FACULTAD DE INFORMÁTICA

UNIVERSIDAD POLITÉCNICA DE MADRID

TESIS DE MÁSTER

MÁSTER DE INVESTIGACIÓN EN TECNOLOGÍAS PARA EL DESARROLLO DE SISTEMAS SOFTWARE COMPLEJOS

EXPLORING THE β-HYPERCUBE

AUTOR: Álvaro García Pérez
TUTOR: Pablo Nogueira Iglesias

SEPTIEMBRE, 2009

Abstract

Regarding higher-order functional languages, Reynolds noted that the semantics of the implementation language can interfere with the semantics of the object language. The same can be said about evaluation order in the context of evaluators. The culprit is the interpretation (or evaluation) of function application. To solve the problem, Reynolds proposed the use of continuation-passing style and defunctionalisation. Danvy extended Reynolds's work and showed that abstract machines can be derived from interpreters. In both cases, the object language is a typed extension of the lambda calculus with primitives and often the implementation language is call-by-value.

We systematically study and implement evaluators and interpreters for the pure lambda calculus in Haskell, a non-strict, call-by-need, purely functional language. We start with naive (and inaccurate) versions, discussing the, now important, issues of free variables and independence from Haskell's evaluation order. We present classic (non-monadic eval-apply and eval-case) as well as monadic evaluators that really implement appropriate semantics and evaluation orders by dint of special features such as strict application, continuations, and monads.

We present novel generic evaluators and interpreters that generalise all previous ones. Genericity is achieved by parametrisation and by mixin composition. We explore and reduce the parameter space, or β-hypercube as we like to call it, and discuss parameter minimalisation and its impact on expressiveness. To our knowledge, our work is the first systematic Haskell implementation of lambda calculus interpreters and evaluators and the first generalistaion in generic versions. Generic interpreters open up an interesting research path to explore: the development of generic (if you will, "pluggable") abstract machines.

Resumen

En relación a los lenguajes funcionales de orden superior, Reynolds advirtió que la semántica del lenguaje de implementación podría interferir con la semántica del lenguaje objeto. Lo mismo puede decirse sobre los ordenes de evaluación en el contexto de evaluadores. La interpretación (o evaluación) de la aplicación de funciones es la culpable. Para resolver éste problema Reynolds propuso el uso del estilo de paso por continuaciones y la defuncionalización. Danvy extendió el trabajo de Reynolds y mostró que máquinas abstractas se pueden derivar a partir de intérpretes. En ambos casos, el lenguaje objeto es una extensión tipada del cálculo lambda con primitivas añadidas y a menudo el lenguaje de implementación pasa los parámetros por valor (*call-by-value*).

Estudiamos e implementamos sistemáticamente evaluadores e intérpretes del cálculo lambda puro en Haskell, un lenguaje funcional puro, no estricto y con paso de parámetros *call-by-need*. Partimos de versiones ingenuas incorrectas, explicando aspectos ahora relevantes sobre variables libres así como la independencia del orden de evaluación de Haskell. Presentamos versiones clásicas (no monádicas *eval-apply* y *eval-case*) así como versiones monádicas que realmente implementan la semántica y ordenes de evaluación apropiados usando características especiales como aplicación estricta, continuaciones, y mónadas.

Presentamos novedosos evaluadores e intérpretes genéricos que generalizan todos los anteriores. La genericidad se consigue mediante la parametrización y mediante composición *mixin*. Exploramos y reducimos el espacio de parámetros, o β-hipercubo como nos gusta llamarlo, y discutimos la minimización de parámetros y su impacto en la expresividad. Hasta donde nosotros sabemos, nuestro trabajo es la primera implementación sistemática en Haskell de intérpretes y evaluadores del cálculo lambda y la primera generalización en versiones genéricas. Los intérpretes genéricos abren un campo de investigación interesante: el desarrollo de máquinas abstractas genéricas (o si nos permite, "enchufables").

Acknowledgements

Thanks to Nelson for encouraging me to do research and Juanjo and IMDEA for doing it possible. Thanks to Julio and Angel for sharing their points of view with me, Emilio for those endless conversations where something new is always learnt, Ana and Guillem for being always willing to give me a hand and the people from BABEL for their friendly support. Thanks to Bruno for a handful of mixins and monads, Jeremy and Richard for his suggestions on the cube and the people in the Algebra of Programming group for sparring me. Thanks to Pablo whose involvement greatly exceeded all his responsibilities as master and friend. Thanks to Eugenia for cheer me up everyday and my parents for their selfless worry and patience.

Contents

1. Introduction **1**
1.1. Motivation and background 1
1.2. Contributions 6
1.3. Related Work 7
1.4. Structure and organisation 8

2. The Untyped λ-calculus **11**
2.1. Syntax 11
2.2. Free and Bound Variables 12
2.3. α-conversion 14
2.4. β-reduction 14
2.5. η-reduction 16
2.6. Normal Forms 17
2.7. Strict and Non-strict Semantics 18
2.8. Evaluation Orders 18
2.9. Recursion 23
2.10. Illustrative λ-expressions used in the code 26

3. Evaluators for the Untyped λ-calculus **31**
3.1. Non-monadic Evaluators 31
3.1.1. Naive Evaluator 31
3.1.2. Evaluator Using Pattern Matching 34
3.1.3. Evaluator Using Strict Application 35
3.1.4. Evaluator Using Continuations 37
3.2. Monadic Evaluators 40
3.2.1. Monads 40
3.2.2. Basic Monadic Evaluator 41
3.2.3. Continuation Monad 43
3.3. Parametric Evaluator 44
3.3.1. Opportunities for Evaluation 45
3.3.2. Parameters and Evaluators 52

3.4. Compositional Evaluator 61
3.4.1. Mixins 61
3.4.2. Uniform Evaluation Orders with Mixins 62
3.4.3. Hybrid Evaluation Orders with Mixins 65

4. Interpreters for the λ-calculus **69**
4.1. Reynolds's Definitional Interpreters 69
4.2. Meta-Circular Interpreter 70
4.2.1. Programing with the MCI 72
4.3. MCI for the Pure λ-calculus 75
4.4. Interpreter with Free Variables 77
4.5. MCI with Applications of Free Variables 79
4.6. CPS MCI for Untyped λ-calculus 81
4.7. Monadic MCI for Untyped λ-calculus 83
4.8. Parametric MCI 85

5. Results and conclusions **89**

6. Future Work **91**

A. Illustrative λ-expressions **93**

B. Naive Evaluators **97**

C. Evaluators Using Pattern Matching **101**

D. Evaluators using the Strict Application Operator (`$!`) **105**

E. Evaluators using the Strict Sequencing Operator (`seq`) **109**

F. Evaluators using CPS **113**

G. Monadic Evaluators **117**

H. Parametric Evaluators **121**

Chapter 1

Introduction

This chapter serves several purposes:

- Provides the background and context of the work carried out in this thesis, introducing some key concepts and terminology.
- Lists motivations, objectives, and the *contributions* of the thesis.
- Puts the contributions in context by briefly discussing related work in the literature.
- Describes the structure and organisation of the thesis.

1.1. Motivation and background

The development of techniques and methods for dealing with complex software systems is of increasing interest. Of particular relevance is the study and development of advanced programming languages with solid foundations, a research area in which several research groups of the Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software are active.

Our interest in this thesis is the study of important foundational aspects of functional programming, a programming paradigm that was developed in academia with solid foundations and has been increasingly catching the attention of a wider audience, including mainstream practitioners and industry.

Our interest is the systematic study of the evaluation and interpretation of the λ-calculus, the core language of functional programming languages, and the development of generic evaluators and interpreters. The implementation language is Haskell, a popular non-strict, purely-functional language. As far as we know this is the first systematic study and implementation of evaluators and interpreters in Haskell that addresses issues such as free

variables in the pure λ-calculus, evaluation-order independence, and genericity via parametrisation and mixin composition. We explore the parameter space or, as we like to call it, the ***β-hypercube***, a name made in jest and moulded after the well-known λ-cube in type systems [16]. We have obtained several results (Section 1.2 and Chapter 5) and have found an interesting research path to explore: the development of general, or "pluggable", abstract machines from generic evaluators and interpreters (Chapter 6). Given the current transfer or adoption of functional programming ideas and languages to the mainstream, our foundational study and research results are of interest to the programming language community in general.

The following paragraphs elaborate in more detail.

Functional programming today. Functional programming languages have powerful mechanisms for parametrisation, abstraction, and reuse that make them ideal candidates for writing code that is more reliable, expressive, and easier to develop and maintain. Functional languages have been developed in academia with solid foundations. For several social reasons, they have not been widely adopted by practitioners and the software industry in general, until now. Impressive theoretical and practical advances that have taken place in the last decade have made functional languages more powerful and expressive and have caught the attention of the software industry. It is commonplace that concepts that originated in the functional paradigm have been adopted by popular languages such as Java or C#. Examples are first-class functions, parametric polymorphism, garbage collection, λ-expressions, and query comprehension. But industry's trend is to go functional. Conspicuous examples are Python, Microsoft's F#, and the growing popularity of Haskell and OCaml. The advent of multicore computing and parallelism makes functional languages not only attractive, but thanks to their powerful abstraction mechanisms and their careful treatment and static control of side-effects, they are perhaps the only reasonable choice [15].

Foundations: evaluation and interpretation in the λ-calculus. The study of foundational aspects of functional programming languages started well before computers existed. Gödel himself developed a functional language to prove his famous incompleteness theorem. Alonzo Church invented the λ-calculus in the 1930s [4] which constitutes the core language of functional programming languages. Scott and Strachey [20] introduced a formal denotational semantics for programming languages, including functional ones based on the λ-calculus. The works of Landin [9], Reynolds [19], and Plotkin [17] settled the basis for the interpretation of the λ-calculus laying the foundations

of denotational and operational semantics. Finally, Danvy [7, 1] has given the operational semantics of different evaluation orders of the λ-calculus, translating the interpreters into abstract machines.

We aim to make a contribution in this research line, systematically studying and implementing evaluators and interpreters for the pure λ-calculus in Haskell—most authors cited have only focused on typed extensions with primitives and in call-by-value implementation languages. We present novel generic evaluators and interpreters by means of parametrisation and mixin composition.

Haskell is our implementation language. Haskell is an advanced non-strict and purely functional programming language designed to be suitable for teaching, research, and for building large systems. Its design emphasises evolution, with more than twenty years of cutting-edge research attesting to its maturity. It is open and free, and is the preferred option for many researchers. Although created for and by academia, it has become a powerful production language, with its stable Haskell 98 version, with the state-of-the-art Glasgow Haskell Compiler, and with more than 1000 libraries.

