Since $\langle T[\rho], l\rangle \to_{\widetilde{\rho}} \langle \mathsf{M}'_i, l\rangle$ implies $\langle T[\rho], l\rangle \to_{\sigma} \langle \mathsf{M}'_i, l\rangle$, then $\sigma_{\mathsf{C}}(T[\rho], l) \equiv \sigma_{\mathsf{C}}(\mathsf{M}'_i, l) \cong M_i$, and the theorem holds by the induction hypothesis.

Case $\mathsf{M}_i \equiv \lambda\!\!\lambda.\mathsf{B}$**:** By the induction hypothesis, the theorem holds for B, in particular at level $l + 1$. Hence, the theorem holds for $\lambda\!\!\lambda.\mathsf{B}$ at level l.

Case $\mathsf{M}_i \equiv \mathsf{M} \cdot \mathsf{N}$**,** $\mathsf{M} \not\equiv \lambda\!\!\lambda.B[\overline{n} : \rho]$**:** If M has an expanded closure, then the theorem holds for M by the induction hypothesis. If M expands to a nf_{C}, then N has to have an expanded closure, and similarly, the theorem holds for N by the induction hypothesis. In either cases, the theorem holds for $\mathsf{M} \cdot \mathsf{N}$. □

6.8 From structural operational semantics to reduction-free normaliser

6.8.1 From structural to reduction semantics

The search functions `search_whnf` and `search_nf` in the code implement the compatibility rules of the structural operational semantics of normal order in $\lambda_{\widetilde{\rho}}$ (Figure 6.8). The search functions deliver for an input term the (normal order or call-by-name) redex subterm to be contracted or the input term back if the input term is irreducible. The entry function `search` invokes `search_nf`. (From now on we omit for brevity the entry functions of all our semantics.) More precisely, function `search_nf` searches for a nf_{C} or for the next redex to be contracted. It relies on `search_whnf` to check if operators in applications are in $\mathrm{whnf}_{\mathsf{C}}$. Function `search_whnf` searches for a $\mathrm{whnf}_{\mathsf{C}}$ or for the next redex in the call-by-name subreduction to be contracted.

The use of two functions explicitly reflects the inclusion of call-by-name within normal order. An alternative equivalent implementation using a single search function with a boolean check for $\mathrm{whnf}_{\mathsf{C}}$-ness would reflect it implicitly. The derivation tree above a second

premiss of $(\mu 1_{\widetilde{\rho}})$ will only contain call-by-name rules because $(\mu 2_{\widetilde{\rho}})$, $(\nu_{\widetilde{\rho}})$ and $(\xi_{\widetilde{\rho}})$ are only applicable when the operator is in $\mathrm{whnf}_{\mathsf{C}}$ or in nf_{C}.

We apply standard derivation steps (CPS transformation, simplification, defunctionalisation, decomposition) and obtain decomposition functions `decompose_whnf`, `decompose_nf`, and `decompose_cont` (the latter the continuation dispatcher that inevitably pops up). By adding the necessary `contract`, `recompose`, and trampoline `iterate` functions (Danvy, 2005) we obtain the trampolined reduction-based normaliser `normalise` that implements the following reduction semantics. The reduction-based normaliser is the starting point of the derivation path shown in Figure 6.1.

$$
\begin{array}{lll}
\text{Red. context:} & \mathbf{C}^l_{\widetilde{no}}[\,] & ::= \quad [\,]^l \mid \mathbf{C}^l_{\widetilde{bn}}[\,]\cdot\mathsf{C} \mid \lambda\!\!\lambda.\mathbf{C}^{l+1}_{\widetilde{no}}[\,] \mid \mathbf{C}^l_{\widetilde{ne}}[\,] \\
& \mathbf{C}^l_{\widetilde{bn}}[\,] & ::= \quad [\,]^l \mid \mathbf{C}^l_{\widetilde{bn}}[\,]\cdot\mathsf{C} \\
& \mathbf{C}^l_{\widetilde{ne}}[\,] & ::= \quad \mathsf{NNF}_{\mathsf{C}}\cdot\mathbf{C}^l_{\widetilde{no}}[\,] \mid \mathbf{C}^l_{\widetilde{ne}}[\,]\cdot\mathsf{C}
\end{array}
$$

$$
\begin{array}{ll}
\text{Contraction:} & \langle\mathbf{C}^0_{\widetilde{no}}[n[\rho]],l\rangle \;\rightarrow_{\widetilde{no}} \begin{cases} \langle\mathbf{C}^0_{\widetilde{no}}[n^{\mathrm{th}}(\rho)],l\rangle & \text{if } n<|\rho| \\ \langle\mathbf{C}^0_{\widetilde{no}}[\lfloor n-(|\rho|-l)\rfloor],l\rangle & \text{if } n\geq|\rho| \end{cases} \\
& \langle\mathbf{C}^0_{\widetilde{no}}[(M\,N)[\rho]],l\rangle \;\rightarrow_{\widetilde{no}}\; \langle\mathbf{C}^0_{\widetilde{no}}[M[\rho]\cdot N[\rho]],l\rangle \\
& \langle\mathbf{C}^0_{\widetilde{no}}[\overline{n}],l\rangle \;\rightarrow_{\widetilde{no}}\; \langle\mathbf{C}^0_{\widetilde{no}}[\lfloor l-n\rfloor],l\rangle \\
& \langle\mathbf{C}^0_{\widetilde{no}}[(\lambda.B)[\rho]],l\rangle \;\rightarrow_{\widetilde{no}}\; \langle\mathbf{C}^0_{\widetilde{no}}[\lambda\!\!\lambda.B[\overline{l+1}:\rho]],l\rangle \\
& \langle\mathbf{C}^0_{\widetilde{no}}[(\lambda\!\!\lambda.B[\overline{n}:\rho])\cdot\mathsf{N}],l\rangle \;\rightarrow_{\widetilde{no}}\; \langle\mathbf{C}^0_{\widetilde{no}}[B[\mathsf{N}:\rho]],l\rangle
\end{array}
$$

The reduction relation $\rightarrow_{\widetilde{no}}$ is defined on pairs $\langle\mathbf{C}^0_{\widetilde{no}}[\mathsf{R}],l\rangle$ consisting of a top-level context (with the closure redex R within the hole) and the lambda level l at which the redex occurs. Reduction contexts keep track of the lambda level (superscripts), starting with level zero (top scope) and incrementing it when entering a $\lambda\!\!\lambda$ scope. Thus, the lambda level l in $\langle\mathbf{C}^0_{\widetilde{no}}[\mathsf{R}],l\rangle$ is such that $\mathbf{C}^0_{\widetilde{no}}[\mathsf{R}]\equiv\ldots[\mathsf{R}]^l\ldots$

If we compare the above reduction semantics with the call-by-name reduction semantics of $\lambda_{\widehat{\rho}}$ (Section 6.5.1) we find that the differences are easily dispelled by considering $\mathbf{C}^0_{\widetilde{bn}}[\,]$ to be the top-level context, by removing the contraction cases for free variables and for formal parameters, and by shortcutting closure abstractions. Free variables are not considered in $\lambda_{\widehat{\rho}}$ which assumes closures without free variables. Formal parameters $\overline{n}$ are not considered in $\lambda_{\widehat{\rho}}$ because call-by-name does not go under lambda. Finally, the last contraction case for call-by-name in Figure 6.5 can be obtained by shortcutting the last two contraction cases of the reduction semantics. The lambda level is never incremented by $\mathbf{C}^0_{\widetilde{bn}}[\,]$ and can be dropped.

Theorem 6.8.1 (Unique decomposition)**.** *A closure* C *is either a* nf_{C} *or there exists a unique context* $\mathbf{C}^0_{\widetilde{no}}[\,]$ *and redex* R *such that* $\mathsf{C}\equiv\mathbf{C}^0_{\widetilde{no}}[\mathsf{R}]$.

Proof. By structural induction on C. Let l be the current lambda level, which starts at zero.

Case $\mathsf{C}\equiv\lfloor n\rfloor$**:** C is in nf_{C}.

Case $\mathsf{C} \equiv M[\rho]$**:** C is a redex and the unique context for C is the hole $[\,]^l$.

Case $\mathsf{C} \equiv \overline{n}$**:** C is a redex and the unique context for C is the hole $[\,]^l$.

Case $\mathsf{C} \equiv \lambda\!\!\lambda.\mathsf{B}$**:** if B is in nf_{C} then C is in nf_{C}. Otherwise, by the ind. hyp. we have $\mathsf{B} \equiv \mathbf{C}^l_{\widetilde{no}}[\mathsf{R}]$ and therefore $\mathsf{C} \equiv \lambda\!\!\lambda.\mathbf{C}^{l+1}_{\widetilde{no}}[\mathsf{R}]$. The unique context for C is $\lambda\!\!\lambda.\mathbf{C}^{l+1}_{\widetilde{no}}[\,]$ derivable from the axiom as follows: $\mathbf{C}^l_{\widetilde{no}}[\,] \Rightarrow \lambda\!\!\lambda.\mathbf{C}^{l+1}_{\widetilde{no}}[\,]$.

Case $\mathsf{C} \equiv \mathsf{M} \cdot \mathsf{N}$ **with** $\mathsf{M} \equiv \lambda\!\!\lambda.\mathsf{B}$**:** whether B is in nf_{C} or not, C is a redex and the unique context for C is $[\,]^l$ derivable from the axiom as follows: $\mathbf{C}^l_{\widetilde{no}}[\,] \Rightarrow [\,]^l$.

Case $\mathsf{C} \equiv \mathsf{M} \cdot \mathsf{N}$ **with** $\mathsf{M} \not\equiv \lambda\!\!\lambda.\mathsf{B}$**:** there are two sub-cases:

Case $\mathsf{M} \in \mathsf{NNF}_{\mathsf{C}}$**:** if N is in nf_{C} then C is in nf_{C}. Otherwise, by the ind. hyp. we have $\mathsf{N} \equiv \mathbf{C}^l_{\widetilde{no}}[R]$ and therefore $\mathsf{C} \equiv \mathsf{M} \cdot \mathbf{C}^l_{\widetilde{no}}[R]$. The unique context for C is $\mathsf{M} \cdot \mathbf{C}^l_{\widetilde{no}}[\,]$ derivable from the axiom as follows: $\mathbf{C}^l_{\widetilde{no}}[\,] \Rightarrow \mathbf{C}^l_{\widetilde{ne}}[\,] \Rightarrow \mathsf{NNF}_{\mathsf{C}} \cdot \mathbf{C}^l_{\widetilde{no}}[\,] \Rightarrow \mathsf{M} \cdot \mathbf{C}^l_{\widetilde{no}}[\,]$.

Case $\mathsf{M} \notin \mathsf{NNF}_{\mathsf{C}}$**:** M is not in nf_{C} or otherwise it would be a closure abstraction, and we are assuming $\mathsf{M} \not\equiv \lambda\!\!\lambda.\mathsf{B}$. By the ind. hyp. we have $\mathsf{M} \equiv \mathbf{C}^l_{\widetilde{no}}[\mathsf{R}]$ and therefore $\mathsf{C} \equiv \mathbf{C}^l_{\widetilde{no}}[\mathsf{R}] \cdot \mathsf{N}$. The only non-terminals leading to a redex in operator position are $\mathbf{C}^l_{\widetilde{bn}}[\,]$ and $\mathbf{C}^l_{\widetilde{ne}}[\,]$. These are disjoint cases: in $\mathbf{C}^l_{\widetilde{bn}}[\,]$ the redex is located in the leftmost operator of a multiple closure application whereas in $\mathbf{C}^l_{\widetilde{ne}}[\,]$ the redex is located at the right of a $\mathsf{NNF}_{\mathsf{C}}$ in the leftmost neutral closure operator of a multiple closure application. In the first case the unique context for C is $\mathbf{C}^l_{\widetilde{bn}}[\,] \cdot \mathsf{N}$ derived from the axiom as follows: $\mathbf{C}^l_{\widetilde{no}}[\,] \Rightarrow \mathbf{C}^l_{\widetilde{bn}}[\,] \cdot \mathsf{C} \Rightarrow \mathbf{C}^l_{\widetilde{bn}}[\,] \cdot \mathsf{N}$. In the second case the unique context for C is $\mathbf{C}^l_{\widetilde{ne}}[\,] \cdot \mathsf{N}$ derived as follows: $\mathbf{C}^l_{\widetilde{no}}[\,] \Rightarrow \mathbf{C}^l_{\widetilde{ne}}[\,] \Rightarrow \mathbf{C}^l_{\widetilde{ne}}[\,] \cdot \mathsf{C} \Rightarrow \mathbf{C}^l_{\widetilde{ne}}[\,] \cdot \mathsf{N}$. □

6.8.2 Syntactic correspondence

We transform the reduction-based normaliser that implements the reduction semantics of Section 6.8.1 to the environment-based eval/apply abstract machine shown below using the following steps: refocusing, *refined* inlining-of-iterate-function, and transition compression. The refined inlining-of-iterate-function step (García-Pérez & Nogueira, 2013, 2014a) exploits the shape invariant of the continuation stack to obtain a machine with a *shallow inspection property*, *i.e.*, a machine whose dispatcher does not inspect the current continuation argument and therefore can be refunctionalised. Refunctionalisation and defunctionalisation require shallow inspection of the continuation stack (Ager *et al.*, 2003b). The machine in Figure 6.9 is the closure-converted version of the substitution-based eval/apply machine in (García-Pérez & Nogueira, 2013).

The machine has three states, normalise to whnf, normalise to nf, and apply. The configurations for each state are type-annotated by subscripts w, n, and a respectively. The machine decrements the level l (6th rule from the bottom) when leaving a $\lambda\!\!\lambda$ scope,

$$S \quad ::= \quad \mathbf{C}_0 \mid \mathbf{C}_1(\mathsf{C}) : S \mid \mathbf{C}_2 : S \mid \mathbf{C}_3(\mathsf{C}) \mid \mathbf{C}_4(\mathsf{C}) : S \mid \mathbf{C}_5(\mathsf{C}) : S$$

	T	$\to$	$(T[\epsilon], \mathbf{C}_0, 0)_n$
(if $n < \lvert\rho\rvert$)	$(n[\rho], S, l)_w$	$\to$	$(n^{\text{th}}(\rho), S, l)_w$
(if $n \geq \lvert\rho\rvert$)	$(n[\rho], S, l)_w$	$\to$	$(\lfloor n - (\lvert\rho\rvert - l)\rfloor, S, l)_a$
	$(\overline{n}, S, l)_w$	$\to$	$(\lfloor l - n\rfloor, S, l)_a$
	$(\lfloor n\rfloor, S, l)_w$	$\to$	$(\lfloor n\rfloor, S, l)_a$
	$((\lambda.B)[\rho], S, l)_w$	$\to$	$(\lambda\!\!\lambda.B[\overline{l+1} : \rho], S, l)_a$
	$((M\ N)[\rho], S, l)_w$	$\to$	$(M[\rho] \cdot N[\rho], S, l)_w$
	$(\mathsf{M} \cdot \mathsf{N}, S, l)_w$	$\to$	$(\mathsf{M}, \mathbf{C}_1(\mathsf{N}) : S, l)_w$
(if $n < \lvert\rho\rvert$)	$(n[\rho], S, l)_n$	$\to$	$(n^{\text{th}}(\rho), S, l)_n$
(if $n \geq \lvert\rho\rvert$)	$(n[\rho], S, l)_n$	$\to$	$(\lfloor n - (\lvert\rho\rvert - l)\rfloor, S, l)_a$
	$(\overline{n}, S, l)_n$	$\to$	$(\lfloor l - n\rfloor, S, l)_a$
	$(\lfloor n\rfloor, S, l)_n$	$\to$	$(\lfloor n\rfloor, S, l)_a$
	$((\lambda.B)[\rho], S, l)_n$	$\to$	$(\lambda\!\!\lambda.B[\overline{l+1} : \rho], S, l)_n$
	$(\lambda\!\!\lambda.B[\overline{l+1} : \rho], S, l)_n$	$\to$	$(B[\overline{l+1} : \rho], \mathbf{C}_2 : S, l+1)_n$
	$((M\ N)[\rho], S, l)_n$	$\to$	$(M[\rho] \cdot N[\rho] : S, l)_n$
	$(\mathsf{M} \cdot \mathsf{N}, S, l)_n$	$\to$	$(\mathsf{M}, \mathbf{C}_3(\mathsf{N}) : S, l)_w$
	$(\lambda\!\!\lambda.B[\overline{n} : \rho], \mathbf{C}_1(\mathsf{N}) : S, l)_a$	$\to$	$(B[\mathsf{N} : \rho], S, l)_w$
	$(\mathsf{M}, \mathbf{C}_1(\mathsf{N}) : S, l)_a$	$\to$	$(\mathsf{M} \cdot \mathsf{N}, S, l)_a$
	$(\mathsf{B}, \mathbf{C}_2 : S, l)_a$	$\to$	$(\lambda\!\!\lambda.\mathsf{B}, S, l-1)_a$
	$(\lambda\!\!\lambda.B[\overline{n} : \rho], \mathbf{C}_3(\mathsf{N}) : S, l)_a$	$\to$	$(B[\mathsf{N} : \rho], S, l)_n$
	$(\mathsf{M}, \mathbf{C}_3(\mathsf{N}) : S, l)_a$	$\to$	$(\mathsf{M}, \mathbf{C}_4(\mathsf{N}) : S, l)_n$
	$(\mathsf{M}, \mathbf{C}_4(\mathsf{N}) : S, l)_a$	$\to$	$(\mathsf{N}, \mathbf{C}_5(\mathsf{M}) : S, l)_n$
	$(\mathsf{N}, \mathbf{C}_5(\mathsf{M}) : S, l)_a$	$\to$	$(\mathsf{M} \cdot \mathsf{N}, S, l)_a$
	$(\mathsf{E}, \mathbf{C}_0, l)_a$	$\to$	E

Figure 6.9: Closure-converted eval/apply normal order abstract machine

thus mirroring rule ($\xi_{\widetilde{\rho}}$) in Figure 6.8. The functions `normalise4_whnf`, `normalise4_nf`, and `normalise4_cont` in the code make up the big-step tail-recursive implementation of the machine.

6.8.3 Functional correspondence

We apply refunctionalisation and inverse CPS to the big-step tail-recursive implementation of the machine shown in Section 6.8.2 and obtain the reduction-free normalisers `normalise6_whnf` and `normalise6_nf` that implement the big-step natural semantics in Figure 6.10.

$$\frac{n < |\rho| \quad \langle n^{\text{th}}(\rho), l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{N}, l\rangle}{\langle n[\rho], l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{N}, l\rangle}\ (\textsc{Var}_{\widetilde{bn}}) \qquad \frac{}{\langle \overline{n}, l\rangle \Downarrow_{\widetilde{bn}} \langle \lfloor l - n \rfloor, l\rangle}\ (\textsc{Par}_{\widetilde{bn}})$$

$$\frac{n \geq |\rho|}{\langle n[\rho], l\rangle \Downarrow_{\widetilde{bn}} \langle \lfloor n - (|\rho| - l) \rfloor, l\rangle}\ (\textsc{Fre}_{\widetilde{bn}}) \qquad \frac{}{\langle \lfloor n \rfloor, l\rangle \Downarrow_{\widetilde{bn}} \langle \lfloor n \rfloor, l\rangle}\ (\textsc{Abs}_{\widetilde{bn}})$$

$$\frac{}{\langle (\lambda.B)[\rho], l\rangle \Downarrow_{\widetilde{bn}} \langle \lambda\!\!\lambda.B[\overline{l+1} : \rho], l\rangle}\ (\textsc{Lam}_{\widetilde{bn}})$$

$$\frac{\langle M[\rho] \cdot N[\rho], l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{C}, l\rangle}{\langle (M\,N)[\rho], l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{C}, l\rangle}\ (\textsc{App}_{\widetilde{bn}})$$

$$\frac{\langle \mathsf{M}, l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{M}', l\rangle \quad \mathsf{M}' \equiv \lambda\!\!\lambda.B[\overline{l+1} : \rho] \quad \langle B[\mathsf{N} : \rho], l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{B}', l\rangle}{\langle \mathsf{M} \cdot \mathsf{N}, l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{B}', l\rangle}\ (\textsc{Red}_{\widetilde{bn}})$$

$$\frac{\langle \mathsf{M}, l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{M}', l\rangle \quad \mathsf{M}' \not\equiv \lambda\!\!\lambda.B[\overline{l+1} : \rho]}{\langle \mathsf{M} \cdot \mathsf{N}, l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{M}' \cdot \mathsf{N}, l\rangle}\ (\textsc{Neu}_{\widetilde{bn}})$$

$$\frac{n < |\rho| \quad \langle n^{\text{th}}(\rho), l\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{N}, l\rangle}{\langle n[\rho], l\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{N}, l\rangle}\ (\textsc{Var}_{\widetilde{no}}) \qquad \frac{}{\langle \overline{n}, l\rangle \Downarrow_{\widetilde{no}} \langle \lfloor l - n \rfloor, l\rangle}\ (\textsc{Par}_{\widetilde{no}})$$

$$\frac{n \geq |\rho|}{\langle n[\rho], l\rangle \Downarrow_{\widetilde{no}} \langle \lfloor n - (|\rho| - l) \rfloor, l\rangle}\ (\textsc{Fre}_{\widetilde{no}}) \qquad \frac{}{\langle \lfloor n \rfloor, l\rangle \Downarrow_{\widetilde{no}} \langle \lfloor n \rfloor, l\rangle}\ (\textsc{Abs}_{\widetilde{no}})$$

$$\frac{\langle \lambda\!\!\lambda.B[\overline{l+1} : \rho], l\rangle \Downarrow_{\widetilde{no}} \langle \lambda\!\!\lambda.\mathsf{B}', l\rangle}{\langle (\lambda.B)[\rho], l\rangle \Downarrow_{\widetilde{no}} \langle \lambda\!\!\lambda.\mathsf{B}', l\rangle}\ (\textsc{Lam}_{\widetilde{no}})$$

$$\frac{\langle B[\overline{l+1} : \rho], l+1\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{B}', l+1\rangle}{\langle \lambda\!\!\lambda.B[\overline{l+1} : \rho], l\rangle \Downarrow_{\widetilde{no}} \langle \lambda\!\!\lambda.\mathsf{B}', l\rangle}\ (\textsc{Bod}_{\widetilde{no}})$$

$$\frac{\langle M[\rho] \cdot N[\rho], l\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{C}, l\rangle}{\langle (M\,N)[\rho], l\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{C}, l\rangle}\ (\textsc{App}_{\widetilde{no}})$$

$$\frac{\langle \mathsf{M}, l\rangle \Downarrow_{\widetilde{bn}} \mathsf{M}' \quad \mathsf{M}' \equiv \lambda\!\!\lambda.B[\overline{l+1} : \rho] \quad \langle B[\mathsf{N} : \rho], l\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{B}', l\rangle}{\langle \mathsf{M} \cdot \mathsf{N}, l\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{B}', l\rangle}\ (\textsc{Red}_{\widetilde{no}})$$

$$\frac{\langle \mathsf{M}, l\rangle \Downarrow_{\widetilde{bn}} \langle \mathsf{M}', l\rangle \quad \mathsf{M}' \not\equiv \lambda\!\!\lambda.B[\overline{l+1} : \rho] \qquad \langle \mathsf{M}', l\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{M}'', l\rangle \quad \langle \mathsf{N}, l\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{N}', l\rangle}{\langle \mathsf{M} \cdot \mathsf{N}, l\rangle \Downarrow_{\widetilde{no}} \langle \mathsf{M}'' \cdot \mathsf{N}', l\rangle}\ (\textsc{Neu}_{\widetilde{no}})$$

Figure 6.10: Natural semantics of normal order in $\lambda_{\widetilde{\rho}}$

6.9 Shortcutting ephemeral expansion

Shortcutting ephemeral expansion consists of coalescing the rules that expand proper closures to ephemerals and of eliminating ephemeral constructs. For the latter we need to introduce a new preponing step.