Haskell is a good choice for developing λ-calculus interpreters and evaluators. It supports prominent features such as algebraic data types, pattern matching, parametric polymorphism, type classes, monads, graphics, concurrency, foreign-function interfaces, etc. Haskell provides a definition style with guards and `where` clauses as well as an expression style with pattern matching and `let` clauses. Haskell eases the intensive use of monads, which provide a clean mechanism to manage sequencing and side effects. For the monadic versions, we use Haskell's `do` notation.

Haskell has non-strict semantics and call-by-*need* (CBD) evaluation order, that is, call-by-name with sharing of sub-expressions which are evaluated once and on demand. Nevertheless, Haskell provides strictness (call-by-value) primitives.

Strictness and evaluation order. Strictness is a semantic concept related to function application. More precisely, a function is ***strict*** on an argument if the result of the application is undefined (e.g., non-terminating) when the argument is undefined. The function is ***non-strict*** if the result is defined when the argument is undefined. Strictness is related to argument usage. Typical examples of non-strict functions are the constant function and conditionals, the latter non-strict on their last two arguments.

An ***evaluation order*** specifies the order in which expressions of the language are evaluated. Reynolds uses the term "application order" per-

haps emphasising the common use of "evaluation order" in reference to function-application expressions (e.g., call-by-X). For example, in call-by-value (CBV) the argument is evaluated before evaluating the application whereas in call-by-name (CBN) arguments are passed unevaluated. A reduction order is a related concept in reduction systems (Chapter 2).

An evaluation order implements a particular semantics. For example, CBV implements a strict semantics whereas CBN a non-strict semantics.

Interpreters vs evaluators. In the context of the present work, an ***interpreter*** is a computable function written in an ***implementation language*** which gives meaning in a ***value*** domain to an element in its source domain, the latter a data type encoding the syntax of an ***object language***. (Reynolds [19] used the terms *defining language* and *defined language* but we prefer to use more clear-cut terminology.) A value domain for a functional language is typically defined thus:

$$\mathit{Value} ::= \mathit{Base} \mid \mathit{Value} \rightarrow \mathit{Value}$$

where *Base* is non-functional primitive data in the implementation language and $\mathit{Value} \rightarrow \mathit{Value}$ is an endofunction in the value domain. An interpreter provides a denotational semantics for the object language, as it assigns a meaning (denotation) to any program of the former. An interesting aspect of denotational semantics is computationally: the meaning of an expression is a composition of the meaning of its subexpressions. An important assumption is that the implementation language's semantics is well understood. When the object language is a desugared version of the implementation language then, following Reynolds [19], we call this a ***meta-circular interpreter***, for it defines the denotational semantics of a language in terms of "itself".

An ***evaluator*** is a computable function which evaluates expressions in the object language to other expressions in the same language. The source and value domain coincide. An evaluator indirectly provides a (big-step) operational semantics [16]. Computation in functional programming languages amounts to the evaluation of functional applications. The value domain is the subset of object-language expressions that cannot be evaluated further. Some authors use *reducer* instead of evaluator, for evaluation is related to the concept of reduction. We use *reducer* and *reduction* in the context of a mathematical reduction relation induced by some *notion of reduction* (Sections 2.4 and 2.8).

The distinction between interpreter and evaluator is subtle and not clear-cut. Often values are somewhat shared by the implementation and object languages, for example, integers may be values in the implementation language and be embedded in the type encoding the object language. In this case

an interpreter can be seen as a convoluted evaluator that evaluates expressions by translating them on-the-fly to implementation-language expressions and evaluating them within the implementation language.

The implementation language gets in the way. An interpreter gives denotational meaning to an object language. An evaluator operational meaning. If the object language is a desugared version of the implementation language we have a meta-circular interpreter or evaluator. Regarding higher-order functional languages, Reynolds [19] noted that the semantics of the implementation language can "interfere" with the semantics of the object language. The same can be said about evaluation order in the context of evaluators. The culprit is the interpretation (or evaluation) of function application. For example, if not careful, the non-strict semantics (or CBN evaluation order) in the implementation language will be "carried over" to a desirably strict and CBV object language

To solve the problem, Reynolds proposed the following. The use of continuation-passing style (CPS), which makes the semantics given by the interpreter independent of the evaluation order in the implementation language, and "defunctionalization", which converts the different uses of higher-order functions to instances of a data type, turning a higher order interpreter into a first-order one. Combining these transformations he gives four different meta-circular interpreters: a naive one, an evaluation-order-independent one, a first-order one, and a first-order evaluation-order-independent one.

CPS is a functional programming style where the control of the program is passed explicitly in the form of a continuation argument to functions. A continuation is a function which represents what to do next in a computation. The continuation takes the result of the computation at the current point and computes the final result from this partial result.

Using continuations, Plotkin [17] proved that is possible to simulate CBV in a CBN language and vice versa.

The pure lambda calculus. The object language of study is the pure or untyped lambda calculus (λ-calculus for short). The λ-calculus and its extensions form the core of present-day functional programming languages. These languages provide mechanisms for function definition and application, computation amounts to the evaluation of applications, and functions are first-class values. In particular, functions can be higher-order, that is, passed to or returned by other functions as well as be part of data types. Church introduced in 1932 the untyped λ-calculus as part of his investigation in the foundation of mathematics. He published in 1936 the portion relevant to

computation. The standard reference is [2].

The λ-calculus supports the definition of anonymous functions using the eponymous λ. The expression $\lambda x.B$ defines a function where the variable x is abstracted as a formal parameter of λ-expression B. Functional application is written $M\ N$, where M is a λ-expression called operator and N is its argument.

The λ-calculus has *free* variables (Section 2.2) which makes its denotational semantics and evaluation more involved. Most researchers have focused on an extended λ-calculus with primitives (Reynolds, Plotkin, Danvy, etc) because they are interested in semantics of full-fledged functional languages. In this work, we focus on the pure λ-calculus and study evaluators and interpreters under the presence of free variables.

Abstract Machines. A machine is typically expressed as a state transition system determined by a language of states, an initial state, a set of final states and a transition function from states to states. A machine provides semantics for a programming language if we consider states containing expressions of the language as well as other control data. Then any expression can be interpreted by running the machine over it, and taking the final state as the value denoted by the expression. Not every definitional interpreter is a machine. The machine establishes fixed semantics for the language and do not use higher-order functions. Reynolds [19] used CPS transformation to obtain an interpreter with fixed semantics and defunctionalization to get rid of the higher order. Danvy [7, 1] refined these techniques and largely studied the correspondence between definitional interpreters and abstract machines. Following Danvy, we will call abstract machines those that operate directly on the terms of the target language, without needing an instruction set.

Landin [9] introduced the SECD machine as a mechanisation of the process of evaluating λ-calculus expressions. Danvy [7] used techniques related to CPS transformation and defunctionalization to derive an abstract machine from any interpreter. In [1] he revealed the denotational content of the SECD machine, which implements CBV. He also showed the equivalence of Krivine's machine to a CBN interpreter and the CEK machine to a CBV. He classified the interpreters and the abstract machines derived from them.

1.2. Contributions

The contributions of this thesis are:

1. We discuss and implement in Haskell evaluators for the untyped λ-calculus that realise the foremost evaluation orders (AOR, NOR, CBN,

and CBV, among others). We start with naive (and inaccurate) versions, discussing issues of free variables and independence from Haskell's evaluation order. We then present classic (non-monadic, eval-case) as well as monadic evaluators that really implement appropriate evaluation orders by dint of special features such as strict application, continuations, and monads. We address and discuss the difficulties of overriding the implementation language's evaluation order when implementing an evaluator for a higher-order language, particularly in the context of non-strict Haskell.

2. We present novel generic evaluators which generalise all specific ones and are capable of implementing all evaluation orders. Genericity is achieved by parametrisation. Generic evaluators are higher-order functions whose parameters factor-out the distinctive bits of behaviour of specific evaluators (evaluation under lambda, evaluation of parameters in application, etc). Particular evaluators can be defined as fixed points of generic ones. We present monadic and non-monadic versions of generic evaluators and investigate the parameter space or β-hypercube, as we like to call it. A finite definition of this space contains the salient evaluation orders. An infinite definition of a similar space is given by means of mutually recursive fixed points. It is possible to define evaluators that act on particular subexpressions only. Hybrid evaluators are presented as combinations of the former. We discuss parameter minimalisation and its impact on expressiveness.

3. We present a novel generic monadic evaluator where genericity is achieved by means of mixins [5]. Mixins let us define elementary components which isolate distinctive bits of behaviour so that particular evaluation orders are defined by composing particular components.

4. We present versions of Reynolds interpreters [19] for the pure λ-calculus and a generic-parametric version that abstracts out the evaluation order.

5. We discuss the possibility of deriving parametric (or 'pluggable') abstract machines from parametrised interpreters using the techniques of Reynolds [19] and Danvy[1].

1.3. Related Work

Reynolds [19] first introduced techniques to properly implement an evaluation order when writing an interpreter for a higher-order functional lan-

guage. We adapt Reynolds's interpreter to the pure λ-calculus. Using his techniques we make the interpreter evaluation-order independent. After that we generalise the interpreter (like we do with evaluators) and propose a parametric one which can be instantiated to implement multiple evaluation orders.

Plotkin [17] gave formal proofs of the validity of Reynolds's CPS to solve the evaluation-order independence problem. He simulated CBN under CBV and vice-versa. We simulate any evaluation order in Haskell, which is call-by-need.

Landin [9] first described an abstract machine for λ-calculus, which implemented a CBV strategy. We describe any order of evaluation by a single parametric interpreter.

Danvy [7] explored the direct and inverse CPS transformation. He also classified many interpreters with different evaluation orders and gave their transformation to abstract machines [1]. We hope to find the transformation of our parametric interpreter into an abstract machine. Similarly with our evaluators.

Sestoft [21] described and implemented a variety of evaluation orders, including the hybrid ones. We supersede those implementations by a unique parametric evaluator, which fits uniform as well as hybrid evaluation orders. Sestoft used SML, a strict language, whereas we use non-strict Haskell. Implementation issues are significantly different.

1.4. Structure and organisation

Chapter 2 overviews the pure λ-calculus with emphasis on substitution, reduction rules, reduction orders, normal forms and recursion.

Chapter 3 shows the implementation of λ-calculus evaluators for the foremost evaluation orders in non-strict Haskell, discussing the complications posed by free variables and Haskell's non-strict semantics. In particular:

Section 3.1 shows and discusses non-monadic evaluators, starting from naive and inaccurate versions to accurate versions that really implement appropriate evaluation orders, making use of diverse features and techniques, from Haskell's strict application to pattern matching and continuations.

Section 3.2 shows monadic versions after introducing the monad concept and its positive impact in code structure. The evaluators de-

fined are universally quantified on a monad type. We show several appropriate monad instances for different evaluation orders.

Section 3.3 shows the parametric evaluators (single and mutually recursive versions) and discusses parameter orthogonality. The parametric evaluator defines a parametrisation space we call the β-hypercube. The number of dimensions in this cube is minimised discussing the impact on expressivity.