6.9.1 Coalescing ephemeral expansion

We eliminate the rules that expand proper closures to ephemerals and obtain the natural semantics. We also eliminate levels in final results because they are no longer needed. The coalesced semantics is shown in Figure 6.11. Rule $\textsc{Lam}_{\widetilde{bn}}$ cannot be eliminated because closure abstractions are in $\mathrm{whnf}_{\mathsf{C}}$. Rules $\textsc{Lam}_{\widetilde{no}}$ and $\textsc{Bod}_{\widetilde{no}}$ are coalesced in rule $\textsc{Lam}_{\widetilde{no}^c}$. Rules $\textsc{App}_{\widetilde{bn}}$ and $\textsc{App}_{\widetilde{no}}$ are coalesced with Red and Neu rules of their respective calculus.

6.9.2 Preponing

We have to eliminate ephemeral constructs. Eliminating ephemeral closure abstractions is straightforward. First, we have to change rule $\textsc{Lam}_{\widetilde{bn}^c}$ so that it delivers the body $B[\overline{l+1} : \rho]$ which is a proper closure. Second, we have to modify the second premiss of $\textsc{Red}_{\widetilde{bn}^c}$, $\textsc{Neu}_{\widetilde{bn}^c}$, $\textsc{Red}_{\widetilde{no}^c}$, and $\textsc{Neu}_{\widetilde{no}^c}$, to a check on whether M' is a proper closure.

Eliminating ephemeral neutral closures would be possible if its operands were in nf_{C} after call-by-name. That is, if the reduction steps within the third premiss $\langle \mathsf{M}', l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{M}''$ of $\textsc{Neu}_{\widetilde{no}^c}$ that normalise the operands of neutral closures were preponed to the call-by-name steps of the first premiss $\langle \mathsf{M}, l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{M}'$ of that same rule. Fortunately, this can be easily achieved by copying the last premiss $\langle \mathsf{N}, l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{N}'$ in $\textsc{Neu}_{\widetilde{no}^c}$ and pasting it as the last premiss in $\textsc{Neu}_{\widetilde{bn}^c}$, and by removing the third premiss $\langle \mathsf{M}', l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{M}''$ in $\textsc{Neu}_{\widetilde{no}^c}$ which is no longer needed because M' would now be in nf_{C}.

The resulting $\textsc{Lam}_{\widetilde{bn}^p}$, $\textsc{Neu}_{\widetilde{bn}^p}$ and $\textsc{Neu}_{\widetilde{no}^p}$ rules, relabelled with a p superscript for readability, are shown below. The remaining rules stay as in Figure 6.11 save for the addition of the superscript.

$$\frac{}{\langle (\lambda.B)[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} B[\overline{l+1} : \rho]}\ (\textsc{Lam}_{\widetilde{bn}^p})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\widetilde{bn}^p} \mathsf{M}' \quad \mathsf{M}' \not\equiv B[\overline{l+1} : \rho] \quad \langle N[\rho], l\rangle \Downarrow_{\widetilde{no}^p} \mathsf{N}'}{\langle (M\ N)[\rho], l\rangle \Downarrow_{\widetilde{bn}^p} \mathsf{M}' \cdot \mathsf{N}'}\ (\textsc{Neu}_{\widetilde{bn}^p})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\widetilde{bn}^p} \mathsf{M}' \quad \mathsf{M}' \not\equiv B[\overline{l+1} : \rho] \quad \langle N[\rho], l\rangle \Downarrow_{\widetilde{no}^p} \mathsf{N}'}{\langle (M\ N)[\rho], l\rangle \Downarrow_{\widetilde{no}^p} \mathsf{M}' \cdot \mathsf{N}'}\ (\textsc{Neu}_{\widetilde{no}^p})$$

Due to preponing now $\Downarrow_{\widetilde{bn}^p}$ and $\Downarrow_{\widetilde{no}^p}$ are mutually recursive. The resulting preponed normaliser is implemented by functions `normalise15_whnf` and `normalise15_nf` in the code.

$$\frac{n < |\rho| \quad \langle n^{\text{th}}(\rho), l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{N}}{\langle n[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{N}}\ (\textsc{Var}_{\widetilde{bn}^c}) \qquad \frac{}{\langle \overline{n}, l\rangle \Downarrow_{\widetilde{bn}^c} \lfloor l - n \rfloor}\ (\textsc{Par}_{\widetilde{bn}^c})$$

$$\frac{n \geq |\rho|}{\langle n[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} \lfloor n - (|\rho| - l)\rfloor}\ (\textsc{Fre}_{\widetilde{bn}^c}) \qquad \frac{}{\langle \lfloor n \rfloor, l\rangle \Downarrow_{\widetilde{bn}^c} \lfloor n \rfloor}\ (\textsc{Abs}_{\widetilde{bn}^c})$$

$$\frac{}{\langle (\lambda.B)[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} \lambda\!\!\lambda.B[\overline{l+1} : \rho]}\ (\textsc{Lam}_{\widetilde{bn}^c})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{M}' \quad \mathsf{M}' \equiv \lambda\!\!\lambda.B[\overline{l+1} : \rho] \quad \langle B[N[\rho] : \rho], l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{B}'}{\langle (M\ N)[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{B}'}\ (\textsc{Red}_{\widetilde{bn}^c})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{M}' \quad \mathsf{M}' \not\equiv \lambda\!\!\lambda.B[\overline{l+1} : \rho]}{\langle (M\ N)[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{M}' \cdot \mathsf{N}}\ (\textsc{Neu}_{\widetilde{bn}^c})$$

$$\frac{n < |\rho| \quad \langle n^{\text{th}}(\rho), l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{N}}{\langle n[\rho], l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{N}}\ (\textsc{Var}_{\widetilde{no}^c}) \qquad \frac{}{\langle \overline{n}, l\rangle \Downarrow_{\widetilde{no}^c} \lfloor l - n \rfloor}\ (\textsc{Par}_{\widetilde{no}^c})$$

$$\frac{n \geq |\rho|}{\langle n[\rho], l\rangle \Downarrow_{\widetilde{no}^c} \lfloor n - (|\rho| - l)\rfloor}\ (\textsc{Fre}_{\widetilde{no}^c}) \qquad \frac{}{\langle \lfloor n \rfloor, l\rangle \Downarrow_{\widetilde{no}^c} \lfloor n \rfloor}\ (\textsc{Abs}_{\widetilde{no}^c})$$

$$\frac{\langle B[\overline{l+1} : \rho], l+1\rangle \Downarrow_{\widetilde{no}^c} \mathsf{B}'}{\langle (\lambda.B)[\rho], l\rangle \Downarrow_{\widetilde{no}^c} \lambda\!\!\lambda.\mathsf{B}'}\ (\textsc{Lam}_{\widetilde{no}^c})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{M}' \quad \mathsf{M}' \equiv \lambda\!\!\lambda.B[\overline{l+1} : \rho] \quad \langle B[N[\rho] : \rho], l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{B}'}{\langle (M\ N)[\rho], l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{B}'}\ (\textsc{Red}_{\widetilde{no}^c})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{M}' \quad \mathsf{M}' \not\equiv \lambda\!\!\lambda.B[\overline{l+1} : \rho] \qquad \langle \mathsf{M}', l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{M}'' \quad \langle N[\rho], l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{N}'}{\langle (M\ N)[\rho], l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{M}'' \cdot \mathsf{N}'}\ (\textsc{Neu}_{\widetilde{no}^c})$$

Figure 6.11: Coalesced natural semantics of normal order in $\lambda_{\widetilde{\rho}}$

Theorem 6.9.1 (Preponing is equivalence-preserving)**.** *Let* C *be a closure and* l *be the current lambda level. Take the following propositions:*

(i) $\langle \mathsf{C}, l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{C}'$ iff $\langle \mathsf{C}, l\rangle \Downarrow_{\widetilde{bn}^p} \mathsf{C}'$ *when* C' *is not a neutral.*

(ii) $\langle \mathsf{C}, l\rangle \Downarrow_{\widetilde{bn}^c} \mathsf{C}'$ iff $\langle \mathsf{C}, l\rangle \Downarrow_{\widetilde{bn}^p} \mathsf{C}''$ *when* C' *is a neutral and* C'' *is the normal form of* C'*.*

(iii) $\langle \mathsf{C}, l\rangle \Downarrow_{\widetilde{no}^c} \mathsf{C}''$ iff $\langle \mathsf{C}, l\rangle \Downarrow_{\widetilde{no}^p} \mathsf{C}''$

Proof. Proposition *(i)* holds because the NEU rule in the coalesced and preponed versions is never used in a derivation and the remaining rules are exactly the same in both versions. Propositions *(ii)* and *(iii)* hold assuming *(i)* and by simultaneous induction on $\Downarrow_{\widetilde{bn}}$ and $\Downarrow_{\widetilde{no}}$ derivations. □

The correctness of preponing can also be observed in the eval-readback version of the natural semantics. The reduction-free normalisers implementing the single-function and eval-readback semantics versions are inter-derivable by inverse and direct lightweight fusion by fixed-point promotion (Ohori & Sasano, 2007). (For the sake of completeness we have included their detailed inter-derivation in the code, see entry functions `normalise7` to `normalise14`). Recall from Section 6.4 that $\Downarrow_{no} = \Downarrow_{rn} \circ \Downarrow_{bn}$. Preponing here consists of moving to the call-by-name stage the first reduction steps of $\Downarrow_{rn}$ for applications, namely those of the first premiss $M \Downarrow_{rn} M'$ of APP_{rn}. In other words, it consists of shifting the point at which eval ends and readback begins when reducing neutrals. This is achieved by removing the first premiss $M \Downarrow_{rn} M'$ of APP_{rn}, and then copying the last two premisses $N \Downarrow_{bn} N'$ and $N' \Downarrow_{rn} N''$ of the same rule, and pasting them as the last two premisses of rule NEU_{bn} in Figure 6.4. The equivalence between the coalesced and the preponed versions of the natural semantics is stepwise, that is, both semantics contract the same redices in the same order.

6.9.3 Shortcut normaliser

After preponing, we are now in a position to eliminate ephemeral constructs. We have to distinguish final results from closures and will use ground terms without levels $\lfloor \Lambda \rfloor$ for final results, since levels are not needed in final results (Section 6.9.1). The resulting calculus which we call $\lambda^*_{\overline{\rho}}$ is similar to $\lambda_{\overline{\rho}}$ except that ground terms do not carry levels.

$$\lambda^*_{\overline{\rho}} \quad \begin{array}{lcl} \mathsf{C} & ::= & \Lambda[\rho] \mid \overline{n} \mid \lfloor \Lambda \rfloor \\ \rho & ::= & \epsilon \mid \mathsf{C} : \rho \end{array}$$

Shortcutting removes ephemeral constructors and rules $\text{ABS}_{\widetilde{bn}^p}$, $\text{APP}_{\widetilde{bn}^p}$, $\text{ABS}_{\widetilde{no}^p}$, and $\text{APP}_{\widetilde{no}^p}$ because only proper closures will be inputs. The resulting rules in Figure 6.12 now deliver ground terms except for $\text{VAR}_{\overline{bn}}$, $\text{RED}_{\overline{bn}}$, and $\text{LAM}_{\overline{bn}}$. Rule $\text{LAM}_{\overline{bn}}$ delivers a proper closure $B[\overline{l+1} : \rho]$ and rules $\text{VAR}_{\overline{bn}}$ and $\text{RED}_{\overline{bn}}$ simply propagate proper closures. The second premiss of $\text{RED}_{\overline{bn}}$ and $\text{RED}_{\overline{no}}$ checks if M' is a proper closure, and the second premiss of $\text{NEU}_{\overline{bn}}$ and of $\text{NEU}_{\overline{no}}$ checks if M' is a ground term. Functions `normalise16_whnf` and `normalise16_nf` in the code implement the shortcut natural semantics.

$$\frac{n < |\rho| \quad \langle n^{\text{th}}(\rho), l\rangle \Downarrow_{\overline{bn}} \mathsf{N}}{\langle n[\rho], l\rangle \Downarrow_{\overline{bn}} \mathsf{N}}\ (\textsc{Var}_{\overline{bn}}) \qquad \frac{}{\langle \overline{n}, l\rangle \Downarrow_{\overline{bn}} \lfloor l - n \rfloor}\ (\textsc{Par}_{\overline{bn}})$$

$$\frac{n \geq |\rho|}{\langle n[\rho], l\rangle \Downarrow_{\overline{bn}} \lfloor n - (|\rho| - l)\rfloor}\ (\textsc{Fre}_{\overline{bn}})$$

$$\frac{}{\langle (\lambda.B)[\rho], l\rangle \Downarrow_{\overline{bn}} B[\overline{l+1} : \rho]}\ (\textsc{Lam}_{\overline{bn}})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\overline{bn}} \mathsf{M'} \quad \mathsf{M'} \equiv B[\overline{n} : \rho] \quad \langle B[N[\rho] : \rho], l\rangle \Downarrow_{\overline{bn}} \mathsf{B'}}{\langle (M\,N)[\rho], l\rangle \Downarrow_{\overline{bn}} \mathsf{B'}}\ (\textsc{Red}_{\overline{bn}})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\overline{bn}} \mathsf{M'} \quad \mathsf{M'} \equiv \lfloor M' \rfloor \quad \langle N[\rho], l\rangle \Downarrow_{\overline{no}} \lfloor N' \rfloor}{\langle (M\,N)[\rho], l\rangle \Downarrow_{\overline{bn}} \lfloor M'\,N' \rfloor}\ (\textsc{Neu}_{\overline{bn}})$$

$$\frac{n < |\rho| \quad \langle n^{\text{th}}(\rho), l\rangle \Downarrow_{\overline{no}} \lfloor N \rfloor}{\langle n[\rho], l\rangle \Downarrow_{\overline{no}} \lfloor N \rfloor}\ (\textsc{Var}_{\overline{no}}) \qquad \frac{}{\langle \overline{n}, l\rangle \Downarrow_{\overline{no}} \lfloor l - n \rfloor}\ (\textsc{Par}_{\overline{no}})$$

$$\frac{n \geq |\rho|}{\langle n[\rho], l\rangle \Downarrow_{\overline{no}} \lfloor n - (|\rho| - l)\rfloor}\ (\textsc{Fre}_{\overline{no}})$$

$$\frac{\langle B[\overline{l+1} : \rho], l+1\rangle \Downarrow_{\overline{no}} \lfloor B' \rfloor}{\langle (\lambda.B)[\rho], l\rangle \Downarrow_{\overline{no}} \lfloor \lambda.B' \rfloor}\ (\textsc{Lam}_{\overline{no}})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\overline{bn}} \mathsf{M'} \quad \mathsf{M'} \equiv B[\overline{l+1} : \rho] \quad \langle B[N[\rho] : \rho], l\rangle \Downarrow_{\overline{no}} \lfloor B' \rfloor}{\langle (M\,N)[\rho], l\rangle \Downarrow_{\overline{no}} \lfloor B' \rfloor}\ (\textsc{Red}_{\overline{no}})$$

$$\frac{\langle M[\rho], l\rangle \Downarrow_{\overline{bn}} \mathsf{M'} \quad \mathsf{M'} \equiv \lfloor M' \rfloor \quad \langle N[\rho], l\rangle \Downarrow_{\overline{no}} \lfloor N' \rfloor}{\langle (M\,N)[\rho], l\rangle \Downarrow_{\overline{no}} \lfloor M'\,N' \rfloor}\ (\textsc{Neu}_{\overline{no}})$$

Figure 6.12: Shortcut natural semantics of normal order in $\lambda^*_{\overline{\rho}}$

$$\mathbf{c} \;::=\; \mathbf{w} \mid \mathbf{n}$$

$$\frac{n < |\rho| \quad \langle n^{\text{th}}(\rho), l, \mathbf{c}\rangle \Downarrow_{\overline{ctl}} \mathsf{N}}{\langle n[\rho], l, \mathbf{c}\rangle \Downarrow_{\overline{ctl}} \mathsf{N}}\ (\textsc{Var}_{\overline{ctl}}) \qquad \frac{}{\langle \overline{n}, l, \mathbf{c}\rangle \Downarrow_{\overline{ctl}} \lfloor l - n \rfloor}\ (\textsc{Par}_{\overline{ctl}})$$

$$\frac{n \geq |\rho|}{\langle n[\rho], l, \mathbf{c}\rangle \Downarrow_{\overline{ctl}} \lfloor n - (|\rho| - l) \rfloor}\ (\textsc{Fre}_{\overline{ctl}})$$

$$\frac{}{\langle (\lambda.B)[\rho], l, \mathbf{w}\rangle \Downarrow_{\overline{ctl}} B[\overline{l+1} : \rho]}\ (\textsc{Lam1}_{\overline{ctl}})$$

$$\frac{\langle B[\overline{l+1} : \rho], l+1, \mathbf{n}\rangle \Downarrow_{\overline{ctl}} \lfloor B' \rfloor}{\langle (\lambda.B)[\rho], l, \mathbf{n}\rangle \Downarrow_{\overline{ctl}} \lfloor \lambda.B' \rfloor}\ (\textsc{Lam2}_{\overline{ctl}})$$

$$\frac{\langle M[\rho], l, \mathbf{w}\rangle \Downarrow_{\overline{ctl}} \mathsf{M'} \quad \mathsf{M'} \equiv B[\overline{l+1} : \rho] \quad \langle B[N[\rho] : \rho], l, \mathbf{c}\rangle \Downarrow_{\overline{ctl}} \mathsf{B'}}{\langle (M\ N)[\rho], l, \mathbf{c}\rangle \Downarrow_{\overline{ctl}} \mathsf{B'}}\ (\textsc{Red}_{\overline{ctl}})$$

$$\frac{\langle M[\rho], l, \mathbf{w}\rangle \Downarrow_{\overline{ctl}} \mathsf{M'} \quad \mathsf{M'} \equiv \lfloor M' \rfloor \quad \langle N[\rho], l, \mathbf{n}\rangle \Downarrow_{\overline{ctl}} \lfloor N' \rfloor}{\langle (M\ N)[\rho], l, \mathbf{c}\rangle \Downarrow_{\overline{ctl}} \lfloor M'\ N' \rfloor}\ (\textsc{Neu}_{\overline{ctl}})$$

Figure 6.13: Natural semantics of normal order in $\lambda^*_{\overline{\rho}}$ with explicit control

6.10 From reduction-free normaliser to push/enter abstract machine

6.10.1 A reduction-free normaliser with explicit control

The mutually recursive $\Downarrow_{\overline{bn}}$ and $\Downarrow_{\overline{no}}$ of the shortcut natural semantics in Figure 6.12 differ in the treatment of abstractions. Rule $\textsc{Lam}_{\overline{bn}}$ takes place when the abstraction is applied to an operand whereas $\textsc{Lam}_{\overline{no}}$ takes place when the abstraction is unapplied. We transform the shortcut normalisers `normalise16_whnf` and `normalise16_nf` into a single `normalise_ctl` normaliser with explicit control that encodes the different treatment of abstractions. We introduce the control characters **w** and **n** that respectively encode a sub-derivation with $\textsc{Lam}_{\overline{bn}}$ and a sub-derivation with $\textsc{Lam}_{\overline{no}}$. The normaliser with explicit control implements the natural semantics of Figure 6.13. The control character **w** is used for operators in applications, and **n** for operands of neutral closures.

6.10.2 From reduction-free normaliser to eval/apply abstract machine

We apply defunctionalisation and CPS transformation to the normaliser with explicit control and obtain the following environment-based eval/apply machine with explicit control. The machine is implemented in the code by functions `normalise_ctl_cont` and `apply_ctl_cont`.

$$\begin{array}{rcl} S & ::= & \mathbf{C}_0 \mid \mathbf{C}_1(\mathsf{C},\mathbf{c}) : S \mid \mathbf{C}_2 : S \mid \mathbf{C}_3(\mathsf{C}) : S \\ \mathbf{c} & ::= & \mathbf{w} \mid \mathbf{n} \end{array}$$

	T	$\rightarrow$	$(T[\epsilon], \mathbf{C}_0, 0, \mathbf{n})$
(if $n < \lvert\rho\rvert$)	$(n[\rho], S, l, \mathbf{c})$	$\rightarrow$	$(n^{\text{th}}(\rho), S, l, \mathbf{c})$
(if $n \geq \lvert\rho\rvert$)	$(n[\rho], S, l, \mathbf{c})$	$\rightarrow$	$(\lfloor n - (\lvert\rho\rvert - l)\rfloor, S, l)$
	$(\overline{n}, S, l, \mathbf{c})$	$\rightarrow$	$(\lfloor l - n\rfloor, S, l)$
	$((\lambda.B)[\rho], S, l, \mathbf{w})$	$\rightarrow$	$(B[\overline{l+1} : \rho], S, l)$
	$((\lambda.B)[\rho], S, l, \mathbf{n})$	$\rightarrow$	$(B[\overline{l+1} : \rho], \mathbf{C}_2 : S, l+1, \mathbf{n})$
	$((M\,N)[\rho], S, l, \mathbf{c})$	$\rightarrow$	$(M[\rho], \mathbf{C}_1(N[\rho], \mathbf{c}) : S, l, \mathbf{w})$
	$(B[\overline{n} : \rho], \mathbf{C}_1(\mathsf{N}, \mathbf{c}) : S, l)$	$\rightarrow$	$(B[\mathsf{N} : \rho], S, l, \mathbf{c})$
	$(\lfloor M\rfloor, \mathbf{C}_1(\mathsf{N}, \mathbf{c}) : S, l)$	$\rightarrow$	$(\mathsf{N}, \mathbf{C}_3(\lfloor M\rfloor) : S, l, \mathbf{n})$
	$(\lfloor B\rfloor, \mathbf{C}_2 : S, l)$	$\rightarrow$	$(\lfloor\lambda.B\rfloor, S, l-1)$
	$(\lfloor N\rfloor, \mathbf{C}_3(\lfloor M\rfloor) : S, l)$	$\rightarrow$	$(\lfloor M\,N\rfloor, S, l)$
	$(\lfloor T\rfloor, \mathbf{C}_0, l)$	$\rightarrow$	T

The horizontal bar in the middle separates the eval configuration from the apply configuration. The eval configuration pattern-matches on the control character $\mathbf{c}$ to decide whether to go under lambda. The apply configuration does not use the control character. The occurrence of the control character discriminates both configurations and there is no need for type annotations. Observe the use of $\mathbf{w}$ when reducing operators in applications and the use of $\mathbf{n}$ when reducing operands in neutral closures. Observe that continuation $\mathbf{C}_1(\mathsf{C}, \mathbf{c})$ carries along the control character which is restored after contraction (first rule of the apply configuration).