Section 3.4 shows compositional evaluators using mixins. We define elemental components for each feature and give every evaluation order as a composition of elemental components.

Chapter 4 shows and discusses interpreters for the λ-calculus overviewing previous work by Reynolds and commenting on the choice of model. We show interpreters for different evaluation orders that use environments, and also show a parametric interpreter. We discuss the relationships with the β-hypercube.

Chapter 5 summarises results and draws the relevant conclusions.

Chapter 6 discusses future work, in particular the development of pluggable abstract machines and the applicability of strategic programming techniques to evaluators and interpreters.

Appendices: lists the code for all evaluators and interpreters.

Chapter 2

The Untyped λ-calculus

This chapter overviews the pure λ-calculus with emphasis on substitution, reduction rules, reduction orders, normal forms and recursion.

2.1. Syntax

Definition 2.1. The set of ***λ-expressions*** Λ is defined by the following grammar:

$$
\begin{array}{lll}
\Lambda ::= & V & \text{Variables.} \\
\quad\mid & \lambda V.\,\Lambda & \lambda\text{−abstractions.} \\
\quad\mid & \Lambda\,\Lambda & \lambda\text{−applications.} \\
\quad\mid & (\Lambda) & \text{Grouping.}
\end{array}
$$

where V is an infinite set of variables $V ::= x \mid y \mid z \mid \ldots$

A ***variable*** is a λ-expression. If x is a variable and B is a λ-expression then $\lambda x.B$ is a λ-expression, known as ***λ-abstraction***, where x is the ***formal parameter*** and B is the ***body***. λ-abstractions only take one formal parameter. Currying is used to write abstractions with more than one parameter [2]. Parenthesis enclosing the body of a λ-abstraction are omitted. The body of a λ-abstraction goes from the dot to the end of the λ-expression (or a balanced left parenthesis enclosing the whole λ-abstraction).

If M and N are valid λ-expressions then $M\,N$ (note the white space between the two expressions) is a valid λ-expression known as a ***λ-application***. By convention, application is a left-associative operation and parenthesis may be omitted for multiple application. That is, $M\,N\,O$ stands for $(M\,N)\,O$. We also call a λ-expression a ***λ-term***.

We use lowercase letters (x, y, etc.) for variables and uppercase letters (M, N, etc) for λ-expressions. Syntactic equivalence is denoted by the symbol $\equiv$.

The set of λ-expressions is represented in Haskell straightforwardly by the following algebraic data type. Variables are encoded by strings instead of characters to account for infinite of them. We include the code for a pretty printer that shows λ-expressions as strings in more mathematical syntax.

```
data SLTerm = Var String
            | Lam String SLTerm
            | App SLTerm SLTerm
              deriving Eq

instance Show SLTerm where
    show (Var s)                 = s
    show (Lam s e)               = "\\" ++ s ++ "." ++ show e
    show (App (Var s1) (Var s2)) = s1 ++ " " ++ s2
    show (App (Var s1) e)        = s1 ++ "(" ++ show e ++ ")"
    show (App e        (Var s2)) = "(" ++ show e ++ ")" ++ s2
    show (App e1       e2)
        = "(" ++ show e1 ++ ")(" ++ show e2 ++ ")"
```

For example, `show (Lam "x" (App (Var "x") (Var "y")))` delivers `\x.xy` and typing the λ-expression in the Haskell interpreter (Glasgow Haskell Compiler Interpreter or GHCi) delivers the same result:

```
*TermsPool> Lam "x" (App (Var "x") (Var "y"))
\x.xy
```

2.2. Free and Bound Variables

A bound variable is another way to refer to a formal parameter of a λ-abstraction. It is a placeholder for the formal parameter, which may occur in its body and the argument passed to that λ-abstraction will be substituted for it. A free variable, in contrast, does not appear bound in any λ-abstraction. It stands for a primitive, this is, the expression where it appears does not prescribe the meaning for that variable, which can stand for whatever one. For a λ-abstraction, the occurrences of a bound variable between the λ and the dot are said to be ***binding occurrences***, where the occurrences in the body are known as ***applied occurrences***.

Definition 2.2. A ***free variable*** is one which does not have any binding occurrence in any enclosing λ-abstraction. A ***bound variable*** is one which has a binding occurrence in some λ-abstraction enclosing it.

A λ-expression containing free variables is called an ***open λ-expression***, otherwise it is called a ***closed λ-expression***.

Definition 2.3. $FV(E)$ is the set of the free variables of E and $BV(E)$ is the set of bound variables of E.

We use the functions

```
import Data.List

freeVars :: SLTerm -> [String]
freeVars (Var x)   = [x]
freeVars (Lam x b) = delete x $ nub (freeVars b)
freeVars (App m n) =
    (nub $ freeVars m) `union` (nub $ freeVars n)

boundVars :: SLTerm -> [String]
boundVars (Var x)   = []
boundVars (Lam x b) = nub $ x : boundVars b
boundVars (App m n) =
    (nub $ boundVars m) `union` (nub $ boundVars n)
```

to compute FV and BV in Haskell.

The same name can be used for a free and a bound variable in a λ-expression, but in different lexical scopes. A λ-abstraction defines a ***lexical scope*** where its bound variables are local to that scope. In a particular lexical scope different variables must have different names. There is a global scope where free variables live. The variable x is both free and bound in $(\lambda x.\lambda y.x)\,x$ (free in the last occurrence of it and bound in the binding occurrence and the applied occurrence of the λ-abstraction inside the parenthesis).

Lexical scopes can be nested the same way λ-abstractions are. Notice that the same name could appear in nested lexical scopes, like in $(\lambda x.(\lambda x.x)\,y\,x)\,z$. In this case the binding occurrence of a bound variable is the one in the nearest λ-abstraction binding it. Thus, the x in the outer λ-abstraction will be substituted by the argument z, whereas the x in the inner one will be substituted by the argument y.

2.3. α-conversion

One expression is equivalent to the same expression after the renaming of some of its bound variables. For example, in $(\lambda x.x)\, z$, the variable x, which is bound, can be renamed to y, resulting $(\lambda y.y)\, z \equiv (\lambda x.x)\, z$.

Definition 2.4. M, N are ***α-congruent*** if N can be obtained after the renaming of the bound variables of M. The process of renaming the bound variables of a λ-expression is called ***α-conversion***.

The two expressions are equivalent in the sense that both convert to the same expression when applying reduction rules (see Section 2.4). We consider α-conversion at the syntactic level, rather than as a reduction rule. This extends syntactic equivalence ($\equiv$) up to α-congruence.

2.4. β-reduction

A notion of reduction is a binary relation on Λ [2].

Definition 2.5. A binary relation R on Λ is ***compatible*** when

$$\frac{M, N \in R}{\lambda x.M, \lambda x.N \in R}\ ,\quad \frac{M, N \in R}{Z\,M, Z\,N \in R}\ ,\quad \frac{M, N \in R}{M\,Z, M\,Z \in R}$$

A notion of reduction R induces the following binary relations: one-step R-reduction ($\stackrel{R}{\rightarrow}$), R-reduction ($\stackrel{R\star}{\rightarrow}$), and R-equality ($\stackrel{R}{=}$).

Definition 2.6. The relation $\stackrel{R}{\rightarrow}$ is the compatible closure of R:

$$\frac{M, N \in R}{M \stackrel{R}{\rightarrow} N}\ ,\quad \frac{M \stackrel{R}{\rightarrow} N}{\lambda x.M \stackrel{R}{\rightarrow} \lambda x.N}\ ,\quad \frac{M \stackrel{R}{\rightarrow} N}{Z\,M \stackrel{R}{\rightarrow} Z\,N}\ ,\quad \frac{M \stackrel{R}{\rightarrow} N}{M\,Z \stackrel{R}{\rightarrow} N\,Z}$$

Definition 2.7. The relation $\stackrel{R\star}{\rightarrow}$ is the reflexive and transitive closure of $\stackrel{R}{\rightarrow}$:

$$\frac{M \stackrel{R}{\rightarrow} N}{M \stackrel{R\star}{\rightarrow} N}\ ,\quad M \stackrel{R\star}{\rightarrow} M\ ,\quad \frac{N \stackrel{R\star}{\rightarrow} L}{M \stackrel{R\star}{\rightarrow} L}$$

Definition 2.8. The relation $\stackrel{R}{=}$ is the equivalence relation generated by $\stackrel{R\star}{\rightarrow}$:

$$\frac{M \stackrel{R\star}{\rightarrow} N}{M \stackrel{R}{=} N}\ ,\quad \frac{M \stackrel{R}{=} N}{N \stackrel{R}{=} M}\ ,\quad \frac{M \stackrel{R}{=} N, N \stackrel{R}{=} L}{N \stackrel{R}{=} L}$$

A ***contraction rule*** is a rule schema prescribing that the expressions matching its left part must be contracted into the right part. A notion of reduction is given by the contraction rule:

$$(\lambda x.B)\,N \to B[x/N]$$

where $B[x/N]$ is the substitution of N for x in B.

Some objections have to be made with respect to this substitution. Consider the expression

$$F \equiv (\lambda x.\lambda y.y\,x)$$

then, reducing $F\,M\,N$ we obtain

$$\begin{array}{lcl} F\,M\,N & \equiv & (\lambda x.\lambda y.y\,x)\,M\,N \\ & \to & (\lambda y.y\,M)\,N \\ & \to & N\,M \end{array}$$

so, $F\,M\,N \to N\,M$ for any M, N. This statement is fallacious, since it is not true with $M \equiv y$ and $N \equiv x$, as showed by the following reduction sequence.

$$\begin{array}{lcl} F\,y\,x & \equiv & (\lambda x.\lambda y.y\,x)\,y\,x \\ & \to & (\lambda y.y\,y)\,x \\ & \to & x\,x \end{array}$$

What is happening here is that the substitution of the y for the x in the first reduction step makes the free variable y to be captured by the binding occurrence of y, so in the second step that y will be substituted by the parameter passed to the λ-abstraction, resulting $x\,x$. We need a substitution operation which does not capture variables.