6.10.3 Removing explicit control

Once the normaliser is in defunctionalised CPS we can observe the correlation between explicit control and the continuation stack. Control character $\mathbf{w}$ can be replaced by checking for the occurrence of $\mathbf{C}_1$ on the top of the stack, as we show next by constructing the grammar of well-formed stacks of the eval/apply machine with explicit control.

$$\begin{array}{rcl} S & ::= & A\,\mathbf{C}_0 \\ A & ::= & \varepsilon \\ & \mid & \{\,\mathbf{C}_1(\mathsf{C}, \mathbf{w}) : \}^*\,\{\,\mathbf{C}_2 : \}^* \\ & \mid & A\,\mathbf{C}_3(\lfloor n\{\mathsf{NF}\}^*\rfloor) \,:\, \{\,\mathbf{C}_1(\mathsf{C}, \mathbf{w}) : \}^*\,\{\,\mathbf{C}_2 : \}^* \end{array}$$

The stack S starts always with the initial continuation $\mathbf{C}_0$. Non-terminal A is introduced to exclude the initial continuation in further (and optional since A derives to ε) recursive occurrences of a well-formed segment of the stack. The machine can go under lambda ($\mathbf{C}_2$) but never after pushing operands on the stack ($\mathbf{C}_1(\mathsf{C}, \mathbf{w})$) because otherwise contraction would occur. The operands are only pushed on the stack with control character $\mathbf{w}$, and that control character is preserved after contraction (*i.e.*, when a lambda abstraction is found, and after retrieving the operand from the stack and pushing it on the environment). If a formal parameter or a free variable is found, its ground index n is embedded into a ground term and pushed on the stack ($\mathbf{C}_3(\lfloor n \rfloor)$). Ground terms are only modified by applying them to other ground terms, which by definition are nfs. Thus, ground terms coincide with neutral terms in normal form $\lfloor n\{\mathsf{NF}\}^* \rfloor$. Once a ground term is pushed on the stack, the machine resumes execution with control character $\mathbf{n}$, which is signaled by the recursive occurrence of A in the grammar of stacks.

The occurrence of $\mathbf{C}_1$ on the top of the stack determines the different treatment of abstractions. We use this fact to eliminate explicit control from the eval/apply machine and obtain the following eval/apply machine with implicit control:

$$S \quad ::= \quad \mathbf{C}_0 \mid \mathbf{C}_1(\mathsf{C}) : S \mid \mathbf{C}_2 : S \mid \mathbf{C}_3(\mathsf{C}) : S$$

	T	$\to$	$(T[\epsilon], \mathbf{C}_0, 0)_e$
(if $n < \lvert\rho\rvert$)	$(n[\rho], S, l)_e$	$\to$	$(n^{\text{th}}(\rho), S, l)_e$
(if $n \geq \lvert\rho\rvert$)	$(n[\rho], S, l)_e$	$\to$	$(\lfloor n - (\lvert\rho\rvert - l) \rfloor, S, l)_a$
	$(\overline{n}, S, l)_e$	$\to$	$(\lfloor l - n \rfloor, S, l)_a$
	$((\lambda.B)[\rho], \mathbf{C}_1(\mathsf{N}) : S, l)_e$	$\to$	$(B[\overline{l+1} : \rho], \mathbf{C}_1(\mathsf{N}) : S, l)_a$
	$((\lambda.B)[\rho], S, l)_e$	$\to$	$(B[\overline{l+1} : \rho], \mathbf{C}_2 : S, l+1)_e$
	$((M\,N)[\rho], S, l)_e$	$\to$	$(M[\rho], \mathbf{C}_1(N[\rho]) : S, l)_e$
	$(B[\overline{n} : \rho], \mathbf{C}_1(\mathsf{N}) : S, l)_a$	$\to$	$(B[\mathsf{N} : \rho], S, l)_e$
	$(\lfloor M \rfloor, \mathbf{C}_1(\mathsf{N}) : S, l)_a$	$\to$	$(\mathsf{N}, \mathbf{C}_3(\lfloor M \rfloor) : S, l)_e$
	$(\lfloor B \rfloor, \mathbf{C}_2 : S, l)_a$	$\to$	$(\lfloor \lambda.B \rfloor, S, l-1)_a$
	$(\lfloor N \rfloor, \mathbf{C}_3(\lfloor M \rfloor) : S, l)_a$	$\to$	$(\lfloor M\,N \rfloor, S, l)_a$
	$(\lfloor T \rfloor, \mathbf{C}_0, l)_a$	$\to$	T

Type annotations are required again to distinguish the eval and apply configurations. The machine with implicit control is implemented by functions `normalise20_cont` and `apply20_cont` in the code.

Pattern-matching on the stack breaks the shallow inspection required to refunctionalise the machine, but this context-dependency is present in KN and has to be introduced at some point in order to derive the machine.

6.10.4 From eval/apply to push/enter machine

To turn the machine into push/enter, the apply function has to be inlined in eval. There are three eval transitions going to apply, namely the 2nd, 3th, and 4th. The last can be

inlined ('compressed') with the first transition of apply. To inline the other two we first 'protrude' (inverse inline) them into a new eval transition for ground terms going to apply:

$$\begin{array}{rrcl} (\text{if } n \geq |\rho|) & (n[\rho], S, l)_e & \to & (\lfloor n - (|\rho| - l)\rfloor, S, l)_e \\ & (\overline{n}, S, l)_e & \to & (\lfloor l - n\rfloor, S, l)_e \\ & (\lfloor n\rfloor, S, l)_e & \to & (\lfloor n\rfloor, S, l)_a \end{array}$$

The rest of the transitions remain the same and are omitted. The protruded machine is implemented in the code by functions `normalise21_cont` and `apply21_cont`. We inline the transitions of apply for ground terms into the new transition in the protruded machine and obtain the push/enter machine below.

$$S \quad ::= \quad \mathbf{C}_0 \mid \mathbf{C}_1(\mathsf{C}) : S \mid \mathbf{C}_2 : S \mid \mathbf{C}_3(\mathsf{C}) : S$$

	T	$\to$	$(T[\epsilon], \mathbf{C}_0, 0)$
(if $n < \lvert\rho\rvert$)	$(n[\rho], S, l)$	$\to$	$(n^{\text{th}}(\rho), S, l)$
(if $n \geq \lvert\rho\rvert$)	$(n[\rho], S, l)$	$\to$	$(\lfloor n - (\lvert\rho\rvert - l)\rfloor, S, l)$
	$(\overline{n}, S, l)$	$\to$	$(\lfloor l - n\rfloor, S, l)$
	$((\lambda.B)[\rho], \mathbf{C}_1(\mathsf{N}) : S, l)$	$\to$	$(B[\mathsf{N} : \rho], S, l)$
	$((\lambda.B)[\rho], S, l)$	$\to$	$(B[\overline{l+1} : \rho], \mathbf{C}_2 : S, l+1)$
	$((M\,N)[\rho], S, l)$	$\to$	$(M[\rho], \mathbf{C}_1(N[\rho]) : S, l)$
	$(\lfloor M\rfloor, \mathbf{C}_1(\mathsf{N}) : S, l)$	$\to$	$(\mathsf{N}, \mathbf{C}_3(\lfloor M\rfloor) : S, l)$
	$(\lfloor B\rfloor, \mathbf{C}_2 : S, l)$	$\to$	$(\lfloor\lambda.B\rfloor, S, l-1)$
	$(\lfloor N\rfloor, \mathbf{C}_3(\lfloor M\rfloor) : S, l)$	$\to$	$(\lfloor M\,N\rfloor, S, l)$
	$(\lfloor T\rfloor, \mathbf{C}_0, l)$	$\to$	T

The machine is implemented in the code by function `normalise22_push`.

Save for two minor *visual* differences that we discuss in the next paragraph, this machine is an optimised version of the original KN that can work with open terms. The optimisation is minor: embedded ground terms do not carry a level, so such levels need not be recovered from the environment when reducing operands of neutral closures, because the machine decrements the current level when leaving a lambda scope, as specified by rule ($\xi_{\widetilde{\rho}}$) in the structural operational semantics of normal order in Figure 6.8. Naturally, the machine can take closed terms as input. The clause for free variables would simply not be used.

The visual differences with the original KN are the following. First, the use of n^{th} for look-up instead of a recursive peeling-off of the environment (n^{th} can be implemented by recursive peel-off, but also by random access). Second, the presence of defunctionalised continuations coming from the stack S of the push/enter machine.

We remove the visual differences. The n^{th} function for look-up is replaced by a peeling-off definition (consequently, the transition for free variables $n[\epsilon]$ has to be adapted). And defunctionalised continuations in S are replaced by the constructors of $\lambda^*_{\widetilde{\rho}}$ (and the control

character λ) that they represent (collected in stack S in the optimised machine below).

$$S \ ::= \ \epsilon \mid \Lambda[\rho] : S \mid \lambda : S \mid \lfloor \Lambda \rfloor : S$$

T	$\rightarrow$	$(T[\epsilon], \epsilon, 0)$
$((n+1)[\mathsf{C} : \rho], S, l)$	$\rightarrow$	$(n[\rho], S, l)$
$(0[\mathsf{C} : \rho], S, l)$	$\rightarrow$	(C, S, l)
$(n[\epsilon], S, l)$	$\rightarrow$	$(\lfloor n + l \rfloor, S, l)$
$((M\ N)[\rho], S, l)$	$\rightarrow$	$(M[\rho], N[\rho] : S, l)$
$((\lambda.B)[\rho], N[\rho'] : S, l)$	$\rightarrow$	$(B[\overline{N[\rho']} : \rho], S, l)$
$((\lambda.B)[\rho], S, l)$	$\rightarrow$	$(B[\overline{l+1} : \rho], \lambda : S, l+1)$
$(\overline{n}, S, l)$	$\rightarrow$	$(\lfloor l - n \rfloor, S, l)$
$(\lfloor M \rfloor, N[\rho] : S, l)$	$\rightarrow$	$(N[\rho], \lfloor M \rfloor : S, l)$
$(\lfloor B \rfloor, \lambda : S, l)$	$\rightarrow$	$(\lfloor \lambda.B \rfloor, S, l-1)$
$(\lfloor N \rfloor, \lfloor M \rfloor : S, l)$	$\rightarrow$	$(\lfloor M\ N \rfloor, S, l)$
$(\lfloor T \rfloor, \epsilon, l)$	$\rightarrow$	T

The machine is implemented by function `normalise23_push` in the code.
We have derived KN and are now at the end of our journey.

6.11 Related and future work

Single-function and eval-readback (Section 6.4) approaches require different CPS transformations. For the former, a single-layer CPS without control delimiters is enough (García-Pérez & Nogueira, 2013) because reduction is performed in a single stage. All the artefacts shown in this chapter are single-function. For eval-readback, either a two-layer CPS or a single-layer CPS with control delimiters is required (Biernacka *et al.*, 2005). Both NBE and eval-readback are popular within the programming languages community. However, single-function structural and natural semantics are conceptually simpler, and their implementations more amenable to program transformation because no specific CPS techniques nor meta-theory for delimiting control is required.

In (Danvy *et al.*, 2013) they present a derivation involving the full-reducing machine of Curien (Curien, 1993) itself based on the KAM machine (Crégut, 1990). Our work and (Danvy *et al.*, 2013) have been independently developed and are, to our knowledge, the only works demonstrating the derivation of full-reducing machines. The differences between our work and (Danvy *et al.*, 2013) are substantial.

- The full-reducing machines are different. Moreover, we *arrive* at KN whereas (Danvy *et al.*, 2013) *departs* from Curien's machine.
- We follow a single-function approach to derive KN, and use single-layer CPS and plain CPS-related techniques. In contrast, (Danvy *et al.*, 2013) follows the eval-

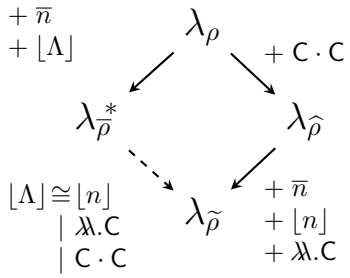
readback approach present in Curien's machine and presents two derivation paths, one using two-layer CPS and another using single-layer CPS with control delimiters.

- The precise control of levels in $\lambda\!\!\lambda$ scopes (rules $\textsc{Lam}_{\widetilde{\rho}}$ and $(\xi_{\widetilde{\rho}})$ in Figure 6.8) results in index alignment and balanced derivations which makes reasoning by structural induction easier and substantiates the optimisation of the original KN machine (Section 6.7.1). In (Danvy *et al.*, 2013) environments carry a lexical adjustment value that is incremented when popping a binding off the environment which complicates reasoning by structural induction on environments.

- In Section 6.10, we introduce explicit control to combine the hybrid and subsidiary reduction-free normalisers into one, and derive an explicit-control eval/apply abstract machine. When removing explicit control the resulting machine is context-dependent, *i.e.*, it does not have the *shallow-inspection* property. This prevents the refunctionalisation of the machine. However, the problem is not in our derivation but in the fact that context-dependency is present in KN, and has to be introduced at some point in order to derive the machine. In any case, we have shown in Sections 6.8.2 and 6.10.2 that environment-based machines with the shallow-inspection property can be derived. In (Danvy *et al.*, 2013), machines do not have explicit control because two-layer CPS or single-layer CPS with control delimiters are used.

In (Grégoire & Leroy, 2002) a full-reducing strategy is specified in eval-readback style that is used in a proof assistant. The eval stage $\mathcal{V}(T)$ is implemented by an optimised, precompiled abstract machine. This machine has been contrived, not derived. The readback stage $\mathcal{N}(T)$ is symbolic. The strategy resulting from the composition of $\mathcal{V}(T)$ and $\mathcal{N}(T)$ is the same as the strategy resulting from the composition of symbolic `eval` and symbolic `byValue` in (Paulson, 1996, p.390), save for the right-to-left sequencing order in which operands are reduced before operators. (The strategy implements strict semantics for redices, but performs β-reduction, not the β_V-reduction of the lambda-value calculus of (Plotkin, 1975), and consequently, it is not a full-reducing strategy of that calculus.) We are currently studying the derivation of a *whole* machine from the single-function natural semantics obtained (via lightweight fusion by fixed-point promotion) from eval-readback `eval` and `byValue`. A question to answer is whether optimisations can be incorporated by program transformation.

The closure calculi $\lambda_{\widetilde{\rho}}$ and $\lambda_{\widetilde{\rho}}^{*}$ we have introduced are rather natural extensions of λ_{ρ}, as illustrated by the following diagram:



We have not defined the reduction theory of $\lambda_{\widetilde{\rho}}$, only presented the reduction strategy $\widetilde{no}$, which has taken us to the KN machine. Such theory is of interest since it has to consider compatibility with environments (reducing bindings inside environments) which poses a challenge.

Addendum

6.12 The reduction theory of $\lambda_{\widetilde{\rho}}$

In Section 6.11 we commented on the challenge of defining the reduction theory of $\lambda_{\widetilde{\rho}}$. The problem was to reduce bindings inside environments, for which the appropriate lambda nesting level has to be figured out. The solution to this problem appears implicit in Definition 6.7.1 (well-formed proper closures) and Lemma 6.7.4 (invariant on closures). The *parameters-as-levels* enforces that the global nesting level is incremented with every insertion of a de Bruijn level in the environment, and thus, the proper bindings between two consecutive de Bruijn levels are to be reduced at the lambda nesting level coinciding with the de Bruijn level at the right. This enables the compatibility rule for the bindings in environments that we depict below:

$$\frac{\langle \mathsf{C}, n\rangle \to_{\widetilde{\rho}} \langle \mathsf{C}', n\rangle}{\langle T[\ldots : \overline{n+1} : \ldots : \mathsf{C} : \ldots : \overline{n} : \ldots], l\rangle \to_{\widetilde{\rho}} \langle T[\ldots : \overline{n+1} : \ldots : \mathsf{C}' : \ldots : \overline{n} : \ldots], l\rangle}\ \textsc{Env}_{\widetilde{\rho}}$$

The $\overline{n+1}$ on the left may not exist, in which case $n = l$ where l is the global nesting level. If no de Bruijn level exists at the right of a closure C, then the C has to be reduced with lambda nesting level 0. Notice that de Bruijn levels in environments can be duly reduced to closures in normal form by subtracting them from the global nesting level l and enclosing them in a ground index.

Rule $\textsc{Env}_{\widetilde{\rho}}$ opens up for the definition of reduction strategies different form normal order. Rule $\textsc{Env}_{\widetilde{\rho}}$, together with the notions of reduction in Figure 6.8, and with the the compatibility rules as the ones in Definition 6.7.7 defines a true reduction relation in $\lambda_{\widetilde{\rho}}$. This reduction relation simulates the reduction relation $\to_\beta$ in plain λK, which is straightforward to prove in the light of Lemmata 6.7.2, 6.7.3, and Theorem 6.7.9. This contribution opens up for the definition in $\lambda_{\widetilde{\rho}}$ of reduction strategies different from normal order.

6.13 Comparative between our inter-derivation and (Munk, 2008)

When we first wrote the manuscript on which Chapter 6 is based, we were not aware of the work (Munk, 2008), where he departs from KN and arrives to a (context-based) reduction semantics that implements full reduction in a calculus of closures. Thus, ours is

not the first inter-derivation of KN. In what follows we summarise the differences between Chapter 6 and (Munk, 2008):

- Section 11.4 of (Munk, 2008) states that a version of the full-reducing Krivine machine KN of Crégut (Crégut, 2007) is derived from the reduction semantics of a full-reducing strategy which, although not stated explicitly, is normal order in eval-readback style in a calculus of closures. A functional correspondence is mentioned, but the intermediate refunctionalised normaliser is missing. A syntactic correspondence is also claimed, but no derivation is provided for it.

  We have tried to implement the syntactic correspondence indicated by Munk, unsuccessfully due to several errors in the presentation.

  For example, Munk starts with a grammar of reduction contexts with two mutually dependent layers:

  $$\begin{array}{rcl} \mathcal{A}[\,] & ::= & [\,] \mid \mathcal{A}[\lambda[\,]] \mid \mathcal{C}[a\,[\,]] \\ \mathcal{C}[\,] & ::= & \mathcal{A}[\,] \mid \mathcal{C}[[\,]\,c] \end{array}$$

  This semantics is the closure-converted version of the reduction semantics that Munk introduces in Section 7.2, which is the one coinciding with our preponed reduction semantics (save for the minor visual use of inside-out contexts, which explains the nested square brackets $[\ldots[\,]\ldots]$ in his notation). The a and c are respectively Munk's closure-converted normal forms and terms, which stand for our neutrals in normal form and our terms.

  In particular, the proposed reduction semantics (Page 128) is incorrect with respect to his calculus of closures (the 'term language' in Page 127). It is not clear which syntactic construct corresponds to the left-hand-side of Munk's contraction rule,

  $$\mathit{Subst}: \quad 1[t[s',m']\cdot s,m] \;\rightarrow\; t[s',m]$$

  since $t[s',m']$ is not a value v, and environments s can only be empty $\bullet$ or contain values v. The same could be said about the left-hand-side of rule Subst' and the right-hand-side of rule *Beta*, where a closure c occurs in the environment. Munk also distinguishes the notion of reduction *Abs* which only applies to contexts $\mathcal{A}[\,]$. The role of this notion of reduction is to expand abstraction closures into closure abstractions (akin to our ephemeral construct $\lambda\!\!\lambda$) thus providing enough syntactical structure as to define the small-step reduction semantics. This expansion resembles the closure-level application constructor in (Biernacka & Danvy, 2007).

  As for the functional correspondence, it is carried after *repeated substantial alterations* in each transformation step to KN's original calculus of closures (Crégut, 2007). These alterations allow Munk to disentangle two auxiliary continuation and meta-continuation dispatchers by means of non-standard derivation steps, which in turn permit a functional correspondence in the same spirit as (Danvy *et al.*, 2013) (also co-authored by Munk) and which uses eval-readback and 2CPS. Alas, the intermediate refunctionalised normaliser is not shown.

If the functional correspondence is to be carried using single functions and 1CPS (as opposed to eval-readback and 2CPS) to obtain well-formed continuation stacks with unique-decomposition, and an abstract machine with shallow inspection, the same problems that we described in Chapter 5 arise and the techniques we introduced there are required. We have commented at length on the differences between (Danvy *et al.*, 2013) and our contribution in Section 6.11 and a similar discussion would apply to (Munk, 2008).

- Whereas in (Munk, 2008) the reduction semantics is given in preponed style, our preponed semantics is obtained during a derivation step, for we depart again from search functions implementing the hybrid SOS of normal order.

  Our derivation departs from a different starting point and considers preponing a step. The preponed semantics is equivalent to the hybrid-style semantics, as proven by simultaneous induction.

  We present a syntactic and a functional correspondence where the closure calculus remains the same, modulo the isomorphism $\Lambda = \overline{n} \mid \lambda\!\!\lambda.\mathsf{C} \mid \mathsf{C} \cdot \mathsf{C}$. KN intrinsically implements a preponed normal order, and hence the detour in Section 6.9 is inexorable if we want to connect the standard (i.e., non-preponed) normal order to KN. The preponed step could arguably be performed in the small-step semantics (either in the SOS or in the context-based reduction semantics) but doing so would be contrived, and a relevant proof of its correctness (which we do provide) would still be needed.

  The preponed version can be easily arrived at after applying a standard program transformation technique, namely LWF, but only in the natural semantics presentation. This is why we preferred to show the preponing step at that point.

  Our derivation is not a consequence of implementation choices, but rather a principled approach to reuse most of the standard program-transformation steps and minimise the ad hoc steps. The preponing step could have only been moved to other places in the derivation (SOS, context-based reduction semantics), where it would still have been required.