Definition 2.9. The ***capture avoiding substitution*** $Z[x/E]_{ca}$ of x for E in Z is defined by cases on Z:

$$\begin{array}{lcll} x[x/E] & = & E & \\ y[x/E] & = & y & \\ (MN)[x/E] & = & M[x/E]\ N[x/E] & \\ (\lambda x.B)[x/E] & = & \lambda x.B & \\ (\lambda y.B)[x/E] & = & \lambda y.B[x/E] & \text{if } y \notin \mathrm{FV}(E) \\ (\lambda y.B)[x/E] & = & \lambda z.B[y/z][x/E] & \text{if } y \in \mathrm{FV}(E),\ \text{picking fresh } z \notin \mathrm{FV}(E) \cap \mathrm{FV}(B) \end{array}$$

We show its Haskell implementation:

```
subst :: String -> SLTerm -> SLTerm -> SLTerm
subst x v@(Var s)   exp = if s == x then exp else v
subst x l@(Lam s b) exp =
   if x == s then
        l
    else
        if not $ elem s $ freeVars exp then
            (Lam s (subst x b exp))
        else
            let z = fresh s $
                    union (freeVars exp) (freeVars b)
            in Lam z (subst x (subst s b (Var z)) exp)
subst x (App m n)   exp =
    App (subst x m exp) (subst x n exp)
```

Definition 2.10. The ***β-rule*** is the contraction rule

$$(\lambda x.B)\,N \rightarrow B[x/N]_{ca}$$

The β-rule defines the one-step β-reduction relation ($\xrightarrow{\beta}$), the β-reduction relation ($\xrightarrow{\beta\star}$) (also known as full β-reduction) and the β-conversion ($\overset{\beta}{=}$). The β-rule is the analogous of the mathematical function application inside the λ-calculus. The one-step β-reduction and the β-reduction are the mechanisms to compute something within the λ-calculus. The β-conversion defines the conversion equivalence under function application.

Definition 2.11. A ***reducible expression*** (redex) in an expression M is any of the subexpressions of M with the form $(\lambda x.B)\,N$.

The redexes are the subexpressions to which we can apply the β-rule. An expression may contain several redexes, for example:

$$\underline{(\lambda x.\underline{(\lambda y.y)x})\,(\underline{(\lambda x.x)z})}$$

where the redexes are underlined.

2.5. η-reduction

There is another contraction rule called the η-rule.

Definition 2.12. The ***η-rule*** is the contraction rule

$$(\lambda x.B\,x) \rightarrow B, \quad \text{provided } x \notin FV(B)$$

We use the equivalence relation $\overset{\beta\eta}{=}$ induced by β and η when exposing recursion with the fixed point combinators, but we don't consider the η-reduction in our interpreters and evaluators. In Section 2.9 we abuse notation with $(\lambda x.B\,x) \overset{\beta\eta}{=} B$ meaning the following:

$$\frac{(\lambda x.B\,x)\,M \overset{\beta\star}{\to} N}{B\,M \overset{\beta\star}{\to} N}$$

2.6. Normal Forms

When an expression cannot be reduced any further we say that it is in normal form.

Definition 2.13. An expression E is in ***normal form*** (NF) iff:

- E is of the form $\lambda x.N$ where N is in NF.
- E is of the form $x\,N_1\,N_2\,\ldots\,N_n$ where N_i are in NF and $n \geq 0$.

There are weaker notions of normal form.

Definition 2.14. An expression E is in ***weak normal form*** (WNF) iff:

- E is of the form $\lambda x.B$ with arbitrary B.
- E is of the form $x\,W_1\,W_2\,\ldots\,W_n$ where W_i are in WNF and $n \geq 0$.

Definition 2.15. An expression E is in ***head normal form*** (HNF) iff:

- E is of the form $\lambda x.H$ where H is in HNF.
- E is of the form $x\,M_1\,M_2\,\ldots\,M_n$ with arbitrary M_i and $n \geq 0$.

Definition 2.16. An expression E is in ***weak head normal form*** (WHNF) iff:

- E is of the form $\lambda x.B$.
- E is of the form $x\,M_1\,M_2\,\ldots\,M_n$ where $n \geq 0$.

2.7. Strict and Non-strict Semantics

Denotational semantics is the study of the assignment of meanings to expressions. Meanings are values in an abstract and mathematical domain which is called a ***semantic domain***. In functional languages, semantic domains are complete partial orders (CPOs), a mathematical domain with a least element called ***bottom*** ($\perp$) that is the value of undefined expressions, and a partial approximation or *less-defined-than* relation [22]. In the untyped λ-calculus, like in functional languages, undefined expressions are those without normal form, this is, expressions which an infinite reduction sequence.

Definition 2.17. A semantic function $[\![\ ]\!]$ is a mathematical function which takes an expression in the object language and gives a value in any semantic domain.

Definition 2.18. A function is ***strict*** on an argument if the result of the application is undefined (e.g., non-terminating) when the argument is undefined. The function is ***non-strict*** if the result is defined when the argument is undefined. Strictness is related to argument usage.

The notion of strictness applies to the meaning of redexes in the λ-calculus. There are λ-abstractions whose reduction does not terminate in any reduction order. Self-application is used in the typical example $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. We say $[\![M]\!] = \perp$ when M's reduction does not terminate in any reduction order. Abusing language, we say $\lambda x.B$ is strict on x when $[\![(\lambda x.B)]\!]\perp = \perp$

2.8. Evaluation Orders

The compatibility of β-reduction entails the recursive application of the β-rule over every redex of an expression, but the mathematical notion of β-reduction does not specify which redex should be reduced first inside an expression with multiple redexes. This makes the direct implementation of the full β-reduction impractical. For that reason an evaluation order is needed that specifies which redex has to be reduced.

Definition 2.19. An ***evaluation order*** is a non-compatible reduction relation, which selects a particular redex in the expression being reduced and applies the β-rule to it.

We will describe the evaluation strategies using ***big-step*** semantics, this is, the transition rules are specified describing the final result of the reduction

relation, instead of the immediate result delivered by a single transition, as done in ***small-step*** semantics [16].

When evaluating an expression, we can traverse the syntactic tree in several ways. If we select the leftmost-innermost redex first, the expression $(\lambda x.(\lambda y.y)\,x)\,((\lambda x.x)\,z)$ would be reduced in the following way.

$$\begin{aligned}(\lambda x.\underline{(\lambda y.y)\,x})\,((\lambda x.x)\,z) &\overset{\beta}{\to} (\lambda x.x)\,((\lambda x.x)\,z)\\ (\lambda x.x)\,(\underline{(\lambda x.x)\,z}) &\overset{\beta}{\to} (\lambda x.x)\,z\\ \underline{(\lambda x.x)\,z} &\overset{\beta}{\to} z\end{aligned}$$

In this reduction sequence we reach the expression z which does not match the head of the β-rule hence it cannot be reduced any further. Leftmost-innermost is called applicative order [2].

Definition 2.20. ***Applicative order*** (AOR) is the evaluation order which reduces first the leftmost-innermost redex, defined by the following natural deduction rules:

$$x \overset{aor}{\to} x \quad , \quad \frac{B \overset{aor}{\to} B'}{\lambda x.B \overset{aor}{\to} \lambda x.B'} \quad ,$$

$$\frac{M \overset{aor}{\to} \lambda x.B \qquad N \overset{aor}{\to} N' \qquad B[x/N']_{ca} \overset{aor}{\to} E}{M\,N \overset{aor}{\to} E} \quad ,$$

$$\frac{M \overset{aor}{\to} M' \not\equiv \lambda x.B \qquad N \overset{aor}{\to} N'}{M\;N \overset{aor}{\to} M'\;N'}$$

Applicative order is said to be an eager evaluation order, because the argument of an abstraction is valuated as soon as possible, substituting its reduced form in the body of the abstraction. Another eager evaluation order is possible, if we do not consider reducing the body of unapplied lambda abstractions. This is called ***call-by-value*** (CBV).

Definition 2.21. ***Call-by-value*** (CBV) is the evaluation order defined by the rules:

$$x \overset{cbv}{\to} x \quad , \quad \lambda x.B \overset{cbv}{\to} \lambda x.B \quad ,$$

$$\frac{M \overset{cbv}{\to} \lambda x.B \qquad N \overset{cbv}{\to} N' \qquad B[x/N']_{ca} \overset{cbv}{\to} E}{M\,N \overset{cbv}{\to} E} \quad ,$$

$$\frac{M \overset{cbv}{\to} M' \not\equiv \lambda x.B \qquad N \overset{cbv}{\to} N'}{M\;N \overset{cbv}{\to} M'\;N'}$$

AOR and CBV differ in the former evaluates the body of λ-abstractions whereas the latter does not. Those evaluation orders consider the case where the operand in an λ-application is a variable or an application of irreducible expressions. This is so because the λ-calculus contains open λ-expressions, with free variables which does not stand for any formal parameter of a λ-abstraction. An interpreter could forbid this situation, and treat always with closed λ-expressions, whose denotational content is always determined by the expression. Alternative it would force the use of an environment binding every free variable in the λ-expression. None of these are our case, where we just evaluate λ-expressions to some notion of irreducible expression (Section 2.6). This applies also to the remaining evaluation orders we are describing in this section.

We can classify the evaluation orders attending to the fact if they implement strict or non-strict semantics. Evaluation orders imposing strict semantics evaluate arguments before substituting them in the body of the abstractions, whereas the ones imposing non-strict semantics substitute the unevaluated form of the argument in the body and then evaluate the resulting expression.

The evaluation order implementing non-strict semantics dual to CBV is known as call-by-name (CBN)

Definition 2.22. ***Call-by-name*** (CBN) is the evaluation order following the rules:

$$x \overset{cbn}{\to} x \quad , \quad \lambda x.B \overset{cbn}{\to} \lambda x.B \quad ,$$

$$\frac{M \overset{cbn}{\to} \lambda x.B \qquad B[x/N]_{ca} \overset{cbn}{\to} E}{M\,N \overset{cbn}{\to} E} \quad ,$$

$$\frac{M \overset{cbn}{\to} M' \not\equiv \lambda x.B}{M\ N \overset{cbn}{\to} M'\ N'}$$

There is an evaluation order imposing non-strict semantics that reduces the bodies of unapplied λ-abstractions. It is called ***normal order*** (NOR) and is an hybrid evaluation order which uses CBN to evaluate the operator in a λ-application. NOR corresponds to a leftmost-outermost strategy.

Definition 2.23. ***Normal order*** (NOR) is the evaluation order which re-

duces first the leftmost-outermost redex, according to the rules:

$$x \overset{nor}{\to} x; \;,\; \frac{B \overset{nor}{\to} B'}{\lambda x.B \overset{nor}{\to} \lambda x.B'} \;,$$

$$\frac{M \overset{cbn}{\to} \lambda x.B \qquad B[x/N]_{ca} \overset{nor}{\to} E}{M\,N \overset{nor}{\to} E} \;,$$

$$\frac{M \overset{cbn}{\to} M' \not\equiv \lambda x.B \qquad M' \overset{nor}{\to} M' \qquad N \overset{nor}{\to} N'}{M\ N \overset{nor}{\to} M''\ N'}$$

Some authors consider also the ***head spine*** evaluation [21]. This is an order that evaluates λ-abstractions but only when they are in head position.

Definition 2.24. If M is subexpression of N then we say M is in ***head position*** when $M = N$. For example, there is a λ-abstraction in head position in the expression $\lambda x.(x\,y)\,z$.