  Irreducible closures:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{WNF_C} & ::= & \lambda\!\!\lambda.\Lambda[\rho] & \mathsf{NF_C} & ::= & \lambda\!\!\lambda.\mathsf{NF_C} \\ & \mid & \lfloor n \rfloor\{\cdot\,\mathsf{NF_C}\}^* & & \mid & \lfloor n \rfloor\{\cdot\,\mathsf{NF_C}\}^* \end{array}$$

  Preponed reduction semantics:

$$\begin{array}{rcl} \mathbf{C}^l_{\widetilde{no}}[\,] & ::= & [\,] \mid \mathbf{C}^l_{\widetilde{bn}'}[\,] \cdot \mathsf{C} \mid \lfloor n \rfloor\{\cdot\,\mathsf{NF_C}\}^* \cdot \mathbf{C}^l_{\widetilde{no}}[\,] \mid \lambda\!\!\lambda.\mathbf{C}^l_{\widetilde{no}}[\,] \\ \mathbf{C}^l_{\widetilde{bn}'}[\,] & ::= & [\,] \mid \mathbf{C}^l_{\widetilde{bn}'}[\,] \cdot \mathsf{C} \mid \lfloor n \rfloor\{\cdot\,\mathsf{NF_C}\}^* \cdot \mathbf{C}^l_{\widetilde{no}}[\,] \end{array}$$

Preponed structural operational semantics:

$$\frac{\mathsf{M} \notin \mathsf{WNF_C} \quad \langle \mathsf{M}, l\rangle \rightarrow_{\widetilde{bn}'} \langle \mathsf{M}', l\rangle}{\langle \mathsf{M}\cdot\mathsf{N}, l\rangle \rightarrow_{\widetilde{bn}'} \langle \mathsf{M}'\cdot\mathsf{N}, l\rangle}\ (\mu_{\widetilde{bn}'})$$

$$\frac{\mathsf{M} \in \mathsf{WNF_C} \quad \mathsf{M} \not\equiv \lambda\!\!\lambda.\mathsf{B} \quad \langle \mathsf{N}, l\rangle \rightarrow_{\widetilde{no}} \langle \mathsf{N}', l\rangle}{\langle \mathsf{M}\cdot\mathsf{N}, l\rangle \rightarrow_{\widetilde{bn}'} \langle \mathsf{M}\cdot\mathsf{N}', l\rangle}\ (\nu_{\widetilde{bn}'})$$

$$\frac{\mathsf{M} \notin \mathsf{WNF_C} \quad \langle \mathsf{M}, l\rangle \rightarrow_{\widetilde{bn}'} \langle \mathsf{M}', l\rangle}{\langle \mathsf{M}\cdot\mathsf{N}, l\rangle \rightarrow_{\widetilde{no}} \langle \mathsf{M}'\cdot\mathsf{N}, l\rangle}\ (\mu_{\widetilde{no}})$$

$$\frac{\mathsf{M} \in \mathsf{WNF_C} \quad \mathsf{M} \not\equiv \lambda\!\!\lambda.\mathsf{B} \quad \langle \mathsf{N}, l\rangle \rightarrow_{\widetilde{no}} \langle \mathsf{N}', l\rangle}{\langle \mathsf{M}\cdot\mathsf{N}, l\rangle \rightarrow_{\widetilde{no}} \langle \mathsf{M}\cdot\mathsf{N}', l\rangle}\ (\nu_{\widetilde{no}})$$

$$\frac{\langle \mathsf{B}, l+1\rangle \rightarrow_{\widetilde{no}} \langle \mathsf{B}', l+1\rangle}{\langle \lambda\!\!\lambda.\mathsf{B}, l\rangle \rightarrow_{\widetilde{no}} \langle \lambda\!\!\lambda.\mathsf{B}', l\rangle}\ (\xi_{\widetilde{no}})$$

But the natural semantics seems the place to apply the preponing step, because there it is a trivial transformation after applying LWF. We have deliberately avoided attempting the preponing step in the abstract machines themselves. Consider trying to prove the machine in Section 6.7.1 equivalent to either the machines in Sections 6.8.2 or 6.9.3. That would have resulted in an unnecessarily complex derivation. The explicit control is also compulsory. This control (which is later turned into a check for the occurrence of constructor $\mathbf{C}_2$ in the stack) is precisely what prevents KN to have shallow-inspection. However, the *two modes* for no and bn' have to be fused into an *explicitly-controlled single mode* before arriving to the machine itself, in order to correlate the control with the occurrence of $\mathbf{C}_2$. Again, the right place to introduce the explicit control is the natural semantics, and correctness of this transformation is direct.

Because KN does not have shallow inspection we must use preponing at some point and explicit control. The modified $\Downarrow_{bn'}$ is a consequence of preponing, and is consistent with the discussion of hybrid and uniform in Section 5.3 ($\Downarrow_{bn'}$ is the semantics resulting by adding the hole to contexts $\mathbf{C}_{ne}[\,]$ there). In Munk's work, as in (Danvy *et al.*, 2013), this is not a concern because the eval-readback approach implements this control either with control delimiters or by the layering which is implicit in 2CPS. Control delimiters and explicit control must not be confused. The former delimits the computation, disentangling what happens before and after reaching an intermediate irreducible form, whereas the latter only tracks the *mode* (no or bn') that will resume the computation at a given moment. Explicit control is compulsory if we aim to keep the single-stage (as opposed to eval-readback) and the single-layer CPS which are present in KN.

- Our work exposes the preponed nature and explicit control present in KN. Munk's

calculus of closures does not truly reflect the internals of KN. Munk disentangles a function `aux` that operates on stacks and level-annotated ground terms P, by removing the global lambda level and the environment. Then, he hits the mark on ground terms being normal forms, and removes the level annotation from ground terms. To do so, he has to introduce this level annotation in the environment. However, disentangling does not allow to remove level annotations completely, as we do by incrementing the global lambda level after leaving a lambda scope. Munk also removes formal parameters (our $\overline{n}$) from U closures and places them *only* in the environments. As a result, the new machine operates on standard lambda terms with an environment which has the formal parameters, and where the stack stores the level-annotated ground terms. This results in a machine with three configurations, KN, `aux` and `aux'`, which corresponds functionally (through the use of 2CPS transformations) to a big-step eval-readback normalisation function in the modified calculus. This is in essence very similar to (Danvy *et al.*, 2013).

- We obtain KN from the SOS of normal order in a calculus of closures that, unlike Munk, does not change through the derivation (modulo the isomorphism between ground terms and ephemerals). The SOS uses *parameters-as-levels* (present in KN) and enjoys good meta-theoretic properties such as *index alignment* and *balanced derivations*, two of the main ideas, which makes the SOS amenable to proof by induction which is one of the contributions. These follow from the treatment of formal parameters and lambda scopes (implying that you have to keep formal parameters in your calculus of closures). Index alignment turns the syntactic adjustments into a global and uniform concern, placing the required machinery (lexical offsets) out of the environments. Balanced derivations enables easy reasoning by structural induction over the SOS, since there is no need to look at the inners of any closure. A fair derivation would have to keep the inherent features of KN in the artefacts. In a way, Munk's resorts to clever tricks and shortcuts (disentangling, simplifying the calculus, using 2CPS techniques) while we just deal with the inherent nature of the machine directly, minimising the tricks to a protrude step in Section 10.4. (akin to the disentangling of aux function in Munk's).

- We have been inspired by the foundational ideas of (Biernacka & Danvy, 2007) regarding substitution functions and proved that normal order in the calculus of closures commutes with normal order in the pure lambda calculus via substitution functions. Again, the intuition is the idea of hybrid strategy. The reduction semantics and other artefacts are obtained by derivation. Constructing the grammar of well-formed continuation stacks was necessary this time to recover explicit control. As we said before, preponing is introduced as a derivation step, we do not depart from a preponed semantics, but from a hybrid-style SOS, which in our opinion is easier to understand and motivate.

- We must add that Munk does not explicitly call his eval-readback function 'normal

> order' although it is that strategy (we know that because we do show the eval-readback version of normal order in Section 6.4). There are other normalising and complete strategies in the lambda calculus (e.g., hybrid normal order (Sestoft, 2002)), so the explicit connection with normal order is not made, although it is there.

Our aim has been to reuse standard techniques and to unearth the intrinsics of KN and keep ad hoc transformation steps to a minimum. Our single-stage and their eval-readback approach require different CPS transformations. For our single-stage artefacts a single-layer CPS without control delimiters is enough. For their eval-readback artefacts, either a two-layer CPS or a single-layer CPS with control delimiters is required (Danvy *et al.*, 2013; Biernacka *et al.*, 2005). An advantage of the eval-readback approach is that the two stages (with their types, continuations, etc) are disentangled and modular. However, in eval-readback the set of CPS and defunctionalisation transformations get more complicated and less direct. We believe that single-stage implementations are more amenable to program transformation because no specific CPS techniques nor meta-theory for delimiting control is required.

By unearthing the intrinsics of KN, we have also arrived at index alignment, balanced derivations, step-by-step correspondence, which are also key contributions and are not present in Munk's work.

Part III

Gradual Typing

7

Deriving Interpretations of the Gradually-Typed Lambda Calculus

Siek & Garcia (2012) have explored the dynamic semantics of the gradually-typed lambda calculus by means of definitional interpreters and abstract machines. The correspondence between the calculus's mathematically described small-step reduction semantics and the implemented big-step definitional interpreters was left as a conjecture. We prove and generalise Siek and Garcia's conjectures using program transformation. We establish the correspondence between the definitional interpreters and the reduction semantics of a closure-converted gradually-typed lambda calculus that unifies and amends various versions of the calculus. We use a hybrid approach and two-layer continuation-passing style so that the correspondence is parametric on the subsidiary coercion calculus. We have implemented the whole derivation for the eager error-detection policy and the downcast blame-tracking strategy. The correspondence can be established for other choices of error-detection policies and blame-tracking strategies, by plugging in the appropriate artefacts for the particular subsidiary coercion calculus.

7.1 Introduction

Since the publication of (Ager *et al.*, 2003b) a decade ago there has been substantial research on inter-derivation by program transformation of implementations of 'semantic artefacts', *i.e.*, implementations of operational semantics, denotational semantics, and abstract machines. The research has established a semantics framework and has contributed to the repertoire of program transformation techniques. Unfortunately, inter-derivation remains underused. Languages and calculi constantly spring up but their semantics are specified on paper and their correspondences are either obviated, conjectured, or proven mathematically.

We think inter-derivation is underused for various reasons. First, for readers unfamiliar with the technicalities, proving correspondences by program transformation provides the same assurance as proving them on paper. Formal verification must be brought into the process. This brings us to the second related criticism: the lack of tools. Inter-derivation may become more popular when techniques and folklore are collected, their requirements for program verification formally studied, and a tool developed, preferably integrated within a popular freely-available tool framework. An often suggested possibility is a Coq library for inter-derivation.

Reusability in the form of parametricity and modularity will be an important requirement for this endeavour, in particular, the support for derivation of parametric semantic artefacts. We think two important ingredients in this regard are *hybrid* calculi (García-Pérez & Nogueira, 2013; García-Pérez *et al.*, 2013), and two-layer continuation-passing style (Danvy *et al.*, 2013). On the one hand hybrid calculi depend on subsidiary sub-calculi, which ought to be turned into a parameter. On the other hand two-layer continuation-passing style can be used to separate the hybrid and subsidiary continuation spaces and help parametrise on the subsidiary. In this paper we showcase the marriage of hybrid semantics and two-layer CPS.

Our case study is the recently popular gradually-typed lambda calculus (Garcia, 2013; Siek & Taha, 2006; Siek *et al.*, 2009; Siek & Garcia, 2012). Gradual typing is about giving programmers the freedom to move from dynamic typing to static typing by letting them add type annotations gradually to their programs. The gradually-typed lambda calculus $\lambda^{?}_{\rightarrow}$ is a simply-typed lambda calculus with a dynamic type `Dyn` that is assigned by the type system to expressions whose type is statically unknown. The expressions of $\lambda^{?}_{\rightarrow}$ are translated to the expressions of an intermediate language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ with explicit casts that carry blame labels. A cast failure delivers a blame label that indicates the location of the failing cast. The dynamic semantics of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ depends on two design decisions (lazy or eager error-detection, downcast or upcast-downcast blame-tracking) which give rise to a design space with four different points: eager-downcast (**ED**), eager-upcast-downcast (**EUD**), lazy-downcast (**LD**), and lazy-upcast-downcast (**LUD**). These points are captured by different coercion sub-calculi.

In (Siek & Garcia, 2012) we find several implemented denotational semantics (definitional interpreters using meta-level functions) that illustrate the implementation of the variants of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. The small-step reduction semantics are defined mathematically (Siek *et al.*, 2009; Siek & Garcia, 2012) and the correspondences with the denotational semantics are left as conjectures. We prove and generalise the conjectures using program derivation, parametrising the $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ artefact on the coercion artefact to permit derivations for the whole design space. The inter-derivation diagram of Figure 7.1 describes the derivation of the semantic artefacts in the paper. Here is our detailed list of **contributions**:

- We present a coercion-based version of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ (Section 7.2) and an eager-downcast coercion calculus **ED** (Section 7.3). These calculi unify and slightly amend and emend the versions in (Siek & Garcia, 2012; Siek *et al.*, 2009) so as to have an

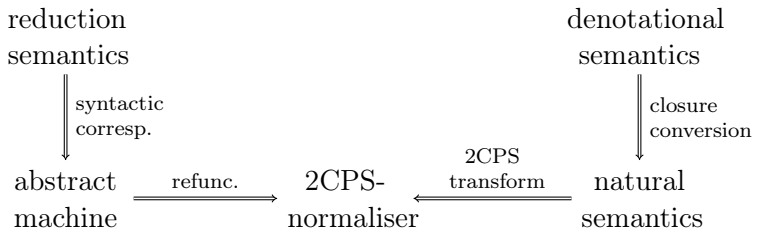


Figure 7.1: Inter-derivation diagram

implementable reduction semantics satisfying unique-decomposition (Felleisen, 1987).

- In Section 7.4 we translate the definitional interpreter (Siek & Garcia, 2012) to ML and derive an instantiation for **ED** dynamic semantics (eager-downcast, named $\mathbf{L} \cup \mathbf{D} \cup \mathbf{E}$ in (Siek *et al.*, 2009)) which is the more appealing for 'it provides thorough error detection and intuitive blame assignment' (Siek *et al.*, 2009, p.13). In Section 7.5 we disentangle translation and normalisation to obtain a purely coercion-based interpreter for expressions that uses a self-contained subsidiary coercion interpreter. In Section 7.6 we finally produce a 2CPS-normaliser that implements the corresponding big-step natural semantics. These steps belong to the right-hand-side of Figure 7.1.

- We extend $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ to $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$, the simply-typed lambda calculus of closures with explicit casts (Section 7.8) whose implementable reduction semantics is the starting point of the syntactic correspondence (Danvy & Nielsen, 2004; Danvy, 2008b; Danvy *et al.*, 2011) on the left-hand-side of Figure 7.1. We state the theorems generalising the conjectures in (Siek & Garcia, 2012) (Section 7.8.1), and prove them via program transformation by linking the two sides of the diagram at the 2CPS-normaliser (Section 7.10, etc).

- The small-step and big-step artefacts for expressions with coercion casts are parametric on the artefacts for coercions. We have presented a full derivation for **ED** dynamic semantics, but thanks to hybrids and 2CPS the artefacts for coercions can be replaced by other artefacts implementing different dynamic semantics. This technique provides the basis for modular derivations of any hybrid semantics, not limited to the definitional interpreters of the gradually-typed lambda calculus.

This paper makes contributions for 'program-derivationists' as well as for 'gradual-type-theorists'. To celebrate the union of the two lines of research, we have summarised in the main sections the important points for each readership, so they can understand the contributions at a glance. In particular, the program-derivationist need not know all details of, and our contributions to, $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. The gradual-type-theorist will find the semantic artefacts in the paper written in traditional mathematical notation. We strongly encourage the

program-derivationist to read the derivation parts of the paper alongside the code,[1] which is the main star of the film. The code is written in Standard ML and organised around the sectioning structure of the paper. Thus, 'Section 1.2' refers to a section of the paper whereas 'Code 3.1' refers to a section of the code.

7.2 $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ with implementable reduction semantics

Figure 7.2 shows the syntax, contraction rules, and implementable reduction semantics of a coercion-based version of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. The foremost point for the program-derivationist is the boxed rule StepCst specifying that the contraction of a cast $\langle c\rangle s$ that applies a coercion c to a simple value s depends on the single-step reduction of c to c' in a coercion sub-calculus $\mathbf{X}$. Thus, $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ is a *hybrid* calculus whose syntax, contraction rules and, by extension, reduction semantics, depend on a *subsidiary* coercion calculus. Fortunately, the dependency on the syntax is not such: coercion expressions are the same across coercion calculi, and what varies is the syntax of 'normal coercions' (normalised coercion expressions) which naturally depends on the reduction semantics. However, the syntax of normal coercions always includes the ones in the contraction rules of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. Such rules have to be part of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ for reasons explained below. Thus, the real dependency is on '$\longmapsto_{\mathbf{X}}$', the reduction semantics for coercions. Observe that $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ admits type cast expressions $\langle S \Leftarrow T\rangle^{\ell} e$ which are present in (Siek & Garcia, 2012, p.2) and in (Siek *et al.*, 2009, Fig.1). In particular, the interpreters in (Siek & Garcia, 2012) work with them. However, type casts are translated off to coercion casts $\langle c\rangle e$. The translation function depends on the reduction semantics for coercions.

A point of interest to the program-derivationist and to the gradual-type-theorist is that in Figure 7.2 blames are expressions, and reduction lifts blames in any reduction context to a result. Consequently, the figure shows a reduction semantics proper that can be implemented (more details below). Blames are results but not expressions in (Siek & Garcia, 2012; Siek *et al.*, 2009). The other contents of the section explain whence our version of the calculus which is of interest mainly to the gradual-type-theorist.

The syntax of types, constants, and primitive operators is the same as in (Siek & Garcia, 2012) and is unmysterious.[2] The syntax of expressions includes the expressions of (Siek *et al.*, 2009, Fig.7), namely, variables, constants, type-annotated abstractions, expression applications $e\,e$, and coercion casts $\langle c\rangle e$. The unannotated abstractions $\lambda x.e$ of (Siek & Garcia, 2012) can be represented by $\lambda x : \texttt{Dyn}.e$. The syntax of expressions also includes applications of primitive operators to expressions, and conditional expressions, both present in (Siek & Garcia, 2012). Finally, expressions also include $\texttt{Blame}\ \ell$ expressions because they

[1] `http://babel.ls.fi.upm.es/~agarcia/papers/Gradual`

[2] For the thorough reader: in (Siek & Garcia, 2012) some expressions carry labels, but in the coercion-based versions only coercions and blames carry labels. Our version is coercion-based and therefore only coercions and blames carry labels, together with type cast expressions which will be later removed.

Syntax: $\boxed{e \in \lambda^{\langle\cdot\rangle}_{\rightarrow}}$

$$
\begin{array}{llcl}
\text{base types} & B & = & \{\texttt{Int}, \texttt{Bool}\} \\
\text{types} & T & ::= & B \mid \texttt{Dyn} \mid T \to T \\
\text{constants} & k & ::= & n \in \mathbb{N} \mid \texttt{t} \mid \texttt{f} \\
\text{operators} & op & ::= & \texttt{inc} \mid \texttt{dec} \mid \texttt{zero?} \\
\text{expressions} & e & ::= & k \mid op\ e \mid \texttt{if}\ e\ e\ e \mid x \mid \lambda x : T.e \mid e\,e \mid \\
 & & & \langle S \Leftarrow T \rangle^{\ell} e \mid \langle c \rangle e \mid \texttt{Blame}\ \ell \\
 & & & \\
\text{simple values} & s & ::= & k \mid \lambda x : T.e \\
\text{values} & v & ::= & s \mid \langle \overline{c} \rangle s \\
\text{results} & r & ::= & v \mid \texttt{Blame}\ \ell
\end{array}
$$

Contraction: $\boxed{e \longrightarrow e}$

$$
\begin{array}{rl}
(\lambda x : T.e)v \longrightarrow [x/v]e & (\beta) \\
op\ n \longrightarrow \delta(op, n) & (\delta) \\
\texttt{if}\ k\ e_1\ e_2 \longrightarrow \begin{cases} e_1 & \text{if } k = \texttt{t} \\ e_2 & \text{if } k = \texttt{f} \end{cases} & (\text{If}) \\
\boxed{\langle c \rangle s \longrightarrow \langle c' \rangle s \quad \text{if } c \longmapsto_{\mathbf{x}} c'\ (\text{StepCst})} & \\
\langle \iota \rangle s \longrightarrow s & (\text{IdCst}) \\
\langle d \rangle \langle \overline{c} \rangle s \longrightarrow \langle \overline{c}\,; d \rangle s & (\text{CmpCst}) \\
\langle \tilde{c} \to \tilde{d} \rangle s\, v \longrightarrow \langle \tilde{d} \rangle (s\, \langle \tilde{c} \rangle v) & (\text{AppCst}) \\
\langle \texttt{Fail}^{\ell} \rangle s \longrightarrow \texttt{Blame}\ \ell & (\text{FailCst}) \\
\langle (\tilde{c} \to \tilde{d})\,; \texttt{Fail}^{\ell} \rangle s \longrightarrow \texttt{Blame}\ \ell & (\text{FailFC})
\end{array}
$$

Reduction semantics: $\boxed{e \longmapsto e}$

$$
\mathbf{E}[\,] \quad ::= \quad [\,] \mid op\ (\mathbf{E}[\,]) \mid \texttt{if}\ (\mathbf{E}[\,])\ e_1\ e_2 \mid (\mathbf{E}[\,])\,e \mid v\,(\mathbf{E}[\,]) \mid \langle c \rangle (\mathbf{E}[\,])
$$

$$
\frac{e \longrightarrow e'}{\mathbf{E}[e] \longmapsto \mathbf{E}[e']} \qquad \frac{}{\mathbf{E}[\texttt{Blame}\ \ell] \longmapsto \texttt{Blame}\ \ell}
$$

Figure 7.2: Syntax, contraction rules, and implementable reduction semantics of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$

can be the result of a contraction, and must thus be a particular kind of expression. The type system for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ can be found in Figure 7.3.

Now to operational semantics. Results r are now expressions: either values v or blames. Values v are simple values s or a coercion expression that applies a wrapper coercion $\overline{c}$

δ-rules $\boxed{\delta(op, n) = k}$

$$\begin{array}{rcl} \delta(\texttt{inc}, n) & = & n + 1 \\ \delta(\texttt{dec}, n) & = & n - 1 \\ \delta(\texttt{zero?}, 0) & = & \texttt{t} \\ \delta(\texttt{zero?}, n) & = & \texttt{f} \quad (n \neq 0) \end{array}$$

Type system for expressions of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ $\boxed{\Gamma \vdash e : T}$

$$\frac{}{\Gamma \vdash k : \mathit{typeC}(k)} \qquad \frac{}{\Gamma \vdash \mathit{op} : \mathit{typeO}(\mathit{op})}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if}\ e_1\ e_2\ e_3 : T} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x \mapsto T \vdash e : S}{\Gamma \vdash (\lambda x : T.e) : T \rightarrow S} \qquad \frac{\Gamma \vdash e_1 : T \rightarrow S \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\, e_2 : S}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \langle S \Leftarrow T \rangle^{\ell} e : S} \qquad \frac{\vdash c : S \Leftarrow T \quad \Gamma \vdash e : T}{\Gamma \vdash \langle c \rangle e : S} \qquad \frac{}{\Gamma \vdash \texttt{Blame}\ \ell : T}$$

Type of constants: $\boxed{\mathit{typeC}(k) = B}$

$$\begin{array}{rcl} \mathit{typeC}(n) & = & \texttt{Int} \\ \mathit{typeC}(\texttt{t}) & = & \texttt{Bool} \\ \mathit{typeC}(\texttt{f}) & = & \texttt{Bool} \end{array}$$

Type of operators: $\boxed{\mathit{typeO}(\mathit{op}) = B \rightarrow B}$

$$\begin{array}{rcl} \mathit{typeO}(\texttt{inc}) & = & \texttt{Int} \rightarrow \texttt{Int} \\ \mathit{typeO}(\texttt{dec}) & = & \texttt{Int} \rightarrow \texttt{Int} \\ \mathit{typeO}(\texttt{zero?}) & = & \texttt{Int} \rightarrow \texttt{Bool} \end{array}$$

Figure 7.3: Complements of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ in Figure 7.2

to a simple value. Wrapper coercions are a subset of normal coercions $\hat{c}$, those that may be applied to simple values. A definition of wrapper and normal coercion for the eager coercion calculus with downcast blame-tracking is given in Section 7.3. In the definitional interpreters of (Siek & Garcia, 2012), simple values include meta-level functions because the interpreters implement denotational semantics (Section 7.4).

The contraction rules are straightforward. The first three specify the contraction of β-,

δ-, and conditional redices. (Function δ can be found in Figure 7.3.) Rule StepCst has already been discussed. The remaining rules, except CmpCst, deal with casts containing normal coercions. Rule CmpCst contracts nested casts (expressions of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$) to coercion sequences. Rule IdCst contracts a cast with an identity coercion. Rule AppCst contracts the application of an arrow coercion $\langle \tilde{c} \rightarrow \tilde{d} \rangle$ to a simple value s (an abstraction if well-typed) when the application is in turn applied to an operand v. The cast is performed against the operand and the result of the application. This is the standard solution for higher-order casts (arrow coercions) (Siek *et al.*, 2009). Rules FailCst and FailFC contract fail coercions to blames. Rule FailFC is given in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ to preserve confluence in the coercion calculus (Section 7.3).[3]

The reduction semantics at the bottom of the figure consists of reduction contexts and single-step contraction rules for redices within context holes. Observe that blames are short-circuited to results by lifting a blame in an arbitrary reduction context to the top level. The reduction contexts specify call-by-value reduction. The reduction of casts would consist of reducing the expression to a simple value, then reducing the coercion within the subsidiary coercion calculus, and the appropriate contraction rule in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ would take it from there.

7.3 The ED coercion calculus

Figure 7.4 shows the syntax, contraction rules, and implementable reduction semantics of **ED**, our version of the eager coercion calculus with downcast blame-tracking, which unifies and slightly amends and emends the versions in (Siek & Garcia, 2012; Siek *et al.*, 2009). The foremost point for both the program-derivationist and the gradual-type-theorist is that the reduction semantics in the figure satisfies the unique-decomposition property (Felleisen, 1987) required for implementation, *i.e.*, every coercion expression is uniquely-decomposable into a coercion reduction context with a redex within the hole. The reduction semantics in (Siek *et al.*, 2009, Fig.4) does not have that property (more details below), and (Siek & Garcia, 2012) is about big-step definitional interpreters so, naturally, it is unconcerned with reduction semantics. The rest of the discussion about our version of the calculus is of interest mainly to the gradual-type-theorist.

First, we discuss coercion expressions. Injectable types to `Dyn` are those types other than `Dyn` because injecting or projecting `Dyn` to `Dyn` is equivalent to the identity coercion. Coercion expressions consist of the identity coercion ι, an injection $I!$ of an injectable type I to `Dyn`, a projection from the dynamic type $I?^{\ell}$ to injectable type I decorated with a blame label ℓ, arrow coercions $c \rightarrow d$, sequences $c\,;d$ (diagrammatic composition) and fail coercions $\texttt{Fail}^{\ell}$ decorated with a blame label. So far, no differences with (Siek & Garcia,

[3]For the thorough reader: in (Siek & Garcia, 2012) rule FailFC is implemented in different places for different semantics. In the lazy artefacts the rule is implemented by `seq_lazy`, and in the eager artefacts by `mk_cast_eager`.

Syntax: $\boxed{c \in \mathbf{ED}}$

$$
\begin{array}{lrcl}
\text{injectable types} & I & ::= & B \mid T \to T \\
\text{coercions} & c, d & ::= & \iota \mid I! \mid I?^{\ell} \mid c \to c \mid c\,;c \mid \mathtt{Fail}^{\ell} \\
\\
\text{wrappers} & \overline{c} & ::= & \tilde{c} \quad \text{where} \quad \tilde{c} \neq (\tilde{c}_1 \to \tilde{c}_2\,;\mathtt{Fail}^{\ell}) \text{ and } \tilde{c} \neq \iota \\
\text{normal parts} & \tilde{c} & ::= & \hat{c} \quad \text{where} \quad \hat{c} \neq \mathtt{Fail}^{\ell} \\
\text{normal coercions} & \hat{c} & ::= & c \quad \text{if} \quad \nexists c'.c \longmapsto_{\mathbf{ED}} c'
\end{array}
$$

Contraction: $\boxed{c \longrightarrow_{\mathbf{ED}} c}$

$$
\begin{array}{rr}
I_1!\,;I_2?^{\ell} \longrightarrow_{\mathbf{ED}} \langle\!\langle I_2 \Leftarrow I_1 \rangle\!\rangle^{\ell} & (\textsc{InOut}) \\
(\tilde{c}_1 \to \tilde{c}_2)\,;(\tilde{d}_1 \to \tilde{d}_2) \longrightarrow_{\mathbf{ED}} ((\tilde{d}_1\,;\tilde{c}_1) \to (\tilde{c}_2\,;\tilde{d}_2)) & (\textsc{Arr}) \\
\iota\,;\hat{c} \longrightarrow_{\mathbf{ED}} \hat{c} & (\textsc{IdL}) \\
\hat{c}\,;\iota \longrightarrow_{\mathbf{ED}} \hat{c} & (\textsc{IdR}) \\
\mathtt{Fail}^{\ell}\,;\hat{c} \longrightarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell} & (\textsc{FailCo}) \\
I!\,;\mathtt{Fail}^{\ell} \longrightarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell} & (\textsc{InFail}) \\
(\mathtt{Fail}^{\ell} \to \hat{d}) \longrightarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell} & (\textsc{FailL}) \\
(\tilde{c} \to \mathtt{Fail}^{\ell}) \longrightarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell} & (\textsc{FailR})
\end{array}
$$

Reduction semantics: $\boxed{c \longmapsto_{\mathbf{ED}} c}$

$$
\begin{array}{rcll}
\mathbf{C}_c[\,] & ::= & [\,] & \text{if } c \text{ is a redex} \\
& \mid & \mathbf{C}_{c_1}[\,]\,;c_2 & \text{if } c \equiv c_1\,;c_2 \\
& \mid & \hat{c}_1\,;\mathbf{C}_{c_2}[\,] & \text{if } c \equiv \hat{c}_1\,;c_2 \\
& \mid & \mathbf{C}_{c_1}[\,] \to c_2 & \text{if } c \equiv c_1 \to c_2 \\
& \mid & \hat{c}_1 \to \mathbf{C}_{c_2}[\,] & \text{if } c \equiv \hat{c}_1 \to c_2 \\
& \mid & \hat{c}_{11}\,;\mathbf{C}_{(\hat{c}_{12}\,;\hat{c}_2)}[\,] & \text{if } c \equiv (\hat{c}_{11}\,;\hat{c}_{12})\,;\hat{c}_2 \\
& \mid & \mathbf{C}_{(\hat{c}_1\,;\hat{c}_{12})}[\,]\,;\hat{c}_{22} & \text{if } c \equiv \hat{c}_1\,;(\hat{c}_{21}\,;\hat{c}_{22})
\end{array}
$$

$$
\frac{c \longrightarrow_{\mathbf{ED}} c'}{c_1 \equiv \mathbf{C}_{c_1}[c] \longmapsto_{\mathbf{ED}} \mathbf{C}_{c_2}[c'] \equiv c_2}
$$

Figure 7.4: Syntax, contraction rules, and reduction semantics of **ED**

2012; Siek *et al.*, 2009) other than notational. The type system of **ED** can be found in Figure 7.5.

The differences arise in the reduction semantics. The reduction semantics at the bottom of Figure 7.4 satisfies unique-decomposition and is therefore implementable. The one in (Siek *et al.*, 2009, Fig.4) is defined modulo a congruence on sequences $(c_1\,;c_2)\,;c_3 \cong$

Translation function for casts: $\boxed{\langle\!\langle T \Leftarrow T \rangle\!\rangle^{\ell} = \hat{c}}$

$$
\begin{array}{rcl}
\langle\!\langle B \Leftarrow B \rangle\!\rangle^{\ell} & = & \iota \\
\langle\!\langle B_2 \Leftarrow B_1 \rangle\!\rangle^{\ell} & = & \mathtt{Fail}^{\ell} \qquad \text{if } B_1 \neq B_2 \\
\langle\!\langle \mathtt{Dyn} \Leftarrow \mathtt{Dyn} \rangle\!\rangle^{\ell} & = & \iota \\
\langle\!\langle \mathtt{Dyn} \Leftarrow B \rangle\!\rangle^{\ell} & = & B! \\
\langle\!\langle B \Leftarrow \mathtt{Dyn} \rangle\!\rangle^{\ell} & = & B?^{\ell} \\
\langle\!\langle T_1 \to T_2 \Leftarrow B \rangle\!\rangle^{\ell} & = & \mathtt{Fail}^{\ell} \\
\langle\!\langle B \Leftarrow S_1 \to S_2 \rangle\!\rangle^{\ell} & = & \mathtt{Fail}^{\ell} \\
\langle\!\langle T_1 \to T_2 \Leftarrow S_1 \to S_2 \rangle\!\rangle^{\ell} & = & \mathit{mkArr}(\langle\!\langle S_1 \Leftarrow T_1 \rangle\!\rangle^{\ell}, \langle\!\langle T_2 \Leftarrow S_2 \rangle\!\rangle^{\ell}) \\
\langle\!\langle \mathtt{Dyn} \Leftarrow S_1 \to S_2 \rangle\!\rangle^{\ell} & = & S_1 \to S_2! \\
\langle\!\langle T_1 \to T_2 \Leftarrow \mathtt{Dyn} \rangle\!\rangle^{\ell} & = & T_1 \to T_2?^{\ell}
\end{array}
$$

Arrow combinator: $\boxed{\mathit{mkArr}(\hat{c}, \hat{c}) = \hat{c}}$

$$
\begin{array}{rcll}
\mathit{mkArr}(\mathtt{Fail}^{\ell}, \hat{c}_2) & = & \mathtt{Fail}^{\ell} & \\
\mathit{mkArr}(\hat{c}_1, \mathtt{Fail}^{\ell}) & = & \mathtt{Fail}^{\ell} & \\
\mathit{mkArr}(\hat{c}_1, \hat{c}_2) & = & \hat{c}_1 \to \hat{c}_2 & \text{otherwise}
\end{array}
$$

Type system for coercions; $\boxed{\vdash c : T \Leftarrow T}$

$$
\frac{}{\vdash \iota : T \Leftarrow T} \qquad \frac{}{\vdash T! : \mathtt{Dyn} \Leftarrow T} \qquad \frac{}{\vdash T?^{\ell} : T \Leftarrow \mathtt{Dyn}} \qquad \frac{}{\vdash \mathtt{Fail}^{\ell} : T \Leftarrow S}
$$

$$
\frac{\vdash c : S_1 \Leftarrow T_1 \quad \vdash d : T_2 \Leftarrow S_2}{\vdash (c \to d) : (T_1 \to T_2) \Leftarrow (S_1 \to S_2)} \qquad \frac{\vdash c : T_1 \Leftarrow T_2 \quad \vdash d : T_2 \Leftarrow T_3}{\vdash (c\,;d) : T_1 \Leftarrow T_3}
$$

Figure 7.5: Complements of **ED** in Figure 7.4

$c_1\,;(c_2\,;c_3)$ that permits the definition of simpler reduction contexts:

$$\mathbf{C}[\,] \;::=\; [\,] \mid \mathbf{C}[\,]\,;c \mid \hat{c}\,;\mathbf{C}[\,] \mid \mathbf{C}[\,] \to c \mid \tilde{c} \to \mathbf{C}[\,]$$

$$\frac{c \cong \mathbf{C}[c_1] \quad c_1 \longrightarrow_{\mathbf{ED}} c_2 \quad \mathbf{C}[c_2] \cong c'}{c \longmapsto_{\mathbf{ED}} c'}$$

To resolve the ambiguity introduced by congruence we fix the association by defining reduction contexts $\mathbf{C}_c[\,]$ which are indexed by the input coercion c, so that decomposition is guided by the shape of c. (Note the difference between syntactic identity '$\equiv$' in Figure 7.4 and congruence '$\cong$' in the original reduction semantics.) The productions of the generative grammar for reduction contexts have precedence and are intended to be 'short circuited', *i.e.*, the the sixth production is 'tried' first and, if $\hat{c}_{11}\,;\hat{c}_{12}$ does not contain any redex—and

hence the decomposition function does not come to a decomposition for $\hat{c}_1\,;\hat{c}_2$—, then the seventh production is tried. Unique-decomposition is proven by structural induction on c.

The difficulties in implementing a deterministic semantics for coercions have already been acknowledged by Garcia (Garcia, 2013). He introduces a composition-free representation for coercions, named *supercoercions*, and a new set of contraction rules. His approach solves the challenge of being 'complete in the face of inert compositions and associativity' (Garcia, 2013), superseding the 'ad hoc reassociation scheme' in (Siek & Garcia, 2012). Here we stick to the ad hoc scheme, since we aim to prove the conjectures relative to (Siek & Garcia, 2012).

The rest of this section is addressed to the gradual-type-theorist. Recall from Section 7.2 that wrapper coercions $\overline{c}$ are normal coercions $\hat{c}$ that may be applied to simple values. Normal coercions are those that cannot be further reduced, and normal parts are coercions other than the fail coercion. The notion of wrapper is induced by the treatment of ι, $\mathtt{Fail}^{\ell}$, and $\tilde{c} \to \tilde{d}\,;\mathtt{Fail}^{\ell}$. Wrappers can only coerce constants and abstractions. We add the required side-condition $\tilde{c} \neq (\tilde{c} \to \tilde{d}\,;\mathtt{Fail}^{\ell})$ to wrappers which is missing in (Siek *et al.*, 2009, Fig.7). The side-condition is required because the contraction rule for $\langle(\tilde{c} \to \tilde{d}\,;\mathtt{Fail}^{\ell})\rangle s$ in Figure 7.2 delivers a blame. In (Siek & Garcia, 2012) an extensional definition of normal coercions is provided that rules out the ill-typed ones according to the type system of $\lambda^{\langle\cdot\rangle}_{\to}$ and **ED**.

The contraction rule InOut coalesces an injection followed by a projection using a translation function $\langle\!\langle I_2 \Leftarrow I_1 \rangle\!\rangle^{\ell}$ (Figure 7.5). This function translates a type cast to a normal coercion. If the projection is illegal the translation function delivers a fail coercion decorated with the projection's blame label. The contraction rules Arr to FailR are those in (Siek & Garcia, 2012, Sec.6.1) but with normal coercions $\hat{c}$ substituted for arbitrary coercions c in order to have a reduction semantics with unique-decomposition and preserve confluence: arbitrarily long sequences ending in $\mathtt{Fail}^{\ell}$ must be allowed to fail due to previous coercions early in the sequence. (This is also the reason why FailFC is specified in $\lambda^{\langle\cdot\rangle}_{\to}$ and not in **ED**, see (Siek *et al.*, 2009) for details.)

Like (Siek & Garcia, 2012) but unlike (Siek *et al.*, 2009) we omit rule $\iota \to \iota \longrightarrow \iota$ because it is superfluous: according to Arr, IdL, and IdR, sequencing the $\iota \to \iota$ arrow to any other arrow has the same effect as sequencing the identity ι.

7.4 Interpretations of the gradually-typed lambda calculus

In (Siek & Garcia, 2012) a definitional interpreter `interp` for a type-cast-based $\lambda^{\langle\cdot\rangle}_{\to}$ is given that is parametric on functions `cast` and `apply`. The choice and name of parameters do much more than reflect the dependency on the coercion sub-calculus **X**. Among other things they also accommodate the translation to coercion casts, and apportion the implementation of contraction rules. Broadly, the `cast` parameter is instantiated to `apply_cast_`**X** which given a type cast and a value, it first invokes the translation

Syntax:

environments	ρ	::=	$\epsilon \mid (x \mapsto v) : \rho$
procedures	$proc \in \mathsf{V} \to \mathsf{R}$	::=	`fn` v `=>` r
simple values	s	::=	$k \mid proc$
values	$v \in \mathsf{V}$	::=	$s \mid \langle \overline{c} \rangle s$
results	$r \in \mathsf{R}$	::=	$v \mid$ `Blame` ℓ

Figure 7.6: Environments and values for the interpreter in (Siek & Garcia, 2012).

function `mk_coerce_X` to the type cast to obtain a *normal* coercion, and then invokes `apply_coercion_X` that applies that normal coercion to the value. The `apply` parameter is instantiated to `apply_X` which, broadly, realises rule AppCst in Figure 7.2. The `cast` parameter realises the other rules in the figure dealing with coercions.

The definitional interpreter is environment-based and implements a denotational semantics. It delivers meta-level-function results, nicknamed 'procedures' in (Siek & Garcia, 2012). Figure 7.6 defines in mathematical notation the environments and the hierarchy of results used by the definitional interpreter. An environment is a colon-separated list of bindings $x \mapsto v$, where x is a variable and v is a value. Values are either simple values or coercion expressions applying a wrapper over a simple value. A simple value is either a constant or a procedure `fn` v `=>` r that takes a value and returns a result that depends on this value. Finally, a result is a value or a blame label.

7.4.1 Translating the original interpreter to ML

We have translated to Standard ML the original definitional interpreter in (Siek & Garcia, 2012) which is written in Scheme. We are more comfortable with ML, which is used in many papers in program derivation. The ML translation can be found in Code 1.1. In the code, we use a nameless representation with de Bruijn indices, but we keep the traditional nameful representation in the mathematical notation. For readability, and to avoid the (un)packing of data constructors, we have embedded the whole hierarchy of normal coercions in one datatype `coercion`. For the hierarchy of values and results we use the mutually dependent datatypes `value` and `result`, with procedures represented by clause `VPROC of value -> result`.

In the ML translation, we have replaced the monadic macro for let-expressions `letB` in (Siek & Garcia, 2012) by case expressions that short-circuit the blames to results.

7.4.2 Instantiating the definitional interpreter

We have to 'instantiate' the parametric definitional interpreter to a particular dynamic semantics in order to establish the correspondence with an implementation of a reduction semantics that allegedly realises exactly that dynamic semantics. We choose the

Look-up function: $\boxed{\rho\,!\,x = v}$

$$\rho\,!\,x = v \quad \text{where } (x \mapsto v) \text{ contains the first occurrence of } x \text{ in } \rho$$

Cast combinator $\boxed{mkCast(\hat{c}, s) = r}$

$$\begin{array}{rcl} mkCast(\iota, s) & = & s \\ mkCast(\texttt{Fail}^{\ell}, s) & = & \texttt{Blame}\ \ell \\ mkCast((\tilde{c} \to \tilde{d}\,;\texttt{Fail}^{\ell}), s) & = & \texttt{Blame}\ \ell \\ mkCast(\bar{c}, s) & = & \langle \bar{c} \rangle s \end{array}$$

Figure 7.7: Auxiliary functions for the denotational semantics in Figure 7.9

ED semantics (Section 7.1). To obtain an instantiation we inline the function calls and produce functions `mk_arrow`, `translate_cast`, `compose_coercion`, `mk_cast`, `apply_coercion`, `apply_cast`,`apply`, and `eval` (our name for `interp_`**ED**) all found in Code 1.2. Function `compose_coercion` corresponds to `seq_eager` in (Siek & Garcia, 2012). Function `translate_cast` is the instantiation of `mk_coerce_d` with parameter `mk_arrow_eager`. Function `translate_cast` implements $\langle\!\langle S \Leftarrow T \rangle\!\rangle^{\ell}$ defined in Figure 7.5.

Figure 7.9 shows in mathematical notation the denotational semantics implemented by Code 1.2. We have written $e[\rho] =_{\langle\cdot\rangle} r$ for the evaluation of expression e in an environment ρ with result r. The notation $e[\rho]$ stands for an unfolded representation of a closure. This denotational semantics is the one on the top right corner of the inter-derivation diagram (Figure 7.1). The mathematical semantics can be checked by both the gradual-type-theorist and the program-derivationist against the Scheme or ML versions of the instantiated definitional interpreter. Function `eval` in Code 1.2 is specified by the evaluation section of the figure, `apply` by the application section, `apply_cast` by the cast application section, `apply_coercion` by the coercion application section, and `compose_coercion` by the coercion composition section.

The evaluation rule EvProc specifies the evaluation of an abstraction in an environment. The evaluation results in a procedure `fn` v `=>` r. The value v is the one passed to the procedure in rule ApProc.

7.4.3 The correctness conjectures

In (Siek & Garcia, 2012) several correspondences between some instantiations of the definitional interpreter and the relevant reduction semantics are conjectured, which we quote:

Conjecture 1. *If the unique cast labelled with ℓ in program e respects subtyping,*[4] *then* `eval_ld(`e`)` $\neq$ `Blame` ℓ.

[4]Here 'subtyping' means that all the casts (or coercions) are safe at static time, see (Siek *et al.*, 2009).

Coercion composition: $\boxed{\hat{c}\,;\hat{c} \Downarrow^{co}_{\mathbf{ED}} \hat{c}}$

$$\frac{\hat{c}_1\,;\hat{c}_2 \in \mathrm{NC}}{\hat{c}_1\,;\hat{c}_2 \Downarrow^{co}_{\mathbf{ED}} \hat{c}_1\,;\hat{c}_2}\ (\textsc{ComNorm}) \qquad \frac{}{I!\,;J?^{\ell} \Downarrow^{co}_{\mathbf{ED}} \langle\!\langle J \Leftarrow I \rangle\!\rangle^{\ell}}\ (\textsc{ComInOut}) \qquad \frac{}{\iota\,;\hat{c} \Downarrow^{co}_{\mathbf{ED}} \hat{c}}\ (\textsc{ComIdL}) \qquad \frac{}{\hat{c}\,;\iota \Downarrow^{co}_{\mathbf{ED}} \hat{c}}\ (\textsc{ComIdR})$$

$$\frac{\tilde{c}_{21}\,;\tilde{c}_{11} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_3 \qquad \tilde{c}_{22}\,;\tilde{c}_{12} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}{(\tilde{c}_{11} \to \tilde{c}_{12})\,;(\tilde{c}_{21} \to \tilde{c}_{22}) \Downarrow^{co}_{\mathbf{ED}} mkArr(\hat{c}_3, \hat{c}_4)}\ (\textsc{ComArr}) \qquad \frac{\hat{c}_{12}\,;\hat{c}_2 \Downarrow^{co}_{\mathbf{ED}} \hat{c}_3 \qquad \hat{c}_{11}\,;\hat{c}_3 \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}{(\hat{c}_{11}\,;\hat{c}_{12})\,;\hat{c}_2 \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}\ (\textsc{ComAssL})$$

$$\frac{\hat{c}_1\,;\hat{c}_{21} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_3 \qquad \hat{c}_3\,;\hat{c}_{22} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}{\hat{c}_1\,;(\hat{c}_{21}\,;\hat{c}_{22}) \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}\ (\textsc{ComAssR}) \qquad \frac{}{\mathtt{Fail}^{\ell}\,;\hat{c} \Downarrow^{co}_{\mathbf{ED}} \mathtt{Fail}^{\ell}}\ (\textsc{ComFailL}) \qquad \frac{}{I!\,;\mathtt{Fail}^{\ell} \Downarrow^{co}_{\mathbf{ED}} \mathtt{Fail}^{\ell}}\ (\textsc{ComFailR})$$

Coercion application: $\boxed{\langle \hat{c} \rangle v \Downarrow^{co}_{cr} r}$ **Cast application:** $\boxed{\langle S \Leftarrow T \rangle^{\ell} v \Downarrow^{co}_{cs} r}$

$$\frac{\overline{c}\,;\hat{d} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_1}{\langle \hat{d} \rangle(\langle \overline{c} \rangle s) \Downarrow^{co}_{cr} mkCast(\hat{c}_1, s)}\ (\textsc{CoeComp}) \qquad \frac{}{\langle \hat{c} \rangle s \Downarrow^{co}_{cr} mkCast(\hat{c}, s)}\ (\textsc{CoeNorm}) \qquad \frac{\langle\ \langle\!\langle S \Leftarrow T \rangle\!\rangle^{\ell}\ \rangle v \Downarrow^{co}_{cr} r}{\langle S \Leftarrow T \rangle^{\ell} v \Downarrow^{co}_{cs} r}\ (\textsc{Cst})$$

Application: $\boxed{v\, v \Downarrow^{co}_{ap} r}$

$$\frac{}{proc\ v \Downarrow^{co}_{ap} proc(v)}\ (\textsc{ApProc}) \qquad \frac{\langle \tilde{c} \rangle v \Downarrow^{co}_{cr} v_1 \qquad proc\ v_1 \Downarrow^{co}_{ap} v_2 \qquad \langle \tilde{d} \rangle v_2 \Downarrow^{co}_{cr} r}{(\langle \tilde{c} \to \tilde{d} \rangle proc) v \Downarrow^{co}_{ap} r}\ (\textsc{ApArr})$$

$$\frac{\langle \tilde{c} \rangle v \Downarrow^{co}_{cr} \mathtt{Blame}\ \ell}{(\langle \tilde{c} \to \tilde{d} \rangle proc) v \Downarrow^{co}_{ap} \mathtt{Blame}\ \ell}\ (\textsc{ApArrBL}) \qquad \frac{\langle \tilde{c} \rangle v \Downarrow^{co}_{cr} v_1 \qquad proc\ v_1 \Downarrow^{co}_{ap} \mathtt{Blame}\ \ell}{(\langle \tilde{c} \to \tilde{d} \rangle proc) v \Downarrow^{co}_{ap} \mathtt{Blame}\ \ell}\ (\textsc{ApArrBR})$$

Figure 7.8: Coercion composition and application for the denotational semantics in Figure 7.9. Auxiliary functions $\langle\!\langle S \Leftarrow T \rangle\!\rangle^{\ell}$ and *mkArr* are defined in Figure 7.5, and function *mkCast* is defined in Figure 7.7.