Definition 2.25. ***Head spine*** evaluation (HE) is the evaluation order which reduces the body of λ-abstractions when they are in head position. The rules defining it are:

$$x \overset{he}{\to} x \;\;,\;\; \frac{B \overset{he}{\to} B'}{\lambda x.B \overset{he}{\to} \lambda x.B'} \;,$$

$$\frac{M \overset{he}{\to} \lambda x.B \qquad B[x/N]_{ca} \overset{he}{\to} E}{M\,N \overset{he}{\to} E} \;,$$

$$\frac{M \overset{he}{\to} M' \not\equiv \lambda x.B}{M\ N \overset{he}{\to} M'\ N}$$

CBV, AOR, CBN and HE are called ***uniform*** evaluation orders, because they can be defined recursively in terms of themselves. The ***hybrid*** evaluation orders, in contrast, are defined in terms of some others, like NOR, which is defined in terms of CBN and itself. Other hybrid evaluation orders exist in the literature, like the ***hybrid applicative order*** (HA) and the ***hybrid NOR*** (HN) [21].

HA is an order similar to AOR where the operator of a λ-application is evaluated with CBV. It implements strict semantics, but no that strict than the ones implemented by AOR. This means that there are some expressions whose evaluation with HA terminates, but they evaluate to undefined with AOR. In particular, an expression containing an undefined argument

in the body of a λ-abstraction in the operator position of a λ-application may terminate when evaluating it with HA. This makes possible the evaluation of recursive functions to NF using the Z fixed point combinator (see Section 2.9).

Definition 2.26. ***Hybrid applicative order*** (HA) is the evaluation order according to the rules:

$$x \overset{ha}{\to} x \quad , \quad \frac{B \overset{ha}{\to} B'}{\lambda x.B \overset{ha}{\to} \lambda x.B'} \quad ,$$

$$\frac{M \overset{cbv}{\to} \lambda x.B \qquad B[x/N]_{ca} \overset{ha}{\to} E}{M\,N \overset{ha}{\to} E} \quad ,$$

$$\frac{M \overset{cbv}{\to} M' \not\equiv \lambda x.B \qquad M' \overset{ha}{\to} M' \qquad N \overset{ha}{\to} N'}{M\ N \overset{ha}{\to} M''\ N'}$$

HN is the analogous to NOR but using HE to evaluate the operator of a λ-application.

Definition 2.27. ***Hybrid NOR*** (HN) is the evaluation order according to the rules:

$$x \overset{hn}{\to} x \quad , \quad \frac{B \overset{hn}{\to} B'}{\lambda x.B \overset{hn}{\to} \lambda x.B'} \quad ,$$

$$\frac{M \overset{he}{\to} \lambda x.B \qquad B[x/N]_{ca} \overset{hn}{\to} E}{M\,N \overset{hn}{\to} E} \quad ,$$

$$\frac{M \overset{hn}{\to} M' \not\equiv \lambda x.B \qquad M' \overset{hn}{\to} M' \qquad N \overset{hn}{\to} N'}{M\ N \overset{hn}{\to} M''\ N'}$$

Another well-known evaluation order is ***call-by-need*** (CBD). CBD is a strategy where each redex is reduced at most once. It needs *sharing* and *memoization*. This strategy is based on graph reduction and will not be covered in this thesis. CBD is the strategy used in Haskell, which will be our defining language. It will be enough to consider that it implements non-strict semantics.

CBV, AOR and HA implement strict semantics, they are said to be ***eager***, whereas CBN, NOR, HE, HN and CBD implement non-strict ones. CBD only evaluates each redex at most once, therefore it is called ***lazy*** evaluation.

Normal forms and evaluation orders [21]:

- AOR, NOR, HA, and HN evaluate to NF.
- CBV evaluates to WNF.
- CBN evaluates to WHNF.
- HE evaluates to HNF.

2.9. Recursion

A recursive function is a function defined in terms of itself. It is well-known how to write recursive functions in the pure λ-calculus, which has no function naming mechanism. We illustrate with an example. Consider the informal definition:

$$fac \triangleq \lambda n.\, cond\,(isZero\;\; n)\; one\,(mult\, n\,(fac\,(pred\, n)))$$

The above primitives (*cond*, *isZero*, etc) are abbreviations for their λ-calculus encoding [2]. Our informal naming mechanism $\triangleq$ does not exist in the λ-calculus. The trick is to define recursive functions as fixed points of higher-order abstractions (called functionals), and to use a fixed point "finder" λ-expression (or combinator, for it is a closed λ-expression). In our case the functional is:

$$Fac \equiv \lambda f.\lambda n.\, cond\,(isZero\;\; n)\; one\,(mult\, n\,(f\;(pred\, n)))$$

Recall we use syntactic equality (≡) as an abbreviation meta-notation, nothing is being named.

A mathematical foundation of recursion is Tarski's fixed point theorem [22] which says that every recursive functional has a least fixed point. We recall that f is a fixed point of F when $Ff = f$. In the λ-calculus there is an equivalent fixed point theorem stating that any functional λ-expression F has a λ-expression X such that $X = F\,X$ [2]. Furthermore, there exists several fixed point combinators C such that $CF = F(CF)$ and $CF = X$. One of the most common is the Y combinator, due to Haskell B. Curry:

$$Y \equiv \lambda f.(\lambda x.(f\;(x\,x)))\,(\lambda x.(f\;(x\,x)))$$

which is indeed a fixed point combinator:

$$
\begin{array}{ll}
 & Y\,F \\
\equiv & \quad \{Y \equiv \lambda f.(\lambda x.(f\,(x\,x)))\,(\lambda x.(f\,(x\,x)))\} \\
 & \underline{(\lambda f.(\lambda x.(f\,(x\,x)))\,(\lambda x.(f\,(x\,x))))\,F} \\
\stackrel{\beta}{\rightarrow} & \\
 & \underline{(\lambda x.F\,(x\,x))\,\lambda x.F\,(x\,x))} \\
\stackrel{\beta}{\rightarrow} & \\
 & F((\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))) \\
\stackrel{\beta}{=} & \quad \{\text{by inverse } \beta\text{-reduction}\} \\
 & F((\lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)))F) \\
\equiv & \quad \{Y \equiv \lambda f.(\lambda x.(f\,(x\,x)))\,(\lambda x.(f\,(x\,x)))\} \\
 & F\,(Y\,F)
\end{array}
$$

If we consider the application of $(Y\ \mathit{Fac})$ to some argument N we have the following reduction sequence in CBN:

$$
\begin{array}{ll}
 & (Y\ \mathit{Fac})\,N \\
\stackrel{\beta\star}{\rightarrow} & \quad \{Y \equiv \lambda f.(\lambda x.(f\,(x\,x)))\,(\lambda x.(f\,(x\,x)))\} \\
 & \underline{\mathit{Fac}\,(Y\ \mathit{Fac})}\,N \\
\stackrel{\beta}{\rightarrow} & \quad \{\mathit{Fac} \equiv \lambda f.\lambda n.\mathit{cond}\,(\mathit{isZero}\,n)\ \mathit{one}\,(\mathit{mult}\,n\,(f\,(\mathit{pred}\,n)))\} \\
 & \underline{(\lambda n.\mathit{cond}\,(\mathit{isZero}\,n)\ \mathit{one}\,(\mathit{mult}\,n\,((Y\ \mathit{Fac})\,(\mathit{pred}\,n))))\,N} \\
\stackrel{\beta}{\rightarrow} & \\
 & \mathit{cond}\,(\mathit{isZero}\,N)\ \mathit{one}\,(\mathit{mult}\,N\,((Y\ \mathit{Fac})\,(\mathit{pred}\,N))) \\
\stackrel{\beta\star}{\rightarrow} & \quad \{[\![N]\!] > 0\} \\
 & \mathit{mult}\,N\,((Y\ \mathit{Fac})\,(\mathit{pred}\,N)) \\
\stackrel{\beta\star}{\rightarrow} & \\
 & \mathit{mult}\,N\,((\mathit{Fac}\,(Y\ \mathit{Fac}))\,(\mathit{pred}\,N)) \\
\stackrel{\beta\star}{\rightarrow} & \quad \{[\![\mathit{pred}\,N]\!] > 0\} \\
 & \mathit{mult}\,N\,(\mathit{mult}\,(\mathit{pred}\,N)\,((Y\ \mathit{Fac})\,(\mathit{pred}\,(\mathit{pred}\,N)))) \\
\stackrel{\beta\star}{\rightarrow} & \\
 & \mathit{mult}\,N\,(\mathit{mult}\,(\mathit{pred}\,N)\,(\ldots\ \mathit{one}\ \ldots))
\end{array}
$$

The result is the factorial of N.

The Y combinator will not work under eager evaluation because in $Y(Fac\,Y)$ it will evaluate the argument $Fac\,Y$ indefinitely. The Z combinator is used instead:

$$Z \equiv \lambda f.(\lambda x.(f\ (\lambda y.(x\,x)\,y)))\,(\lambda x.(f\ (\lambda y.(x\,x)\,y)))$$

We show it at work under CBV (recall Section 2.5):

$$\begin{array}{ll}
 & Z\,F \\
\equiv & \quad \{Z \equiv \lambda f.(\lambda x.(f\ (\lambda y.(x\,x)\,y)))\,(\lambda x.(f\ (\lambda y.(x\,x)\,y)))\} \\
 & \underline{(\lambda f.(\lambda x.(f\ (\lambda y.(x\,x)\,y)))\,(\lambda x.(f\ (\lambda y.(x\,x)\,y)))\,F} \\
\stackrel{\beta}{\rightarrow} & \quad \{F \text{ is a } \lambda\text{-abstraction}\} \\
 & \underline{(\lambda x.F\ (\lambda.y(x\,x)\,y))\,(\lambda x.F\ (\lambda y.(x\,x)\,y))} \\
\stackrel{\beta}{\rightarrow} & \\
 & F\ (\lambda y.(\lambda x.F\ (\lambda y.(x\,x)\,y))\,(\lambda x.F\ (\lambda y.(x\,x)\,y))\,y) \\
\stackrel{\beta}{=} & \quad \{\text{by inverse } \beta\text{-reduction}\} \\
 & F\ (\lambda y.(Z\,F)\,y) \\
\stackrel{\beta\eta}{=} & \quad \{\text{by } \eta\text{-conversion}\} \\
 & F\,(Z\,F)
\end{array}$$

In this case the argument passed to the recursive function f is lifted to a λ-abstraction $(\lambda t.f\,t)$ which is called a ***thunk***. This prevents f to be evaluated under CBN and CBV, as it appears in the body of a λ-abstraction. To force the evaluation of f an arbitrary terminating argument (a dummy argument denoted by $_$) is passed. When using Z to write a recursive function, the branches of the conditional must be lifted to thunks also, and the dummy argument must be passed to the invocations of the recursive function (notice that passing the formal argument of the thunk where the recursive call takes place is equivalent).