Evaluation: $\boxed{e[\rho] =_{\langle\cdot\rangle} r}$

$$\frac{}{k[\rho] =_{\langle\cdot\rangle} k}\ (\textsc{EvConst}) \qquad \frac{e[\rho] =_{\langle\cdot\rangle} k}{(op\ e)[\rho] =_{\langle\cdot\rangle} \delta(op, k)}\ (\textsc{EvOper}) \qquad \frac{e[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(op\ e)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvOperB})$$

$$\frac{e_1[\rho] =_{\langle\cdot\rangle} \texttt{t} \quad e_2[\rho] =_{\langle\cdot\rangle} r}{(\texttt{if}\ e_1\ e_2\ e_3)[\rho] =_{\langle\cdot\rangle} r}\ (\textsc{EvIfL}) \qquad \frac{e_1[\rho] =_{\langle\cdot\rangle} \texttt{f} \quad e_3[\rho] =_{\langle\cdot\rangle} r}{(\texttt{if}\ e_1\ e_2\ e_3)[\rho] =_{\langle\cdot\rangle} r}\ (\textsc{EvIfR}) \qquad \frac{e_1[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(\texttt{if}\ e_1\ e_2\ e_3)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvIfB})$$

$$\frac{e[(x \mapsto v) : \rho] =_{\langle\cdot\rangle} r}{(\lambda x : T.e)[\rho] =_{\langle\cdot\rangle} (\texttt{fn}\ v\ \texttt{=>}\ r)}\ (\textsc{EvProc}) \qquad \frac{e_1[\rho] =_{\langle\cdot\rangle} v_1 \quad e_2[\rho] =_{\langle\cdot\rangle} v_2 \quad v_1\ v_2 \Downarrow^{co}_{ap} v_3}{(e_1\ e_2)[\rho] =_{\langle\cdot\rangle} v_3}\ (\textsc{EvApp})$$

$$\frac{e_1[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(e_1\ e_2)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvAppBL}) \qquad \frac{e_1[\rho] =_{\langle\cdot\rangle} v_1 \quad e_2[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(e_1\ e_2)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvAppBR})$$

$$\frac{\rho\,!\,x =_{\langle\cdot\rangle} r}{x[\rho] =_{\langle\cdot\rangle} r}\ (\textsc{EvVar}) \qquad \frac{e[\rho] =_{\langle\cdot\rangle} v \quad \langle S \Leftarrow T\rangle^{\ell} v \Downarrow^{co}_{cs} r}{(\langle S \Leftarrow T\rangle^{\ell} e)[\rho] =_{\langle\cdot\rangle} r}\ (\textsc{EvCast}) \qquad \frac{e[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(\langle S \Leftarrow T\rangle^{\ell} e)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvCastB})$$

$$\frac{}{(\texttt{Blame}\ \ell)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvBlame})$$

Figure 7.9: Denotational semantics in mathematical notation implemented by equivalent definitional interpreters `interp_ed` ((Siek & Garcia, 2012) and Code 1.1) and `eval` (Code 1.2). Auxiliary function $\rho\,!\,x$, is defined in Figure 7.7. Application $\Downarrow^{co}_{ap}$ and cast application $\Downarrow^{co}_{cs}$ are defined in Figure 7.8.

Conjecture 2. *For any well-typed program e,* `eval_ld(`e`)` $= o$ *if and only if* $\langle\!\langle e \rangle\!\rangle \longmapsto^*_{\mathbf{LD}} r$ *and* `observe(`r`)` $= o$.

Conjecture 9.1. *Given two well-typed coercions in normal form, c_1 and c_2, we have* `seq_ed(`c_1`,`c_2`)` $= \hat{c}_3$ *and* $(c_1 \,;\, c_2) \longmapsto^*_{\mathbf{ED}} \hat{c}_3$.

Conjectures 1 and 2 state the correctness of the instantiated interpreter relative to the **LD** semantics. Conjecture 1 states subtyping soundness (by the way, `eval_ld` type-checks expressions and invokes `interp_ld`). Conjecture 2 states the correspondence between the instantiated definitional interpreter and the **LD** reduction semantics. Function `observe` is a reflect function as in normalisation by evaluation: it produces denotational (meta-level) results from results produced by reduction. Expressions with type casts are translated by $\langle\!\langle \cdot \rangle\!\rangle$ to expressions with coercion casts. There are analogous Conjectures 3 and 4 in (Siek & Garcia, 2012) for the **LUD** semantics. However, no conjectures are stated for the **ED** and the **EUD** semantics. Conjecture 9.1 states the correctness relative to the **ED** semantics of the composition of two normal coercions.

As discussed in the introduction, we generalise and prove these conjectures for the **ED** semantics (Section 7.5.2 and 7.8) by inter-deriving the reduction semantics and the instantiated definitional interpreter. The other conjectures can be proven similarly by plugging a different reduction semantics for coercions (Section 7.11.2).

7.5 Prelude: from casts to coercions

The denotational semantics in Figure 7.9 is defined for a type-cast-based $\lambda^{(\cdot)}_{\rightarrow}$. In this section we discuss how we have turned it (actually, its implementation `eval`) into a purely coercion-based semantics by applying two program transformation steps. The bulk of this section will be of interest to the program-derivationist who is advised to read Code 2 alongside this section.

7.5.1 Fissioning evaluator and translation function

Function `apply_cast` (Section 7.4.2, Code 1.2) first translates type casts to coercion casts and then invokes `apply_coercion`. We want to get rid of the translation and obtain a coercion-based normaliser. We inline `apply_cast` in `eval` to get rid of `apply_cast`, and then perform lightweight *fission* by fix point promotion to separate `translate_expression` from `eval1`, the obtained translation-free evaluation function. The fission transformation (a.k.a. trampoline transformation) is the inverse of the *fusion* transformation described in (Ohori & Sasano, 2007). Function `apply_cast` is no longer used. The resulting coercion-based interpreter `eval1` and the `translate_expression` are found in Code 2.1. The latter implements $\langle\!\langle \cdot \rangle\!\rangle$ in Conjecture 2. It fires on type casts, is an identity on variables, constants, and blame expressions, and recursively proceeds over other expressions. Hereafter we can forget about type casts.

7.5.2 Deriving a self-contained coercion normaliser

As a result of the previous inlining, the coercion-based interpreter for expressions `eval1` invokes `compose_coercion`, which implements the natural semantics $\Downarrow^{co}_{\mathbf{ED}}$ in Figure 7.8,[5] to normalise sequences of normal coercions. In order to prove the correspondence with the reduction semantics we need a self-contained coercion normaliser. We have produced such normaliser `normalise_coercion_nor` in Code 2.2.3, which implements the natural semantics $\Downarrow_{\mathbf{ED}}$ shown in Figure 7.10.[6] To obtain this normaliser we first write a coercion normaliser (Code 2.2) that normalises sequences and arrows left-to-right, invoking respectively `compose_coercion` or `mk_arrow` afterwards, and is an identity on the other normal coercions. We replace the calls to `compose_coercion` by recursive calls to the normaliser on a sequence (Code 2.2.1), inline sequencing within normalisation (Code 2.2.2) and defer the normalisation of sequences of arrows by constructing intermediate arrows with sequences (Code 2.2.3). All these steps are equivalence-preserving, and `normalise_coercion_nor` behaves like `compose_coercion` for sequences of normal coercions. Function `eval_nor` in Code 2.2.3 is the instantiated definitional interpreter that employs the self-contained `normalise_coercion_nor`.

7.6 From denotational semantics to 2CPS-normaliser

In this section, we apply closure conversion (Danvy, 2008a) to defunctionalise the meta-level procedures of the definitional interpreter. We obtain a natural semantics (a big-step normaliser) that is the starting point of the functional correspondence (Ager *et al.*, 2003b; Danvy & Millikin, 2009; Danvy *et al.*, 2011) (the derivation of an abstract machine from a natural semantics).

7.6.1 Closure conversion

Closure-conversion consists of defunctionalising the procedures in `eval_nor` (Code 2.2.3) by enumerating the inhabitants of the function space and by introducing a datatype constructor (defunctionalised continuation) for each of the inhabitants. An auxiliary function will apply such constructors to the intermediate results of computation. There is only one inhabitant, namely, the function packed within the `VPROC` constructor, which takes an operand and invokes `eval_nor` on the procedure body, passing an environment enlarged with the operand. We defunctionalise and introduce constructor `VPROC1` in datatype `value_clos` (Code 3.1). The constructor stores the body and the environment of the lambda expression representing the procedure, which together make up a *closure* (Landin, 1964). The auxiliary function that applies the defunctionalised continuation is inlined in function

[5] Figure 7.8 defines a natural semantics with the proviso that rule ComAssL has precedence over rule ComAssR. This precedence is a consequence of the precedence of the productions in the generative grammar of reduction contexts alluded to in Section 7.3.

[6] As before, Figure 7.10 defines a natural semantics with the proviso that rule CoeAssL has precedence over rule CoeAssR.

Coercion normalisation: $\boxed{c \Downarrow_{\mathbf{ED}} \hat{c}}$

$$\frac{}{\hat{c} \Downarrow_{\mathbf{ED}} \hat{c}}\ (\textsc{CoeTriv}) \qquad \frac{c_1 \Downarrow_{\mathbf{ED}} I! \quad c_2 \Downarrow_{\mathbf{ED}} J?^{\ell}}{c_1 \,;\, c_2 \Downarrow_{\mathbf{ED}} \langle\!\langle J \Leftarrow I \rangle\!\rangle^{\ell}}\ (\textsc{CoeInOut}) \qquad \frac{c \Downarrow_{\mathbf{ED}} \hat{c} \quad d \Downarrow_{\mathbf{ED}} \hat{d}}{(c \to d) \Downarrow_{\mathbf{ED}} \mathit{mkArr}(\hat{c}, \hat{d})}\ (\textsc{CoeArr})$$

$$\frac{c_1 \Downarrow_{\mathbf{ED}} \hat{c}_1 \quad c_2 \Downarrow_{\mathbf{ED}} \iota}{c_1 \,;\, c_2 \Downarrow_{\mathbf{ED}} \hat{c}_1}\ (\textsc{CoeIdL}) \qquad \frac{c_1 \Downarrow_{\mathbf{ED}} \iota \quad c_2 \Downarrow_{\mathbf{ED}} \hat{c}_2}{c_1 \,;\, c_2 \Downarrow_{\mathbf{ED}} \hat{c}_2}\ (\textsc{CoeIdR})$$

$$\frac{c_1 \Downarrow_{\mathbf{ED}} (\tilde{c}_{11} \to \tilde{c}_{12}) \quad c_2 \Downarrow_{\mathbf{ED}} (\tilde{c}_{21} \to \tilde{c}_{22}) \quad ((\tilde{c}_{21} \,;\, \tilde{c}_{11}) \to (\tilde{c}_{12} \,;\, \tilde{c}_{22})) \Downarrow_{\mathbf{ED}} \hat{c}_3}{c_1 \,;\, c_2 \Downarrow_{\mathbf{ED}} \hat{c}_3}\ (\textsc{CoeSeqArr})$$

$$\frac{c_1 \Downarrow_{\mathbf{ED}} (\hat{c}_{11} \,;\, \hat{c}_{12}) \quad c_2 \Downarrow_{\mathbf{ED}} \hat{c}_2 \quad \hat{c}_{12} \,;\, \hat{c}_2 \Downarrow_{\mathbf{ED}} \hat{c}_3 \quad \hat{c}_{11} \,;\, \hat{c}_3 \Downarrow_{\mathbf{ED}} \hat{c}_4}{c_1 \,;\, c_2 \Downarrow_{\mathbf{ED}} \hat{c}_4}\ (\textsc{CoeAssL}) \qquad \frac{c_1 \Downarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell} \quad c_2 \Downarrow_{\mathbf{ED}} \hat{c}_2}{c_1 \,;\, c_2 \Downarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell}}\ (\textsc{CoeFailL})$$

$$\frac{c_1 \Downarrow_{\mathbf{ED}} \hat{c}_1 \quad c_2 \Downarrow_{\mathbf{ED}} (\hat{c}_{21} \,;\, \hat{c}_{22}) \quad \hat{c}_1 \,;\, \hat{c}_{21} \Downarrow_{\mathbf{ED}} \hat{c}_3 \quad \hat{c}_3 \,;\, \hat{c}_{22} \Downarrow_{\mathbf{ED}} \hat{c}_4}{c_1 \,;\, c_2 \Downarrow_{\mathbf{ED}} \hat{c}_4}\ (\textsc{CoeAssR}) \qquad \frac{c_1 \Downarrow_{\mathbf{ED}} I! \quad c_2 \Downarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell}}{c_1 \,;\, c_2 \Downarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell}}\ (\textsc{CoeFailR})$$

Coercion application: $\boxed{\langle \hat{c} \rangle v \Downarrow_{cr} r}$

$$\frac{\overline{c} \,;\, \hat{d} \Downarrow_{\mathbf{ED}} \hat{c}_1}{\langle \hat{d} \rangle (\langle \overline{c} \rangle s) \Downarrow_{cr} \mathit{mkCast}(\hat{c}_1, s)}\ (\textsc{NSComp}) \qquad \frac{}{\langle \hat{c} \rangle s \Downarrow_{cr} \mathit{mkCast}(\hat{c}, s)}\ (\textsc{NSNorm})$$

Figure 7.10: Coercion normalisation and coercion application for the natural semantics in Figure 7.11. Auxiliary functions $\langle\!\langle S \Leftarrow T \rangle\!\rangle^{\ell}$ and *mkArr* are defined in Figure 7.5, and function *mkCast* is defined in Figure 7.7.

Closure application: $\boxed{v\, v \Downarrow_{ap} r}$

$$\frac{e[(x \mapsto v) : \rho] \Downarrow r}{(\lambda\!\!\lambda.e[(x : T) : \rho])v \Downarrow_{ap} r}\ (\textsc{NsProc}) \qquad \frac{\langle \tilde{c}\rangle v \Downarrow_{cr} v_1 \quad (\lambda\!\!\lambda.e[(x : T) : \rho])v_1 \Downarrow_{ap} v_2 \quad \langle \tilde{d}\rangle v_2 \Downarrow_{cr} r}{(\langle \tilde{c} \to \tilde{d}\rangle(\lambda\!\!\lambda.e[(x : T) : \rho]))v \Downarrow_{ap} r}\ (\textsc{NsArr})$$

$$\frac{\langle \tilde{c}\rangle v \Downarrow_{cr} \texttt{Blame}\ \ell}{(\langle \tilde{c} \to \tilde{d}\rangle(\lambda\!\!\lambda.\mathsf{cl}))v \Downarrow_{ap} \texttt{Blame}\ \ell}\ (\textsc{NsArrBL}) \qquad \frac{\langle \tilde{c}\rangle v \Downarrow_{cr} v_1 \quad (\lambda\!\!\lambda.\mathsf{cl})\ v_1 \Downarrow_{ap} \texttt{Blame}\ \ell}{(\langle \tilde{c} \to \tilde{d}\rangle(\lambda\!\!\lambda.\mathsf{cl}))v \Downarrow_{ap} \texttt{Blame}\ \ell}\ (\textsc{NsArrBR})$$

Closure normalisation: $\boxed{e[\rho] \Downarrow r}$

$$\frac{}{k[\rho] \Downarrow k}\ (\textsc{NSConst}) \qquad \frac{e[\rho] \Downarrow n}{(op\ e)[\rho] \Downarrow \delta(op, n)}\ (\textsc{NSOper}) \qquad \frac{e[\rho] \Downarrow \texttt{Blame}\ \ell}{(op\ e)[\rho] \Downarrow \texttt{Blame}\ \ell}\ (\textsc{NSOperB})$$

$$\frac{e_1[\rho] \Downarrow \texttt{t} \quad e_2[\rho] \Downarrow r}{(\texttt{if}\ e_1\ e_2\ e_3)[\rho] \Downarrow r}\ (\textsc{NSIfL}) \qquad \frac{e_1[\rho] \Downarrow \texttt{f} \quad e_3[\rho] \Downarrow r}{(\texttt{if}\ e_1\ e_2\ e_3)[\rho] \Downarrow r}\ (\textsc{NSIfR}) \qquad \frac{e_1[\rho] \Downarrow \texttt{Blame}\ \ell}{(\texttt{if}\ e_1\ e_2\ e_3)[\rho] \Downarrow \texttt{Blame}\ \ell}\ (\textsc{NSIfB})$$

$$\frac{\rho\,!\,x \Downarrow r}{x[\rho_1] \Downarrow r}\ (\textsc{NSVar}) \qquad \frac{}{(\lambda x : T.e)[\rho] \Downarrow (\lambda\!\!\lambda.e[(x : T) : \rho])}\ (\textsc{NSFun}) \qquad \frac{e_1[\rho] \Downarrow v_1 \quad e_2[\rho] \Downarrow v_2 \quad v_1\ v_2 \Downarrow_{ap} r}{(e_1\, e_2)[\rho] \Downarrow r}\ (\textsc{NSApp})$$

$$\frac{e_1[\rho] \Downarrow \texttt{Blame}\ \ell}{(e_1\, e_2)[\rho] \Downarrow \texttt{Blame}\ \ell}\ (\textsc{NSAppBL}) \qquad \frac{e_1[\rho] \Downarrow \lambda\!\!\lambda.e[(x : T) : \rho'] \quad e_2[\rho] \Downarrow \texttt{Blame}\ \ell}{(e_1\, e_2)[\rho] \Downarrow \texttt{Blame}\ \ell}\ (\textsc{NSAppBR})$$

$$\frac{e[\rho] \Downarrow v \quad \langle c\rangle v \Downarrow_{cr} r}{(\langle c\rangle e)[\rho] \Downarrow r}\ (\textsc{NSCoe}) \qquad \frac{e[\rho] \Downarrow \texttt{Blame}\ \ell}{(\langle c\rangle e)[\rho] \Downarrow \texttt{Blame}\ \ell}\ (\textsc{NSCoeB}) \qquad \frac{}{(\texttt{Blame}\ \ell)[\rho] \Downarrow \texttt{Blame}\ \ell}\ (\textsc{NSBla})$$

Figure 7.11: Natural semantics for closure normalisation. Coercion normalisation $\Downarrow_{\mathbf{ED}}$ and coercion application $\Downarrow_{cr}$ are defined in Figure 7.10. Auxiliary function $\rho\,!\,x$ is defined in Figure 7.7.

Environments for the closure-converted semantics: $\boxed{\rho}$

$$\rho \quad ::= \quad \epsilon \mid (x \mapsto v) : \rho \mid (x : T) : \rho$$

Closure-converted look-up function: $\boxed{\rho\,!\,x = \{\mathsf{cl} \mid T\}}$

$$\rho\,!\,x \quad = \quad \begin{cases} \mathsf{cl} & \text{if } \mathit{bind} = (x \mapsto \mathsf{cl}) \\ T & \text{if } \mathit{bind} = (x : T) \end{cases} \qquad \begin{array}{l} \text{where } \mathit{bind} \text{ contains the} \\ \text{first occurrence of } x \text{ in } \rho \end{array}$$

Figure 7.12: Environments and look-up function for the closure-converted semantics in Figure 7.11.

`apply_clos`. The resulting natural semantics is shown in Figure 7.11 and corresponds to the bottom right corner of the inter-derivation diagram (Figure 7.1).

In mathematical notation `VPROC1` will be represented by symbol $\lambda\!\!\lambda$ to suggest the relationship with the meta-level. The symbol will also help us discriminate between the result $\lambda\!\!\lambda.e[\rho']$ of evaluating an abstraction closure $(\lambda x : T.e)[\rho]$, and an input closure $e[\rho']$. The closure-converted result hierarchy is morally the same as the original result hierarchy (Figure 7.6) except that procedures are represented at the object-level by $\lambda\!\!\lambda.e[\rho']$.

In rule NSFun, the type-annotated formal parameter $(x : T)$ is stored directly in the environment instead than attached to $\lambda\!\!\lambda$ for lexical scoping reasons explained in Section 7.8. The definition of environment and look-up is duly adapted (Figure 7.12).

Datatypes `item_clos` and `environment_clos` in Code 3.1 implement the closure-converted environments in Figure 7.7. Function `mk_cast_clos` is the closure-converted cast combinator, with values and results in the closure-converted results hierarchy. Functions `apply_coercion_clos`, `apply_clos`, and `eval_clos` in Code 3.1 implement the natural semantics in Figure 7.11.

Recall from Section 7.4.2 that the code uses an unfolded representation of closures. A datatype for closures will be introduced in Section 7.10 for the reduction semantics.

7.6.2 2-layer continuation-passing-style transformation

The hybrid nature of the semantic artefacts require specific CPS transformation techniques to keep coercion and expression semantics apart. We use 2-layer CPS (2CPS) (Danvy *et al.*, 2013) to introduce two function spaces for the rest of the computation: an inner space of continuations for coercion normalisation, and an outer space of meta-continuations for expression normalisation. We 2CPS-transform `eval_clos` by naming intermediate results of computation, respectively for coercion and expression normalisation, and by turning all the calls into tail calls. Functions `normalise_coercion_cps`, `mk_cast_cps`, `apply_coercion_cps`, `apply_cps`, and `eval_cps` in Code 3.2 implement the 2CPS-normaliser (a refunctionalised

abstract machine) in the bottom middle of Figure 7.1.

After 2CPS transformation, the functional correspondence would have continued by defunctionalising the 2CPS-normaliser. However, we halt the functional correspondence at this point and move on to the reduction semantics (top left corner of Figure 7.1). In Section 7.8, we introduce the calculus of closures $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$, which allows to define the closure-converted small-step reduction semantics.