The factorial function is rewritten to work with Z and CBV in the following way:

$$Fac \equiv \lambda f.\lambda n.cond\ (isZero\,n)\ (\lambda t.one)\ (\lambda t.mult\,n\,((f\ (pred\,n))\,t))$$

We show the use of Z in action. We write $[E]_{CBV}$ for the result (if any) of

reducing E with a CBV strategy:

$$
\begin{array}{ll}
 & ((Z\ Fac)\ N)\ _ \\
\equiv & \quad \{Z \equiv \lambda f.(\lambda x.(f\ (\lambda y.(x\,x)\,y)))\ (\lambda x.(f\ (\lambda y.(x\,x)\,y)))\} \\
 & \underline{(((\lambda f.(\lambda x.(f\ (\lambda y.(x\,x)\,y)))\ (\lambda x.(f\ (\lambda y.(x\,x)\,y)))\ Fac)\ N)}\ _ \\
\stackrel{\beta}{\rightarrow} & \\
 & ((\underline{(\lambda x.Fac\,(\lambda y.(x\,x)\,y))\ (\lambda x.Fac\,(\lambda y.(x\,x)\,y))})\ N)\ _ \\
\stackrel{\beta}{\rightarrow} & \\
 & ((Fac\,(\lambda y.((\lambda x.Fac\,(\lambda y.(x\,x), y))\ (\lambda x.Fac\,(\lambda y.(x\,x)\,y)))\,y))\ N)\ _ \\
\equiv & \quad \{Z_Fac \equiv (\lambda x.Fac\,(\lambda y.(x\,x)\,y))\ (\lambda x.Fac\,(\lambda y.(x\,x)\,y))\} \\
 & ((\underline{Fac\,(\lambda y.(Z_Fac)\,y)})\ N)\ _ \\
\stackrel{\beta}{\rightarrow} & \quad \{Fac \equiv \lambda f.\lambda n.cond\ (isZero\,n)\ (\lambda t.one)\ (\lambda t.mult\,n\,((f\ (pred\,n))\,t))\} \\
 & (\underline{(\lambda n.cond\ (isZero\,n)\ (\lambda t.one)} \\
 & \underline{(\lambda t.mult\,n\,(((\lambda y.(Z_Fac)\,y)\ (pred\,n))\,t)))\ N})\ _ \\
\stackrel{\beta}{\rightarrow} & \quad \{N \text{ is in WNF}\} \\
 & (cond\ (isZero\ N)\ (\lambda t.one)\ (\lambda t.mult\ N\ (((\lambda y.(Z_Fac)\,y)\ (pred\ N))\,t)))\ _ \\
\stackrel{\beta\star}{\rightarrow} & \quad \{[\![N]\!] > 0\} \\
 & \underline{(\lambda t.mult\ N\ (((\lambda y.(Z_Fac)\,y)\ (pred\ N))\,t))\ _} \\
\stackrel{\beta}{\rightarrow} & \\
 & mult\ N\ (((\lambda y.(Z_Fac)\,y)\ (pred\ N))\ _) \\
\stackrel{\beta\star}{\rightarrow} & \quad \{mult_N \equiv [mult\ N]_{CBV}\} \\
 & mult_N\ (((\lambda y.(Z_Fac)\,y)\ (pred\ N))\ _) \\
\stackrel{\beta\star}{\rightarrow} & \quad \{pred_N \equiv [pred\ N]_{CBV},\ [\![pred_N]\!] > 0\} \\
 & mult_N\ (mult\ pred_N\ (((\lambda y.(Z_Fac)\,y)\ (pred\ pred_N))\ _)) \\
\stackrel{\beta\star}{\rightarrow} & \quad \{mult_pred_N \equiv [mult\ pred_N]_{CBV},\ \ldots,\ one \equiv [one]_{CBV}\} \\
 & mult_N\ (mult_pred_N\ (\ldots\ one\ \ldots))
\end{array}
$$

The result is the factorial of N.

2.10. Illustrative λ-expressions used in the code

We select a set of illustrative λ-expressions to be evaluated or interpreted by our code. Each expression is representative of some features of the λ-

calculus. The results given from them show up the peculiarities in the variety of evaluators and interpreters presented in this thesis.

Identity and simple redex. When applying `ident` to some expression it must return the expression itself whereas `redex` is a λ-expression that is always evaluated.

```
ident = Lam "x" (Var "x")
redex = App (Lam "x" (Var "x")) (Var "x")
```

```
*TermsPool> ident
\x.x
*TermsPool> redex
(\x.x)x
```

Capturing avoiding substitution. This expression is used to illustrate capture avoiding substitution at work. It must provoke the renaming of the bound `y` variable.

```
capture = App (Lam "x" (Lam "y" (Var "x"))) (Var "y")
```

```
*TermsPool> capture
(\x.\y.x)y
```

Application of Variables. These expressions test behaviour on non-redex normal forms consisting of applications of free variables:

```
free = App (App (Lam "x" (App (Var "x") (Var "y")))
                (Var "x"))
           (App  (Lam "y" (App (Var "z") (Var "y")))
            (Var "y"))

varApp = App (Var "x") (Lam "x" (Var "y"))
```

```
*TermsPool> free
((\x.x y)x)((\y.z y)y)
*TermsPool> varApp
x(\x.y)
```

Church numerals. We use numerals for the factorial function. We build them up from the `zero` and the auxiliary `suc` function. Notice that they are not the canonical form for Church numerals, but an unreduced version.

```
zero  = Lam "f" (Lam "x" (Var "x"))
one   = App suc zero
two   = App suc one
three = App suc two
```

```
*TermsPool> zero
\f.\x.x
*TermsPool> one
(\n.\f.\x.f((n f)x))(\f.\x.x)
*TermsPool> two
(\n.\f.\x.f((n f)x))((\n.\f.\x.f((n f)x))(\f.\x.x))
*TermsPool> three
(\n.\f.\x.f((n f)x))((\n.\f.\x.f((n f)x))
                     ((\n.\f.\x.f((n f)x))(\f.\x.x)))
```

Strictness. These expressions contain undefined subexpressions. `omega` must make every evaluator and interpreter run forever. Evaluators and interpreters which evaluate under λ-abstraction must run forever with `lamOmega` whereas the rest must not. `constOmega` tests if the evaluator or interpreter implements strict semantics.

```
omega      = App twice twice
    where twice = Lam "x" (App (Var "x") (Var "x"))
lamOmega   = Lam "x" omega
constOmega = App (Lam "x" (Var "y")) omega
```

```
*.TermsPool> omega
(\x.x x)(\x.x x)
*.TermsPool> lamOmega
\x.(\x.x x)(\x.x x)
*TermsPool> constOmega
(\x.y)((\x.x x)(\x.x x))
```

Recursion. We illustrate how to implement recursion in the λ-calculus. We have the two versions seen in Section 2.9, the one with the Y combinator, which is appropriate with non-strict evaluation orders that do not evaluate under λ-abstraction, and the one with the Z combinator, which is appropriate for strict evaluation orders that do not evaluate under λ-abstraction. We use the auxiliary functions `cond`, `isZero`, `one`, `mult` and `pre`.

```
y = Lam "f" (App fTwice fTwice)
    where fTwice = Lam "x"
                    (App (Var "f")
                     (App (Var "x") (Var "x")))
fact = Lam "f"
       (Lam "n"
        (App (App (App cond (App isZero (Var "n")))
                  one)
         (App (App mult (Var "n"))
          (App (Var "f") (App pre (Var "n"))))))

z = Lam "f" (App fYTwice fYTwice)
    where fYTwice = Lam "x"
                     (Lam "y"
                      (App (App (Var "f")
                                (App (Var "x") (Var "x")))
                       (Var "y")))
fact_d = Lam "f"
         (Lam "n"
          (App (App (App cond (App isZero (Var "n")))
                (Lam "d" one))
           (Lam "d" (App (App mult (Var "n"))
                     (App (App (Var "f")
                               (App pre (Var "n")))
                      (Var "d"))))))
```

```
*TermsPool> y
\f.(\x.f(x x))(\x.f(x x))
*TermsPool> fact
\f.\n.(((\c.\t.\f.(c t)f)((\n.(n((\x.\y.x)(\p.\q.q)))
(\p.\q.p))n))((\n.\f.\x.f((n f)x))(\f.\x.x)))
(((\m.\n.\f.m(n f))n)(f((\x.\y.\z.((x(\p.\q.q(p y)))
((\p.\q.p)z))(\x.x))n)))
*TermsPool> z
\f.(\x.\y.(f(x x))y)(\x.\y.(f(x x))y)
*TermsPool> fact_d
\f.\n.(((\c.\t.\f.(c t)f)((\n.(n((\x.\y.x)(\p.\q.q)))
(\p.\q.p))n))(\d.(\n.\f.\x.f((n f)x))(\f.\x.x)))
(\d.((\m.\n.\f.m(n f))n)((f((\x.\y.\z.((x(\p.\q.q(p y)))
((\p.\q.p)z))(\x.x))n))d))
```

The whole code for the illustrative expressions and the auxiliary functions (`suc`, `cond`, `isZero`, `pre`, `mult`) can be found in Appendix A

Chapter 3

Evaluators for the Untyped λ-calculus

In this chapter we discuss and implement λ-calculus evaluators for the foremost evaluation orders in non-strict Haskell. We discuss the complications posed by free variables and Haskell's non-strict semantics. We assume the Haskell representation of λ-expressions given in Section 2.1. We are not interested in issues of representation and efficient substitution. We are happy with the standard notion of substitution given in Chapter 2. In Section 3.1 we present evaluators in classic, non-monadic style. In Section 3.2 we present evaluators in monadic style [23]. In Sections 3.3 and 3.4 we present, respectively, parametric and mixin-based generic evaluators.

We identify two main styles of writing evaluators in Haskell:

Eval-apply: Application evaluation is performed by an auxiliary `apply` function that pattern matches on expressions to force their evaluation. This is the style often found in the literature because the implementation languages chosen lack `case` expressions.

Eval-case: Application evaluation is performed by a `case` expression that pattern matches on expressions to force their evaluation. Haskell supports `case` expressions and we use them systematically for they dispense with the need for an extra `apply` thus simplifying the code.

3.1. Non-monadic Evaluators

3.1.1. Naive Evaluator

A naive CBV evaluator for the untyped λ-calculus is shown below. All naive evaluators are shown in Appendix B.

```
evalCBV :: SLTerm -> SLTerm
evalCBV v@(Var s)   = v
evalCBV l@(Lam v b) = l
evalCBV (App m n)   =
    let  m' = evalCBV m
         n' = evalCBV n
    in case m' of
           (Lam v b) -> evalCBV $ subst v b n'
           _         -> App m' n'
```

We are using pattern matching to define our evaluator by case analysis on the patterns of their input. Pattern matching can be used in the definition of a function, where the body of the matching rule follows the `=`, or after a `case of` sentence, where the delimiter `->` specifies the body for that case. For example, the lines:

```
evalCBV (App m n)   =
    let  m' = evalCBV m
         n' = evalCBV n
    in ...
```

indicate the behaviour of the evaluator in the case of λ-applications. When applied to an expression e, the former is evaluated to WHNF, that is, to an expression of the form `App` e_1 e_2 where e_1, e_2 are arbitrary expressions. Variable `m` is matched with e_1 and variable `n` is matched with e_2, which are then used in the left side to deliver the result. Patterns may have the wildcard `_`, which matches any expression. That expression is ignored in the right side, like in:

```
    in case m' of
           (Lam v b) -> evalCBV $ subst v b n'
           _         -> App m' n'
```

where `App m' n` is returned in case `m'` does not match the pattern `(Lam v b)`, no matter which expression is `m'`. Pattern matching also allows `@` notation to refer, in the left side, to the whole expression being matched, as was used in the first line of the evaluator:

```
evalCBV v@(Var s) = v
```

where `v` stands for the whole parameter of the function.

In CBV, arguments in redexes must be evaluated before substituting their result in the body of the abstractions. However, Haskell has a CBD evaluation order and Haskell expressions are evaluated on demand. The bindings

within a `let` are not evaluated until they are used by the expression after the `in`. Consequently, `n'` (which contains a call to `evalCBV`) is passed to `subst` unevaluated, contrary to CBV. The interference of the implementation language's evaluation order was noted by Reynolds [19] in the context of definitional interpreters.

We run several tests taking λ-expressions from Section 2.10

```
*NaiveCBV> evalCBV ident
\x.x
*NaiveCBV> evalCBV redex
x
*NaiveCBV> evalCBV capture
\y1.y
*NaiveCBV> evalCBV free
(x y)(z y)
*NaiveCBV> evalCBV varApp
x(\x.y)
*NaiveCBV> evalCBV two
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*NaiveCBV> evalCBV omega
  C-c C-cInterrupted.
*NaiveCBV> evalCBV lamOmega
\x.(\x.x x)(\x.x x)
*NaiveCBV> evalCBV constOmega
y
*NaiveCBV> evalCBV $ App (App y fact) three
  C-c C-cInterrupted.
*NaiveCBV> evalCBV $ App (App (App z fact_d) three) (Var "_")
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
(((\f.\x.x)f)x))f))f))f)
*NaiveCBV>
```

The Church numeral is just reduced to WNF. The evaluator returns for `lamOmega` (because CBV does not evaluate the body of λ-abstractions) and *wrongly* from `constOmega`. This is because in this naive version Haskell non-strict semantics is interfering. The factorial function works only with the Z combinator, and yields a WNF as result.

Haskell offers the ability to force *evaluation*[1] via pattern matching and via some primitive operators for strict application. We can rewrite the naive evaluator using this mechanisms (Section 3.1.2 and Section 3.1.3)

3.1.2. Evaluator Using Pattern Matching

Using pattern matching we can force Haskell to *evaluate* some expression. In the naive evaluator (Section 3.1.1), for example, we know `m'` is *evaluated* because we try to match that expression to the pattern `Lam v b`. We can apply the same trick to the `n'`. (Appendix C shows similar versions for other evaluation orders.)

```
evalCBV :: SLTerm -> SLTerm
evalCBV v@(Var s)   = v
evalCBV l@(Lam v b) = l
evalCBV (App m n)   =
    let m' = evalCBV m
        n' = evalCBV n
    in case m' of
           (Lam v b) -> case n' of
                          (Var _) -> evalCBV $ subst v b n'
                          _       -> evalCBV $ subst v b n'
           _         -> case n' of
                          (Var _) -> App m' n'
                          _       -> App m' n'
```

Notice the redundancy which cannot be factored out: at least one pattern case must be provided to force evaluation, a single case with wildcard will not work. Running the evaluator we get:

```
*PatternCBV> evalCBV ident
\x.x
*PatternCBV> evalCBV redex
x
*PatternCBV> evalCBV capture
\y1.y
*PatternCBV> evalCBV free
(x y)(z y)
*PatternCBV> evalCBV varApp
x(\x.y)
```

[1]From now on we use *evaluation* in italics when referring to Haskell's evaluation.

```
*PatternCBV> evalCBV two
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*PatternCBV> evalCBV omega
  C-c C-cInterrupted.
*PatternCBV> evalCBV lamOmega
\x.(\x.x x)(\x.x x)
*PatternCBV> evalCBV constOmega
  C-c C-cInterrupted.
*PatternCBV> evalCBV $ App (App y fact) three
  C-c C-cInterrupted.
*PatternCBV> evalCBV $ App (App (App z fact_d) three) (Var "_")
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
(((\f.\x.x)f)x))f))f))f)
*PatternCBV>
```

The results are as expected. The evaluator never returns for `omega` and for `constOmega`, reflecting strict semantics. Recursion works with the Z combinator.

3.1.3. Evaluator Using Strict Application

Haskell has explicit function application operators, a non-strict one `($)` and a strict one `($!)`. Strict application forces the *evaluation* of its second argument endowing Haskell with CBV function application.

We can write the CBV evaluator using this operator. (Appendix D shows similar versions for other evaluation orders.)

```
evalCBV :: SLTerm -> SLTerm
evalCBV v@(Var s)   = v
evalCBV l@(Lam v b) = l
evalCBV (App m n)   =
    let  m' = evalCBV m
         n' = evalCBV n
    in case m' of
           (Lam v b) -> evalCBV $ subst v b $! n'
           _         -> App m' $! n'
```

Running the evaluator we have:

```
*StrictCBV> evalCBV ident
\x.x
*StrictCBV> evalCBV redex
x
*StrictCBV> evalCBV capture
\y1.y
*StrictCBV> evalCBV free
(x y)(z y)
*StrictCBV> evalCBV varApp
x(\x.y)
*StrictCBV> evalCBV two
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*StrictCBV> evalCBV omega
  C-c C-cInterrupted.
*StrictCBV> evalCBV lamOmega
\x.(\x.x x)(\x.x x)
*StrictCBV> evalCBV constOmega
  C-c C-cInterrupted.
*StrictCBV> evalCBV $ App (App y fact) three
  C-c C-cInterrupted.
*StrictCBV> evalCBV $ App (App (App z fact_d) three) (Var "_")
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
(((\f.\x.x)f)x))f))f))f)
*StrictCBV>
```

which are the expected results.

Strict application is defined in terms of the strict sequencing operator `seq` which is a Haskell primitive (it cannot be programmed in CBD) whose semantics is described by the following equation ([8]):

$$[\![seq\ a\ b]\!] = \begin{cases} \bot & \text{if } [\![a]\!] = \bot \\ [\![b]\!] & \text{if } [\![a]\!] \neq \bot \end{cases}$$

Strict application is implemented in terms of `seq`:

```
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Evaluators can be written using `seq` directly. (Appendix E shows similar versions for other evaluation orders.)

```
evalCBV :: SLTerm -> SLTerm
evalCBV v@(Var s)   = v
evalCBV l@(Lam v b) = l
evalCBV (App m n)   =
    let  m' = evalCBV m
         n' = evalCBV n
    in case m' of
         (Lam v b) -> n' `seq` (evalCBV $ subst v b n')
         _         -> n' `seq` (App m' n')
```

3.1.4. Evaluator Using Continuations

Reynolds [19] applied continuation-passing style to implement an interpreter for a simple applicative object language that implemented the right evaluation order. Plotkin [17] used the same technique to implement CBV in CBN and vice-versa. Hatcliff and Danvy [7] extended this technique giving transformations from direct style to CPS and vice-versa, using Moggi's computational metalanguage [12].

Recall from Chapter 1 that ***continuation-passing style*** is a functional programming style where the control of the program is passed explicitly in the form of a continuation argument to functions. A continuation is a function which represent what to do next in a computation. The continuation takes the result of the computation at the current point and computes the final result from this partial result. A function in CPS style takes a continuation and returns the result from applying that continuation to its partial result. Of course, one continuation can be the result of passing a continuation to a CPS function, which takes another continuation, and so forth. Eventually, a primitive continuation, which is not the result of passing any other continuation to a CPS function, may be passed (often the `id` function). This makes the control of the program explicit and accessible to the programmer.

A CPS function may take some arguments of arbitrary types and then must return the type `(a -> r) -> r` where `a` is the type of the partial result of the function, `(a -> r)` is the continuation and `r` is the type of the final result of the whole computation.

To give an example, consider the program that takes an integer, multiplies it by two and then adds one to the result. Without continuations this program can be implemented as follows:

```
byTwo :: Int -> Int
byTwo n = 2 * n

addOne :: Int -> Int
addOne n = n + 1

mainProgram :: Int -> Int
mainProgram n = addOne $ byTwo n
```

In order to transform this program into CPS the types of the elementary functions (`byTwo` and `addOne`) must accept a continuation an give a result.

```
byTwoCPS :: Int -> (Int -> r) -> r
byTwoCPS n c = c (2 * n)

addOneCPS :: Int -> (Int -> r) -> r
addOneCPS n c = c (n + 1)
```

The `mainProgram` must also change, passing each function as the continuation of the previous one, ending with the `id` continuation:

```
mainProgramCPS :: Int -> Int
mainProgramCPS n = byTwoCPS n (\x -> addOneCPS x id)
```

The same idea can be used to explicitly manage the evaluation of the subexpressions in a CBV evaluator, like in the following code. (As before, Appendix F shows CPS implementations for other evaluation orders.)