7.7 Tackling the other side of the diagram

Section 7.4 to Section 7.6 have dealt with the right-hand side of the inter-derivation diagram (Figure 7.1). We now move to the other side. The 2CPS-normaliser is an artefact with closures, but the reduction semantics given in Sections 7.2 and 7.3 are for plain expressions. In Section 7.8 we extend $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ to $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$, a simply-typed lambda calculus of closures with explicit casts, whose reduction semantics is the starting point of the syntactic correspondence that will arrive at the 2CPS-normaliser.

7.8 The calculus of closures

Figure 7.13 shows the syntax, contraction rules, and implementable reduction semantics of $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$. This section is also of interest to the gradual-type-theorist. Observe that boxed rule $\textsc{StepCst}_{\rho}$ is present.

Closures cl consist of *proper closures* $e[\rho]$ and some additional *ephemeral closures*, in the spirit of (Biernacka & Danvy, 2007), that lift expression scopes to closure scopes, and are needed to define the reduction contexts $\mathbf{Cl}[\ ]$ in the bottom of the figure. Ephemeral constructors consist of closure constants $\mathsf{con}\ k$, closure primitive application $\mathsf{prim}\ op\ \mathsf{cl}$, closure conditionals $\mathsf{if}\ \mathsf{cl}\ \mathsf{cl}\ \mathsf{cl}$, closure applications $\mathsf{cl} \cdot \mathsf{cl}$, closure abstractions $\lambda\!\!\lambda.\mathsf{cl}$, closure coercion casts $\langle c \rangle \mathsf{cl}$, and closure blames $\mathsf{Blame}\ \ell$. The type system for closures is a straightforward extension of the type-system for expressions (Figure 7.3) and we omit it for lack of space. The hierarchy of results is the one for expressions but lifted to ephemeral closures. Observe that closure blames are closure results.

The contraction rules are separated in three groups. The first seven rules induce *ephemeral expansion*, *i.e.*, a relation that lifts proper closures to their corresponding ephemeral constructors, and distributes the outermost environment over the closure scopes. Ephemeral expansion is needed in small-step artefacts, but will be shortcut (Biernacka & Danvy, 2007) when deriving the big-step semantics by applying compression of corridor transitions in Section 7.10.4.

Observe that in rule $\textsc{Lam}_{\rho}$ lambda abstractions are ephemerally expanded although reduction will not 'go under lambda'. We have introduced the ephemeral closure abstraction $\lambda\!\!\lambda.\mathsf{cl}$ to match the procedure representations in the closure-converted natural semantics of Figure 7.11. The $\lambda\!\!\lambda$ symbol helps discriminate between an input closure and the result of reducing an abstraction closure (recall the similar discussion in Section 7.6.1). Rule $\textsc{Lam}_{\rho}$

Syntax: $\boxed{\mathsf{cl} \in \lambda\rho^{\langle\cdot\rangle}_{\rightarrow}}$

$$
\begin{array}{lrcl}
\text{environments} & \rho & ::= & \epsilon \mid (x \mapsto \mathsf{v}) : \rho \mid (x : T) : \rho \\
\text{closures} & \mathsf{cl} & ::= & e[\rho] \mid \mathbf{con}\ k \mid \mathbf{prim}\ op\ \mathsf{cl} \mid \mathbf{if}\ \mathsf{cl}\ \mathsf{cl}\ \mathsf{cl} \\
 & & & \mid \lambda\!\!\lambda.\mathsf{cl} \mid \mathsf{cl} \cdot \mathsf{cl} \mid \langle c \rangle \mathsf{cl} \mid \mathbf{Blame}\ \ell \\
 & & & \\
\text{closure simple values} & \mathsf{s} & ::= & \mathbf{con}\ k \mid (\lambda\!\!\lambda.e[(x : T) : \rho]) \\
\text{closure values} & \mathsf{v} & ::= & \mathsf{s} \mid \langle \overline{c} \rangle \mathsf{s} \\
\text{closure results} & \mathsf{r} & ::= & \mathsf{v} \mid \mathbf{Blame}\ \ell
\end{array}
$$

Contraction: $\boxed{\mathsf{cl} \longrightarrow_\rho \mathsf{cl}}$

$$
\begin{array}{rl}
k[\rho] \longrightarrow_\rho \mathbf{con}\ k & (\textsc{Con}_\rho) \\
(op\ e)[\rho] \longrightarrow_\rho \mathbf{prim}\ op\ (e[\rho]) & (\textsc{Prim}_\rho) \\
(\mathtt{if}\ e_1\ e_2\ e_3)[\rho] \longrightarrow_\rho \mathbf{if}\ (e_1[\rho])\ (e_2[\rho])\ (e_3[\rho]) & (\textsc{IfTE}_\rho) \\
(\lambda x : T.e)[\rho] \longrightarrow_\rho (\lambda\!\!\lambda.e[(x : T) : \rho]) & (\textsc{Lam}_\rho) \\
(e_1\, e_2)[\rho] \longrightarrow_\rho (e_1[\rho]) \cdot (e_2[\rho]) & (\textsc{App}_\rho) \\
(\langle c \rangle e)[\rho] \longrightarrow_\rho \langle c \rangle (e[\rho]) & (\textsc{Coer}_\rho) \\
(\mathtt{Blame}\ \ell)[\rho] \longrightarrow_\rho \mathbf{Blame}\ \ell & (\textsc{Bla}_\rho) \\
 & \\
x[\rho] \longrightarrow_\rho \mathsf{cl} \quad \text{where } \rho\,!\,x = \mathsf{cl} & (\textsc{Var}_\rho) \\
 & \\
(\lambda\!\!\lambda.e[(x : T) : \rho]) \cdot \mathsf{v} \longrightarrow_\rho e[(x \mapsto \mathsf{v}) : \rho] & (\beta_\rho) \\
\mathbf{prim}\ op\ (n[\rho]) \longrightarrow_\rho \mathbf{con}\ (\delta(op, n)) & (\delta_\rho) \\
\mathbf{if}\ (\mathbf{con}\ k)\ \mathsf{cl}_1\ \mathsf{cl}_2 \longrightarrow_\rho \begin{cases} \mathsf{cl}_1 & \text{if } k = \mathtt{t} \\ \mathsf{cl}_2 & \text{if } k = \mathtt{f} \end{cases} & (\textsc{If}_\rho) \\
\boxed{\langle c_1 \rangle \mathsf{s} \longrightarrow_\rho \langle c_2 \rangle \mathsf{s} \quad \text{if } c_1 \longmapsto_{\mathtt{x}} c_2} & (\textsc{StepCst}_\rho) \\
 & \\
\langle \iota \rangle \mathsf{s} \longrightarrow_\rho \mathsf{s} & (\textsc{IdCst}_\rho) \\
\langle d \rangle \langle \overline{c} \rangle \mathsf{s} \longrightarrow_\rho \langle \overline{c}\,; d \rangle \mathsf{s} & (\textsc{CmpCst}_\rho) \\
(\langle \tilde{c} \to \tilde{d} \rangle \mathsf{s}) \cdot \mathsf{v} \longrightarrow_\rho \langle \tilde{d} \rangle (\mathsf{s} \cdot \langle \tilde{c} \rangle \mathsf{v}) & (\textsc{AppCst}_\rho) \\
\langle \mathtt{Fail}^\ell \rangle \mathsf{s} \longrightarrow_\rho \mathbf{Blame}\ \ell & (\textsc{FailCast}_\rho) \\
\langle (\tilde{c} \to \tilde{d})\,; \mathtt{Fail}^\ell \rangle \mathsf{s} \longrightarrow_\rho \mathbf{Blame}\ \ell & (\textsc{FailFC}_\rho)
\end{array}
$$

Reduction semantics: $\boxed{\mathsf{cl} \longmapsto_\rho \mathsf{cl}}$

$$
\begin{array}{rcl}
\mathbf{Cl}[\,] & ::= & [\,] \mid \mathbf{prim}\ op\ (\mathbf{Cl}[\,]) \mid \mathbf{if}\ (\mathbf{Cl}[\,])\ \mathsf{cl}\ \mathsf{cl} \\
 & & \mid (\mathbf{Cl}[\,]) \cdot \mathsf{cl} \mid \mathsf{v} \cdot (\mathbf{Cl}[\,]) \mid \langle c \rangle (\mathbf{Cl}[\,])
\end{array}
$$

$$
\frac{\mathsf{cl} \longrightarrow_\rho \mathsf{cl}'}{\mathbf{Cl}[\mathsf{cl}] \longmapsto_\rho \mathbf{Cl}[\mathsf{cl}']} \qquad \frac{}{\mathbf{Cl}[\mathbf{Blame}\ \ell] \longmapsto_\rho \mathbf{Blame}\ \ell}
$$

Figure 7.13: Syntax, contraction rules, and implementable reduction semantics of $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$. Auxiliary look-up function $\rho\,!\,x$ is defined in Figure 7.12.

also pushes a type annotation $(x : T)$ on the environment, similar to rule NSFun in Figure 7.11. The purpose of this is to close the scope of the abstraction body e such that every variable points to some element in the environment, preventing dangling variables in e.[7]

The second group of rules consists of rule $\textsc{Var}_\rho$ alone, which performs substitution on demand by looking up the binding of a variable. The look-up function always returns a closure cl because the reduction semantics never goes under lambda and the type system enforces that all the variables are bound.

The third and last group are the closure versions of the contraction rules in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. These rules induce reduction proper. Notice that (β_ρ) discards the $\lambda\lambda$ in the operator and replaces the formal parameter's type annotation by the actual binding.

The reduction contexts $\mathbf{Cl}[\ ]$ and the reduction semantics $\longmapsto_\rho$ for closures are simply the closure version of the reduction contexts and reduction semantics of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. The reduction semantics of $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$ simulates stepwise the reduction semantics in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ by means of substitution function σ that flattens all the delayed substitutions in a closure. The proof, which is omitted, goes in a similar way as the proof of the stepwise connection between $\rightarrow_{no}$ and $\rightarrow_{\widetilde{no}}$ in Section 6.7.9.

7.8.1 The correctness theorems

We are now in a position to state the correspondence between the instantiated definitional interpreter and the reduction semantics for closures with respect to **ED**:

Theorem 7.8.1. *Given a well-typed coercion c_1 we have $c_1 \Downarrow_{\mathbf{ED}} \hat{c}_2$ iff $c_1 \longmapsto^*_{\mathbf{ED}} \hat{c}_2$.*

Proof. By establishing the correspondence between `normalise_coercion_nor` (Code 2.2.3, Section 7.5.2) and `normalise_coercion` (Code 5.3.2, Section 7.9). □

Theorem 7.8.2. *If every coercion labelled with ℓ in program e respects subtyping, then $e[\epsilon] \not\longmapsto^*_\rho \mathsf{Blame}\ \ell$.*

Proof. The proof is straightforward by induction on $\longmapsto_\rho$. □

Theorem 7.8.3. *Given a well-typed expression e, we have $e[\epsilon] \Downarrow \mathsf{r}$ iff $e[\epsilon] \longmapsto^*_\rho \mathsf{r}$.*

Proof. By establishing the correspondence between `eval_clos` (Code 3.1, Section 7.6.1) and `normalise` (Code 5.3.2, Section 7.9). □

Different from the conjectures in Section 7.4.3, we do not need a separate theorem stating soundness of subtyping for the natural semantics $\Downarrow$ because Theorem 7.8.3 proves it equivalent to reduction semantics $\longmapsto_\rho$.

[7] This feature is reminiscent of the dummy bindings standing for formal parameters in Crégut's full-reducing Krivine machine (Crégut, 1990, 2007), and in the equivalent semantic artefacts inter-derived in (García-Pérez *et al.*, 2013).

7.9 Implementing the reduction semantics

We turn to the implementation of $\longmapsto_\rho$ in Figure 7.13. Similarly to the 2CPS discussion of Section 7.6.2 we use continuations for $\longmapsto_{\mathbf{x}}$ (the reduction semantics of coercions) and meta-continuations for $\longmapsto_\rho$ (the reduction semantics of closures). In Code 5 we describe a transformation step which is uninteresting to the gradual-type-theorist and for lack of space we merely outline it. We start with hybrid search functions that implement a structural operational semantics. We then derive the reduction semantics from the search functions, by CPS transformation, simplification, and defunctionalisation. This standard practice (Biernacka & Danvy, 2007; Danvy & Millikin, 2008; Danvy, 2008a; Danvy & Millikin, 2009; Danvy *et al.*, 2011) is not essential to establish a syntactic correspondence, but it reveals better the correspondence between reduction contexts and defunctionalised continuations. Moreover, the transformation step clarifies two important points and justifies the accompanying design decisions. We elaborate on this two points in the following paragraphs. We strongly advise the program-derivationist to read Code 5 alongside this section.

The first point: the simplification step prescribes that the search functions discard the current continuation when a redex is found. However, a tail-recursive implementation of our hybrid semantics would require to keep the closure meta-continuation in order to throw into it the found coercion redex. Since the closure meta-continuation will be dropped by the simplified coercion semantics, the closure semantics needs to invoke the coercion semantics in non-tail-recursive fashion, by delimiting its invocation by passing the initial continuation.[8] Thus, the implementation of the small-step semantics is not in 2CPS anymore, but rather in *continuation-composing style* (*i.e.*, two 1-layer CPS programs which are glued together by the closure semantics delimiting the invocations of the coercion semantics). All this is unavoidable. Dropping the meta-continuation in the inner simplified semantics is essential for the separated transformation of closure and coercion semantics. For the coercions, the tail calls to iterate need to happen immediately after decompose, enabling to light-weight fuse them in Code 6.3.

The second point: decomposition (to have a term and its context) is fundamental to implement a trampolined style reduction semantics (Ganz *et al.*, 1999) (a driver loop iterating decomposition, contraction, and recomposition). We have a hybrid reduction semantics involving closures and coercions. The following rule (implicitly entailed by $\textsc{StepCst}_\rho$ in Figure 7.13) illustrates the inclusion of the inner semantics in the outer one:

$$\frac{\mathbf{C}[c] \longmapsto_{\mathbf{x}} \mathbf{C}[c']}{\mathbf{Cl}[\langle \mathbf{C}[c] \rangle s] \longmapsto_\rho \mathbf{Cl}[\langle \mathbf{C}[c'] \rangle s]}$$

In order to implement the subsidiary coercion semantics $\longmapsto_{\mathbf{x}}$ in trampolined style, we have to modify the datatype representing the outer redices $\mathbf{Cl}[\langle \mathbf{C}[c] \rangle s]$ to include the inner

[8] The sceptical program-derivationist is invited to attempt the simplification step in a true 2CPS program implementing a semantics with hybrid redices, like the ones entailed by rule $\textsc{StepCst}_\rho$ in Figure 7.13.

decomposition $\mathbf{C}[c]$, rather than just a plain coercion $c_1 \equiv \mathbf{C}[c]$. This is implemented by clause

```
ESTEPCST1 of decomposition * simple_value
```

of datatype `redex1` in Code 5.3.1. This is ultimately isomorphic to 2CPS (*i.e.*, iterated CPS) since the inner decomposition includes both the continuation and the meta-continuation. The apparent detour from using 2CPS upfront is due to the need to perform the simplification step in each of the closure and coercion semantics in a modular way.

Code 5.3 implements the reduction semantics $\longmapsto_\rho$ in Figure 7.13, which corresponds to the top left corner in Figure 7.1. In the following sections, we apply the syntactic correspondence and arrive at an abstract machine which will be refunctionalised into the 2CPS-normaliser in Section 7.6.2 and Code 3.2, thus closing the gap and completing the inter-derivation.

7.10 The syntactic correspondence

In Section 7.9, we implemented the reduction semantics $\longmapsto_\rho$, which is the starting point of the syntactic correspondence arriving at the abstract machine on the bottom left corner of Figure 7.1. The syntactic correspondence (Danvy & Nielsen, 2004; Danvy, 2008b; Danvy *et al.*, 2011) consists of refocusing, inlining of contraction function, lightweight-fusion by fix point promotion (Ohori & Sasano, 2007), and compression of corridor transitions. These steps are standard and hence merely outlined in the chapter, except for the specific details concerning our hybrid semantics. In the fourth step, we elaborate on two different classes of corridor transitions found in the literature (Danvy, 2008b) (Section 7.10.4).

On occasion, we generically refer to both the coercion and closure artefacts by naming the entry function, *e.g.*, `normalise1` in Code 6.1.

7.10.1 Refocusing

The refocus function maps a pair (contractum, context) to the decomposition for the next redex in the reduction sequence. Extensional refocus consist of (respectively on coercions and closures) recomposition followed by decomposition. The refocusing step deforests this detour, turning the extensional refocus function into an intensional refocus function which is an alias for the decompose function (Danvy & Nielsen, 2004).

Since our semantics is hybrid, we apply refocusing to the coercion and the closure artefacts in succession. First, we deforest recomposition followed by decomposition turning the extensional `refocus_coercion` in Code 5.3.2 into the intensional `refocus1_coercion` in Code 6.1. This is an alias for function `decompose_coercion`. Functions `iterate1_coercion` and `normalise1_coercion` follow from that. Before performing the same operation in the closure artefact, we coalesce all the $\textsc{StepCst}_\rho$ steps in the closure reduction semantics.

The modified inclusion-of-semantics rule now reads:

$$\frac{c \longmapsto^{*}_{\mathbf{x}} \hat{c}}{\mathbf{Cl}[\langle c \rangle s] \longmapsto_{\rho} \mathbf{Cl}[\langle \hat{c} \rangle s]}$$

This transformation trivially preserves equivalence. To implement the rule above, we modify the decomposition of closures accordingly. In Code 6.1, the clause for meta-continuation `MC5` in function `decompose1_meta_cont` now invokes `normalise1_coercion`. In the same program clause, there is no case returning a `ESTEPCST1` redex, since the coercion found `nc` (on which the program is pattern-matching after `normalise1_coercion`) is trivially a normal coercion.

We turn `refocus_closure` in Code 5.3.2 into the intensional `refocus1_closure` in Code 6.1 which is an alias for `decompose1_closure`.

7.10.2 Inlining the contraction function

We inline the contraction functions in the corresponding iterate functions (Danvy, 2008b), obtaining `normalise2` in Code 6.2. Due to the modified rule in Section 7.10.1, the case for `ESTEPCST1` redices in `contract_closure` is no longer considered.

7.10.3 Lightweight-fusing decompose and iterate

There are several invocations of decomposition followed by iteration in the iterate and normalise functions. We fuse them together in a single normalise function by applying lightweight fusion. As in Section 7.10.1, we proceed in succession for the coercion and the closure artefacts. The resulting reduction-free normaliser `normalise3` is in Code 6.3.

7.10.4 Compressing *static* and *dynamic* corridor transitions

Some of the transitions in `normalise3` are to configurations where there is only one possible further transition. These are called *corridor* transitions, and by hereditarily compressing them, the iterate functions will become unused and could be safely removed.

The conventional corridor transitions (for which we use the epithet *static*) are those detected by looking at the code of the normalising functions, *i.e.*, the program clauses where the right-hand side consists of a single tail call, or of a selection statement having a unique case (Danvy, 2008b; Danvy *et al.*, 2011). These transitions are compressed by successively unfolding the right-hand side of the program clauses involved. The shortcut operation coalescing ephemeral expansion (Biernacka & Danvy, 2007) belongs to this category of corridor transitions. By compressing the static corridor transitions we obtain the big-step normaliser `normalise4` in Code 6.4.

The not-so-conventional corridor transitions (for which we use the epithet *dynamic*), are those starting at a configuration where the input term is irreducible, *i.e.*, a normal coercion or a value (Danvy, 2008b, p.140-141). For coercions, remember from Section 7.4 that we use a single `coercion` datatype both for arbitrary coercions and for the hierarchy

Embed function for closure values: $\boxed{\downarrow \mathsf{v} = e[\rho]}$

$$\begin{array}{rcl} \downarrow (\mathbf{con}\ k) & = & k[\epsilon] \\ \downarrow (\lambda\!\!\lambda.e[(x:T):\rho]) & = & (\lambda x:T.e)[\rho] \\ \downarrow (\langle\!\langle c \rangle\!\rangle \mathsf{v}) & = & (\langle c \rangle e)[\rho] \quad \text{where } \downarrow \mathsf{v} = e[\rho] \end{array}$$

Figure 7.14: Embed function for closure values

of normal coercions. For the closures, in Code 4 we introduced datatype `value_clos` and function `embed_clos`, the latter implementing the embedding function $\downarrow\mathsf{v}$ in Figure 7.14.[9] Since the programs are in defunctionalised CPS, all the calls are tail calls (respectively to the coercion or closure semantics). The computation will eventually throw the irreducible input term into the current continuation (meta-continuation respectively). Thus, these administrative transitions can be coalesced until that point, *i.e.*, a call to `normalise4_cont` or `normalise4_meta_cont` respectively. Compressing dynamic corridor transitions reveals more opportunities to compress static corridor transitions. By compressing them all we obtain `normalise5` in Code 6.5, which implements the abstract machine at the bottom left corner of Figure 7.1.

Let us show one of such dynamic corridor transitions:

```
  normalise4_closure (embed_clos f1, MC3 (CCOER (c1, embed_clos v), MC5 (c2, mk)))
=
  normalise4_meta_cont (MC3 (CCOER (c1, embed_clos v), MC5 (c2, mk)), f1)
=
  normalise4_closure (CCOER (c1, embed_clos v), MC4 (f1, MC5 (c2, mk)))
=
  normalise4_closure (embed_clos v, MC5 (c1, MC4 (f1, MC5 (c2, mk))))
=
  normalise4_meta_cont (MC5 (c1, MC4 (f1, MC5 (c2, mk))), v)
```

The normaliser specifies a control-flow invariant for the cases matching the initial clause of the corridor transition. The invariant allows the meta-continuation stack to be loaded with a *fixed* sequence of defunctionalised meta-continuations, which will help us refunctionalise the abstract machine into several mutually recursive functions in Section 7.11.1.

7.11 Closing the gap

In this section we close the gap between the right- and left-hand sides of the inter-derivation diagram (Figure 7.1). We defunctionalise the abstract machine into a 2CPS program with

[9] The embedding function is used in `contract_closure` in Code 5.3.2. Embed only considers closure values because closure blames are short-circuited to closure results and do not appear in redices. Observe that embed followed by normalise is the identity.

several mutually recursive functions which is almost the 2CPS-normaliser in Figure 7.1. Then, we apply some cosmetic transformations to remove minor differences between the refunctionalised abstract machine and the 2CPS-normaliser, thus concluding the derivation.

7.11.1 Refunctionalising the abstract machine

We observe two facts about the dispatcher for meta-continuations `normalise5_meta_cont`:

1. In the clause for `MC5`, the program either invokes the clause itself (`normalise5_meta_cont` passing `MC5`), or makes a delimited non-tail call to `normalise5_coercion` and then returns a blame or throws some intermediate result into the current meta-continuation. This clause can be refunctionalised into a stand-alone recursive function, which we name `apply6_coercion` (Code 7.1).

2. In the clause for `MC4`, the program either calls `normalise5_closure`, or invokes the dispatcher `normalise5_meta_cont` passing a new continuation which adds up to three new constructors, *i.e.*, `MC5 (c1, MC4 (f1, MC5 (c2, mk)))`. This clause can be refunctionalised into a stand alone recursive function which unwinds the defunctionalised continuation, and invokes `apply6_coercion` and itself according to the occurrences of `MC4` and `MC5` in the defunctionalised continuation. We name the function `apply6` (Code 7.1).

   Although the clause for `MC4` invokes `normalise5_closure`, equivalence is preserved because the latter never invokes `normalise5_meta_cont` directly passing `MC4`.

The rest of the clauses and functions are straightforwardly defunctionalised.

We also undelimit the inner continuations, turning the non-tail calls into tail calls. This is only possible at a reduction-free artefact, since the program does not need to consider the individual redices in the reduction sequence, in particular the coercion redices (recall the discussion in Section 7.9). We have decided to do this transformation after refunctionalisation to save us from introducing a new datatype for defunctionalised continuations. The result is the 2CPS refunctionalised abstract machine `normalise6` in Code 7.1.

7.11.2 Cosmetic transformations

We remove some minor differences between the refunctionalised normaliser `normalise6` in Code 7.1 and the 2CPS-normaliser `eval_cps` in Code 3.2.

We inline `apply6_coercion` in Code 7.1 into itself (notice that the value `sv` passed in the recursive call is a simple value), duplicating the selection statement in the second clause. This selection statement is, in turn, protruded (*i.e.*, inversely inlined) into combinator `mk_cast7` (Code 7.2). We unfold datatype `closure` into a pair (`expression`, `environment_clos`) and protrude combinator `mk_arrow7` (Code 7.2). The result is `normalise7` in Code 7.2, which is exactly the same as the 2CPS-normaliser `eval_clos` in Code 3.2.

This establishes the correspondence between $=_{\langle\cdot\rangle}$ and $\longmapsto$, and between $\Downarrow_{\mathbf{ED}}$ and $\longmapsto_{\mathbf{ED}}$, constituting a proof by program transformation of Theorems 7.8.1 and 7.8.3.

Theorems 7.8.1 and 7.8.3 can be generalised for other choices of dynamic semantics by applying the correspondence described through the chapter starting with a different set of coercion artefacts. Layering and 2CPS allows us to reuse the off-the-shelf infrastructure, in particular the closure artefacts.

7.12 Conclusions and related work

We have shown the inter-derivation of semantic artefacts for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. Our choice of the 2CPS is motivated by the need for a modular coercion semantics that can be plugged into the $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ hybrid semantics. This allows us to generalise the theorems for a family of dynamic semantics reusing most of the inter-derivation for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$.

We have presented the calculus $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$, which is an important ingredient to inter-derive the closure-converted version of the definitional interpreters in (Siek & Garcia, 2012). The semantics in $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$ simulates step-by-step-wise the semantics in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$.

In (Danvy *et al.*, 2013) the 2CPS is applied to inter-derive a full-reducing *eval-readback* machine of Curien (Curien, 1993) that normalises pure untyped lambda calculus terms. The machine relies on a *hybrid* reduction strategy with two separated stages for eval and readback. We have independently investigated in (García-Pérez *et al.*, 2013) a different approach for *single-stage* (as opposed to eval-readback) hybrid artefacts, showcasing the derivation of the full-reducing Krivine machine (Crégut, 1990) from the operational semantics of normal order. In (Danvy *et al.*, 2013) the subsidiary strategy is modular, but this introduces a conceptual overhead in the 2CPS transformations. In (García-Pérez *et al.*, 2013) we show how to use plain CPS when the target strategy is single-staged, but this requires reasoning on the shape of the continuation stack. Both approaches differ in their weaknesses and strengths, as well as in their range of applicability.

The semantic artefacts in this chapter are qualitatively different from those in (Danvy *et al.*, 2013) and (García-Pérez *et al.*, 2013). The semantics $\Downarrow$ here is single-stage (there is only one pass of the big-step definitional interpreter) but its implementation is 2-layered. In the big-step artefacts we use 2CPS. In the small-step artefacts we disentangle the inner continuation space by delimiting the continuations in the non-tail calls to the coercion semantics. This way, we keep the semantics in (Siek & Garcia, 2012), but arrive at a solution which is modular with respect to the coercion semantics.

Garcia (2013) has tackled and solved the challenge of defining a reduction semantics for coercions which is 'complete in the face of inert compositions and associativity'. We have rather followed the 'ad hoc reassociation scheme' in (Siek & Garcia, 2012), proving the correctness conjectures there. Garcia also introduces threesome-based variants of the Blame Calculus. We believe that the semantics for the threesome-based gradually-typed lambda calculi are good candidates for applying the techniques in this chapter. Thanks to hybrids and 2CPS, modularity with respect to the blame calculi would be straightforward.

8
General Conclusions

The idea of hybrid strategies, which appeared informally in (Sestoft, 2002), articulates most of the work in this thesis. Hybrid strategies are paramount when studying full reduction because the full-reducing strategies that exhibit desirable meta-theoretical properties happen to be hybrid. This thesis shows that the hybrid character is intrinsic to the strategy's *nature*, not to its *style*, *i.e.*, it is unconnected to any particular definition or to representational concerns.

The beta-cube systematises the space of strategies unveiled by (Sestoft, 2002), and shows how a particular kind of hybrid strategies (those relying on a single subsidiary strategy which is a right identity of the hybrid) have a corresponding NBE-style presentation, in which the eval stage corresponds to the subsidiary and the readback stage distributes reduction over the subterms of the result delivered by eval.

The weak-reducing character of the lambda-value calculus (Egidi *et al.*, 1991, 1992; Paolini & Ronchi Della Rocca, 1999; Ronchi Della Rocca & Paolini, 2004; Accattoli & Paolini, 2012) has been emphasised because the notion of lambda-value normal form was considered uninteresting (Ronchi Della Rocca & Paolini, 2004). The emphasis on weak-reducing machines, like the SECD machine of (Landin, 1964), and the misconception about the by-value and the by-wnf calling policies entail a notion of operational relevance which is not in accord with the reduction theory of the lambda-value calculus, but only with its weak-reducing sub-theory (Paolini & Ronchi Della Rocca, 1999). This has been overlooked in the literature because weak reduction entails good model-theoretical properties (Abramsky, 1990; Egidi *et al.*, 1991, 1992). However, the lambda-value calculus has a meta-theory of its own, which has remained unexplored. This thesis helps to reinstate the full-reducing character of lambda-value by disclosing interesting aspects of its meta-theory. This thesis develops the notion of needed reduction (Barendregt *et al.*, 1987) in this calculus, based on which quasi-v-solvability is defined. Quasi-v-solvability is more accurate than the existing 'call-by-value solvability' (Paolini & Ronchi Della Rocca, 1999; Accattoli & Paolini, 2012). This contribution helps for the endeavour of establishing a 'standard theory' of the lambda-value calculus which resembles that of the classical lambda calculus (Barendregt,

1984; Abramsky, 1990).

A reduction strategy stands at the basis of an operational semantics, which is implemented by a so called semantic artefact. The semantic artefacts can be inter-derived by program transformation (Reynolds, 1998; Ager *et al.*, 2003b; Danvy, 2006a; Danvy *et al.*, 2007; Danvy & Millikin, 2008; Danvy *et al.*, 2011). The full-reducing strategies can be implemented by a single-stage hybrid artefact or by a multiple-stage artefact in NBE-style. An n-stage artefact can be fused by lightweight fusion by fixed-point promotion (LWF) into a single-stage hybrid artefact with n modes. The inter-derivation techniques in (Munk, 2008; Danvy *et al.*, 2013) depart from the n-stage artefacts, which requires the use of n-layer CPS and yields an abstract machine with n stacks. This thesis shows how to carry the inter-derivation to the single-stage hybrid artefacts by observing a shape invariant of the control stack. This enables an inter-derivation using single-layer CPS and yields an abstract machine with only one stack. The shape invariant of the control stack reflects the 'staging' control structure in the n-stage artefacts.

Most of the implementations of abstract machines rely on closures and on the environment technique (Landin, 1964). The full-reducing strategies considered in this thesis can be closure converted into strategies in the $\lambda_{\widetilde{\rho}}$-calculus introduced in Chapter 6. This novel calculus of closures, which relies on the use of both be Bruijn indices and levels, generalises and extends several calculi in the literature (Curien, 1991; Biernacka & Danvy, 2007; Crégut, 2007). Contrary to other calculi of closures, $\lambda_{\widetilde{\rho}}$ simulates reduction in plain λK in a step-by-step fashion. Besides, $\lambda_{\widetilde{\rho}}$ enforces SOS with *index alignment* and *balanced derivations*, an important contribution which helps to solve the paramount issue with binders, *i.e.*, reasoning locally in a scope where the binding of a free variable is not available (Aydemir *et al.*, 2008).

Layered (*i.e.*, hybrid) operational semantics are constantly springing up. The gradually-typed lambda calculus (Siek & Taha, 2006; Siek *et al.*, 2009; Siek & Garcia, 2012) is a recent example, where a family of dynamic semantics for coercions can be plugged into the static semantics of a host simply-typed lambda calculus. This thesis shows how the 2CPS techniques can be applied together with the hybrid strategies to inter-derive interpretations of the gradually-typed lambda calculus in a way which is parametric in the semantic artefact for coercions (*i.e.*, the different inter-derivations for the choices of coercion semantics can be plugged in the inter-derivation of the host calculus, which would be reused).

Bibliography

Abadi, M., Cardelli, L., Curien, P.-L. & Lévy, J.-J. (1991). Explicit substitutions. *Journal of Functional Programming* **1**(4), 375–416.

Abelson, H., Sussman, G. J. & Sussman, J. (1985). *Structure and Interpretation of Computer Programs*. Cambridge, Massachusetts: MIT Press.

Abramsky, S. (1990). The lazy lambda calculus. In: *Research Topics in Functional Programming* (Turner, D. A., ed.). Reading, MA: Addison-Welsey, pp. 65–116.

Accattoli, B. & Paolini, L. (2012). Call-by-value solvability, revisited. In: *Eleventh International Symposium on Functional and Logic Programming (FLOPS 2012)*.

Aehlig, K. & Joachimski, F. (2004). Operational aspects of untyped normalisation by evaluation. *Mathematical Structures in Computer Science* **14**(4), 587–611.

Ager, M. S., Biernacki, D., Danvy, O. & Midtgaard, J. (2003a). From interpreter to compiler and virtual machine: a functional derivation. Tech. Rep. RS-03-14, BRICS, Department of Computer Science, Aarhus University, Denmark.

Ager, M. S., Biernacki, D., Danvy, O. & Midtgaard, J. (2003b). A functional correspondence between evaluators and abstract machines. In: *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press.

Ager, M. S., Danvy, O. & Midtgaard, J. (2005). A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science* **342**(1), 149–172.

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R. & Weirich, S. (2008). Engineering formal metatheory. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*.

Baader, F. & Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press.

Barendregt, Kennaway, Klop & Sleep (1987). Needed reduction and spine strategies for the lambda calculus. *Information and Computation* **75**(3), 191–231.

Barendregt, H. (1972). Solvability in lambda caculi. In: *Proceedings of the Orléans Congrès de Logique*.

Barendregt, H. (1984). *The Lambda Calculus, Its Syntax and Semantics*. North Holland.

Barendregt, H. P. (1971). *Some Extensional Term Models for Combinatory Logics and Lambda Calculi*. Ph.D. thesis, University of Utrecht, Utrecht, NL.

Berger, U. & Schwichtenberg, H. (1991). An inverse of the evaluation functional for typed λ-calculus. In: *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*.

Biernacka, M., Biernacki, D. & Danvy, O. (2005). An operational foundation for delimited continuations in the CPS hierarchy. *CoRR* **abs/cs/0508048**.

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

Borges, J. L. (1944). Tlön, Uqbar, Orbis Tertius. In: *Ficciones*. Editorial Sur, Buenos Aires, Argentina.

Burton, T. (1994). *Ed Wood*. Buena Vista Pictures.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics* **58**, 354–363.

Church, A. (1941). *The Calculi of Lambda Conversion*. Princeton University Press.

Crégut, P. (1990). An abstract machine for lambda-terms normalization. In: *LISP and Functional Programming*.

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

Curien, P.-L. (1991). An abstract framework for environment machines. *Theoretical Computer Science* **82**(2), 389–402.

Curien, P.-L. (1993). *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhaüser.

Curien, P.-L. (2007). Definability and full abstraction. *Electr. Notes Theor. Comput. Sci* **172**, 301–310.

Curry, H. B. & Feys, R. (1958). *Combinatory Logic*, vol. 1. North-Holland.

Danvy, O. (1996). Type-directed partial evaluation. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. St. Petersbur (FL), USA.

Danvy, O. (1998). Online type-directed partial evaluation. In: *Fuji International Symposium on Functional and Logic Programming.*

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

Danvy, O. (2006a). *An Analytical Approach to Programs as Data Objects.* Ph.D. thesis, University of Aarhus. *Doctor Scientiarum* in Computer Science.

Danvy, O. (2006b). Refunctionalization at work. In: *Proceedings of the 8th International Conference on Mathematics of Program Construction.*

Danvy, O. (2008a). Defunctionalized interpreters for programming languages. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming.*

Danvy, O. (2008b). From reduction-based to reduction-free normalization. In: *Advanced Functional Programming*, vol. 5832 of *Lecture Notes in Computer Science.* Springer.

Danvy, O. & Filinsky, A. (1990). Abstracting Control. In: *In Proceedings of the 1990 ACM Conference on LISP and Functional Programming.*

Danvy, O. & Hatcliff, J. (1992). Thunks (continued). In: *WSA.*

Danvy, O. & Johannsen, J. (2013). From outermost reduction semantics to abstract machine. In: *Pre-Proceedings of 23rd Symposium on Logic-Based Program Synthesis and Transformation.*

Danvy, O., Johannsen, J. & Zerny, I. (2011). A walk in the semantic park. In: *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011* (Khoo, S.-C. & Siek, J. G., eds.). ACM.

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

Danvy, O. & Millikin, K. (2009). Refunctionalization at work. *Sci. Comput. Program* **74**(8), 534–549.

Danvy, O., Millikin, K. & Munk, J. (2007). A correspondence between reduction-based and reduction-free normalization functions. Unpublished draft.

Danvy, O., Millikin, K. & Munk, J. (2013). A correspondence between full normalization by reduction and full normalization by evaluation. A scientific meeting in honor of Pierre-Louis Curien.

Danvy, O. & Nielsen, L. R. (2004). Refocusing in reduction semantics. Tech. Rep. RS-04-26, BRICS, Department of Computer Science, Aarhus University, Denmark.

De Bruijn, N. G. (1978). A namefree lambda caculus with facilities for internal definitions of expressions and segments. Tech. Rep. 78-WSK-03, Technological University Eindhoven, Netherlands.

Dershowitz, N. (1981). Termination of linear rewriting systems. In: *ICALP: Annual International Colloquium on Automata, Languages and Programming*.

Dijkstra, E. W. (1968). Go to statement considered harmful. *Communications of the ACM* **11**, 147–148.

Eco, U. (2001). *Come si fa una tesi di laurea*. Bompiani.

Egidi, L., Honsell, F. & Rocca, S. R. D. (1992). Operational, denotational and logical descriptions: a case study. *Fundam. Inform* **16**(1), 149–169.

Egidi, L., Honsell, F. & Ronchi della Rocca, S. (1991). The lazy call–by–value λ–calculus. In: *Proceedings of Mathematical Foundations of Computer Science. (MFCS '91)* (Tarlecki, A., ed.), vol. 520 of *LNCS*. Berlin, Germany: Springer.

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

Felleisen, M. & Flatt, M. (2002). Programming languages and lambda calculi. notes for Utah CS6520.

Felleisen, M. & Friedman, D. P. (1986). Control operators, the SECD-machine, and the lambda-calculus. In: *Formal Description of Programming Concepts III*.

Felleisen, M. & Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* **103**, 235–271.

Filinski, A. & Rohde, H. K. (2004). A denotational account of untyped normalization by evaluation. In: *7th International Conference on Foundations of Software Science and Computation Structures*, vol. 2987 of *Lecture Notes in Computer Science*. Barcelona, Spain.

Filinski, A. & Rohde, H. K. (2005). Denotational aspects of untyped normalization by evaluation. *Theoretical Informatics and Applications* **39**(3), 423–453.

Flecha, M. (1584). *Las ensaladas de Flecha*. Praga: Lorge Negrino. Recopiladas por Mateo Flecha su sobrino.

Ganz, S. E., Friedman, D. P. & Wand, M. (1999). Trampolined style. In: *Proceedings of International Conference on Functional Programming*.

Garcia, R. (2013). Calculating threesomes, with blame. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming.*

García-Pérez, A. & Nogueira, P. (2013). A syntactic and functional correspondence between reduction semantics and reduction-free full normalisers. In: *Proceedings of Partial Evaluation and Program Manipulation.*

García-Pérez, A. & Nogueira, P. (2014a). On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. *Science of Computer Programming* **95**(2), 176–199.

García-Pérez, A. & Nogueira, P. (2014b). A standard theory for the pure lambda-value calculus. Submitted to *11th International Workshop on Domain Theory and Application.*

García-Pérez, A., Nogueira, P. & Moreno-Navarro, J. J. (2013). Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order. In: *Proceedings of Principles and Practice of Declarative Programming.*

García-Pérez, A., Nogueira, P. & Sergey, I. (2014). Deriving interpretations of the gradually-typed lambda calculus. In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation.*

García-Pérez, A., Nogueira, P. & Gallego Arias, E. J. (2010). The beta cube (extended abstract). In: *Proceedings of the 1st International Workshop on Strategies in Rewriting, Proving, and Programming (IWS'10)* (Muñoz, C. & Kirchner, H., eds.). Edinburgh, UK.

Geupel, O. (1989). Overlap closure and termination of term rewriting systems. Tech. Rep. MIP-8922, Universität Passau (Germany).

Grégoire, B. & Leroy, X. (2002). A compiled implementation of strong reduction. In: *Proceedings of International Conference on Functional Programming.*

Guttag, J. V., Kapur, D. & Musser, D. R. (1983). On proving uniform termination and restricted termination of rewrite systems. *SIAM Journal of Computing* **12**(1), 189–214.

Hardin, T., Maranget, L. & Pagano, B. (1998). Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming* **8**(2), 131–172.

Hatcliff, J. & Danvy, O. (1997). Thunks and the λ-calculus. *Journal of Functional Programming* **7**(3), 303–319.

Henglein, F. (1994). Dynamic typing: Syntax and proof theory. *Science of Computer Programming* **22**(2), 197–230. Special Issue on European Symposium on Programming 1992.

Herbelin, H. & Zimmermann, S. (2009). An operational account of call-by-value minimal and classical λ-calculus in "natural deduction" form. In: *Ninth International Conference, TLCA '07, Brasilia, Brazil. July 2009, Proceedings* (Curien, P.-L., ed.), vol. 5608 of *Lecture Notes in Computer Science*. Springer.

Hindley, J. & Seldin, J. (2008). *Lambda-calculus and combinators, an introduction.* Cambridge University Press.

Huneker, J. (1943). Preface to Etude op. 25 no. 11. In: *Chopin: Etudes for the Piano* (Friedheim, A., ed.), vol. 33 of *Schirmer's Library of Musical Classics*.

Kahn, G. (1987). Natural semantics. In: *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)* (Brandenburg, F.-J., Vidal-Naquet, G. & Wirsing, M., eds.), vol. 247 of *Lecture Notes in Computer Science*. Springer-Verlag, pp. 22–39.

Kesner, D. (2007). The theory of calculi with explicit substitutions revisited. HAL-CCSd-CNRS. Hal-00111285 version 3.

Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J. A., Rafkind, J., Tobin-Hochstadt, S. & Findler, R. B. (2012). Run your research: on the effectiveness of lightweight mechanization. In: *Proceedings of Symposium on Principles of Programming Languages*.

Kluge, W. (2010). *Abstract Computing Machines: A Lambda Calculus Perspective.* Springer Publishing Company, Incorporated.

Krivine, J.-L. (2007). A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation* **20**(3), 199–207.

Landin, P. (1964). The mechanical evaluation of expressions. *Computer Journal* **6**(4), 308–320.

Leroy, X. (1991). The ZINC experiment: an economical implementation of the ML language. Tech. Rep. 117, INRIA.

Lescanne, P. (1994). From lambda-sigma to lambda-upsilon: A journey through calculi of explicit substitutions. In: *Proceedings of the 21st Symposium on Principles of Programming Languages*.

Mac Lane, S. (1971). *Categories for the Working Mathematician.* Springer.

McGowan (1970). The correctness of a modified SECD machine. In: *STOC: ACM Symposium on Theory of Computing (STOC)*.

Munk, J. (2008). *A Study of Syntactic and Semantic Artifacts and its Application to Lambda Definability, Strong Normalization, and Weak Normalization in the Presence of State.* Master's thesis, BRICS, Aarhus University, Denmark.

Ohori, A. & Sasano, I. (2007). Lightweight fusion by fixed point promotion. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007* (Hofmann, M. & Felleisen, M., eds.). ACM.

Paolini, L. & Ronchi Della Rocca, S. (1999). Call-by-value solvability. *ITA* **33**(6), 507–534.

Paulson, L. C. (1996). *ML for the Working Programmer*. Cambridge, England: Cambridge University Press, second ed.

Pierce, B. (2002). *Types and Programming Languages*. The MIT Press.

Plotkin, G. (1975). Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science* **1**, 125–159.

Plotkin, G. (1981). A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark.

Quine, W. V. O. (1940). *Mathematical Logic*. Boston, MA: Harvard University Press.

Reynolds, J. C. (1998). Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation* **11**(4), 363–397. Reprinted from the proceedings of the 25th ACM National Conference (1972).

Ronchi Della Rocca, S. & Paolini, L. (2004). *The Parametric Lambda Calculus*. Springer Verlag.

Scholz, H. & Hasenjaeger, G. (1961). *Grundzuege der mathematischen Logik*. Springer.

Scott, D. (1970). Outline of a mathematical theory of computation. In: *Proceedings of the 4th Annual Princeton Conference on Information Sciences and Systems*.

Scott, D. & Strachey, C. (1971). Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford University Computing Laboratory.

Sestoft, P. (2002). Demonstrating lambda calculus reduction. In: *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, vol. 2566 of *Lecture Notes in Computer Science*. Springer.

Siek, J. G. & Garcia, R. (2012). Interpretations of the gradually-typed lambda calculus. In: *Proceedings of the Scheme and Functional Programming Workshop*.

Siek, J. G., Garcia, R. & Taha, W. (2009). Exploring the design space of higher-order casts. In: *Proceedings of the 18th European Symposium on Programming Languages and Systems*.

Siek, J. G. & Taha, W. (2006). Gradual typing for functional languages. In: *Proceedings of the Workshop on Scheme and Functional Programming*.

Stoy, J. E. (1979). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, Massachusetts: The MIT Press.

Swierstra, W. (2012). From mathematics to abstract machine: a formal derivation of an executable Krivine machine. In: *Proceedings of the 4th Workshop on Mathematically Structured Functional Programming*.

TeReSe (2003). *Term Rewriting Systems*, vol. 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Turing, A. M. (1937). On computable numbers, with an application to the Entscheidungsproblem, A correction. *Procedings of the London Mathematical Society* **43**(2), 544–546.

Van Oostrom, V. (2008). Z. Slides available on `http://www.phil.uu.nl/~oostrom/publication/talk/lix010208.pdf`.

Wadsworth, C. (1976). The relation between computational and denotational properties for Scott's D_∞ models. *Siam J. Comput.* **5**(3), 488–521.