```
evalCBV :: SLTerm -> (SLTerm -> r) -> r
evalCBV v@(Var s)   c = c v
evalCBV l@(Lam v b) c = c l
evalCBV (App m n)   c =
    evalCBV m
    (\m' -> evalCBV n
            (\n' -> case m' of
                    (Lam v b) -> evalCBV (subst v b n') c
                    _         -> c $ App m' n'))
```

The explicit management of the program control implements the intended eager evaluation in the lazy Haskell's *evaluation*. Every invocation of the evaluator pushes its actions into the continuation, which is the future of the computation, and pass the new continuation to the recursive invocation of the evaluator. This way, when the whole computation (which has been sliced into continuations consisting of CPS functions taking continuations. . .) is *evaluated*, the result corresponds to the intended semantics. Running the evaluator over the tests:

```
*ContCBV> evalCBV ident
\x.x
*ContCBV> evalCBV redex
x
*ContCBV> evalCBV capture
\y1.y
*ContCBV> evalCBV free
(x y)(z y)
*ContCBV> evalCBV varApp
x(\x.y)
*ContCBV> evalCBV varApp
(x y)(z y)
*ContCBV> evalCBV two
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*ContCBV> evalCBV omega
  C-c C-cInterrupted.
*ContCBV> evalCBV lamOmega
\x.(\x.x x)(\x.x x)
*ContCBV> evalCBV constOmega
  C-c C-cInterrupted.
*ContCBV> evalCBV $ App (App y fact) three
  C-c C-cInterrupted.
*ContCBV> evalCBV $ App (App (App z fact_d) three) (Var "_")
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
(((\f.\x.x)f)x))f))f))f)
```

we confirm that the CPS mechanism preserves the intended semantics.

3.2. Monadic Evaluators

Moggi [12] first used monads in programming languages semantics and Wadler [23] adopts them to structure functional programs. A monad is a category-theoretical concept which lets the representation of different notions of computations, which may entail side effects, as different types in a pure functional programming setting. A pure computation can be lifted to monadic, and two monadic computations can be sequenced obtaining a new monadic computation. Wadler showed how to change some features of a monadic program using different monads and minimal changes in the program. He proposed the implementation of an evaluator for a simple programming language that can be extended with additional features with the use of monads.

Following a similar idea, we propose a monadic evaluator where the strategies to get rid of the semantics issues are encapsulated in monads. Then, varying the monad those strategies can be selected. This leads to a clearer and untangled implementation for our evaluators.

3.2.1. Monads

Monads are constructs from category theory that are very useful in functional programming. In particular, they provide a formal means of dealing with side effects. The monad is a functor (a mappable type constructor) which satisfies identity and associative laws:

Definition 3.1. A ***monad*** is a functor and three operations, *return*, ($\gg\!=$), and *join* satisfying certain laws [3]. We are interested in the first two and their laws which are defined in the Haskell standard `Monad` class [14]:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
```

left-identity: `return x >>= f = f x`

right-identity: `m >>= return = m`

associativity: `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

Haskell's type system is not powerful enough to check that the axioms are met. Programmers must prove the axioms.

In words, the `return` function lifts pure types into monadic types. The `(>>=)` sequences monadic computations. In the example we can see a function that takes and integer, multiplies it by two and then adds one to the result in a monadic style:

```
byTwoAddOne :: Monad m => Int -> m Int
byTwoAddOne = (\n -> return (n * 2) >>=
                (\n' -> return (n' + 1)))
```

Haskell offers the `do` notation which is syntactic sugar for uses of `return` and `(>>=)` notation. The previous example is rewritten in `do` notation as follows:

```
byTwoAddOneDo :: Monad m => Int -> m Int
byTwoAddOneDo = \n -> do n' <-  return $ n * 2
                         return $ n' + 1
```

Notice the imperative taste. A function returning a monadic type must appear in each line, whose inner parameter can be saved for the next lines using the `<-` notation.

3.2.2. Basic Monadic Evaluator

We write naive evaluators in monadic style, replacing `let` expressions by `do` expressions. The CBV evaluator follows. (Appendix G shows versions for other evaluation orders.)

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

The type variable `m` stands for any monad, so the type of the evaluator is overloaded. The evaluator does not implement any monad. It must be instantiated, taking some existing monad, to a particular evaluator which implements a semantics preserving strategy.

Not every monad leads to a semantics preserving evaluator. If we instantiate the monadic evaluator with the trivial `IdMonad`:

```
data Id a = In { out :: a }
```

```
instance Monad Id where
    return   = In
    m >>= t  = t (out m)
```

we obtain an CBV evaluator equivalent to the naive one (Section 3.1.1), because the `return` wraps the expression in a new data type and the `>>=` passes the unwrapped expression to the `f` function. When applied to the battery of tests it behaves like the naive version:

```
*MonadCBV> evalCBV ident :: Id SLTerm
\x.x
*MonadCBV> evalCBV redex :: Id SLTerm
x
*MonadCBV> evalCBV capture :: Id SLTerm
\y1.y
*MonadCBV> evalCBV free :: Id SLTerm
(x y)(z y)
*MonadCBV> evalCBV varApp :: Id SLTerm
x(\x.y)
*MonadCBV> evalCBV two :: Id SLTerm
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*MonadCBV> evalCBV omega :: Id SLTerm
  C-c C-cInterrupted.
*MonadCBV> evalCBV lamOmega :: Id SLTerm
\x.(\x.x x)(\x.x x)
*MonadCBV> evalCBV constOmega :: Id SLTerm
y
*MonadCBV> evalCBV $ App (App y fact) three :: Id SLTerm
  C-c C-cInterrupted.
*MonadCBV> :{
*MonadCBV| evalCBV $ App (App (App z fact_d) three) (Var "_")
*MonadCBV| :: Id SLTerm
*MonadCBV| :}
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
(((\f.\x.x)f)x))f))f))f)
*MonadCBV>
```

For a monad to be suitable for an evaluator which preserves semantics, the $\gg=$ operation must force the *evaluation* of the inner parameter in the first monadic argument. The explicit manipulation of the program control carried out by CPS can be used to achieve that. In Section 3.2.3 we describe the Haskell's `Cont` monad, which precisely implements CPS.

3.2.3. Continuation Monad

There is a monad for continuations in the Haskell's standard library. The definition of the `Cont` monad is the following [13]:

```
newtype Cont r a = Cont {
    runCont :: (a -> r) -> r
}

instance Monad (Cont r) where
    return a = Cont ($ a)
    m >>= k  = Cont $ \c -> runCont m $ \a -> runCont (k a) c
```

Here `return` lifts pure functions to the continuation. It packs the partial left-application of the `$` operator to its argument inside the `Cont` data type. Recall that `$` has the type:

```
($) :: (a -> b) -> a -> b
```

Its partial left-application to some expression of type `a` has the type

```
(a -> b) -> b
```

which, renaming `r` by `b`, is the type of a CPS function (and the type of the `runCont` field of the `Cont` data type).

The `(>>=)` packs into the data type a new CPS function taking the CPS function inside `m` and passing to it the continuation returned from the application of `k` to the intermediate value `a`.

With this monad instance CBV behaves:

```
*MonadCBV> (runCont $ evalCBV ident) id
\x.x
*MonadCBV> (runCont $ evalCBV redex) id
x
*MonadCBV> (runCont $ evalCBV capture) id
\y1.y
*MonadCBV> (runCont $ evalCBV free) id
```

```
(x y)(z y)
*MonadCBV> (runCont $ evalCBV varApp) id
x(\x.y)
*MonadCBV> (runCont $ evalCBV two) id
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*MonadCBV> (runCont $ evalCBV omega) id
  C-c C-cInterrupted.
*MonadCBV> (runCont $ evalCBV lamOmega) id
\x.(\x.x x)(\x.x x)
*MonadCBV> (runCont $ evalCBV constOmega) id
  C-c C-cInterrupted.
*MonadCBV> (runCont $ evalCBV $ App (App y fact) three) id
  C-c C-cInterrupted.
*MonadCBV> :{
*MonadCBV| (runCont $
*MonadCBV| evalCBV $ App (App y fact) three)
*MonadCBV| id
*MonadCBV| :}
  C-c C-cInterrupted.
*MonadCBV> :{
*MonadCBV| (runCont $
*MonadCBV| evalCBV $ App (App (App z fact_d) three) (Var "_"))
*MonadCBV| id
*MonadCBV| :}
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
(((\f.\x.x)f)x))f))f))f)
*MonadCBV>
```

which is the expected, since the `Cont` monad preserves the semantics.

3.3. Parametric Evaluator

In Section 3.1 we have seen several families of evaluators, using different programming styles and different strategies to make them semantics preserving. In this section we identify six places where evaluation calls itself recursively in the evaluator family. This leads us to an evaluator which varies its

behaviour in each of those six places. The evaluator receives six other 'small' evaluators as arguments and applies them in the right place. Using a fixed point operator we define evaluators recursively in term of themselves. The result is a parametric evaluator which allows the definition of every evaluation order, choosing the places where the evaluator calls itself.

3.3.1. Opportunities for Evaluation

If we analyse the families of evaluators obtained in the previous section we realise they all follow a very similar pattern. In the first place, all leave expressions of the form (`Var s`) untouched. For the expressions (`Lam v b`) there are two possible alternatives: to leave them unevaluated or to evaluate their body. This is the first place in the code where an evaluation may occur. For applications (`App m n`), `m` must be evaluated first (say `m'` is the result). This is the second place in the code where an evaluation may occur. If `m'` is a λ-abstraction we must apply the β-rule. At this point, evaluators with strict semantics evaluate `n` (third place where evaluation may occur) and then proceed to substitution whereas the ones with non-strict semantics proceed to substitution directly. The substitution may be evaluated also, constituting the fourth point.

Finally, if `m'` is not a λ-abstraction the application must return the appropriate normal form which might require the evaluation of `m` and `n`. Those are the fifth and sixth places respectively. All this can be seen easier if we rearrange the code of the evaluators in the following fashion. Let us rewrite the naive AOR evaluator.

```
evalAOR :: SLTerm -> SLTerm
evalAOR v@(Var s) = v
evalAOR (Lam v b) = let b' = evalAOR b
                    in Lam v b'
evalAOR (App m n) =
    let  m' = evalAOR m
    in case m' of
           (Lam v b) -> let n' = evalAOR n
                        in evalAOR $ subst v b n
           _         -> let m'' = evalAOR m
                            n'' = evalAOR n
                        in App m'' n''
```

Taking parameters for every of those places we can give the parametric evaluator. (More parametric evaluators for every strategy in this chapter can be found in Appendix H.)

```
type Ev = SLTerm -> SLTerm
eval :: Ev -> Ev -> Ev ->
        Ev -> Ev -> Ev -> Ev
eval la op1 ar1 su op2 ar2 v@(Var s) = v
eval la op1 ar1 su op2 ar2 (Lam v b) =
    let b' = la b
    in Lam v b'
eval la op1 ar1 su op2 ar2 (App m n) =
    let  m' = op1 m
    in case m' of
           (Lam v b) -> let n' = ar1 n
                        in su $ subst v b n'
           _         -> let m'' = op2 m
                            n'' = ar2 n
                        in App m'' n''
```

with the specification of the uniform evaluation orders as fixed points:

```
aor  = eval aor aor aor aor aor aor
cbv  = eval id  cbv cbv cbv cbv cbv
cbn  = eval id  cbn id  cbn cbn id
he   = eval he  he  id  he  he  id
```

The places where evaluation may occur are the `let` construct in the `(Lam v b)` case (variable `b'`), the `let` constructs in the `(App m n)` case (the variables `m'`, `n'`, `m''`, and `n''`), and after doing the substitution (the `subst` function). If some evaluator does not need to evaluate in every place, the `id` function could be used as parameter. We call the parameters `la`, `op1`, `ar1`, `su`, `op2` and `ar2` respectively. We call this the evaluator with ***orthogonal*** parameters, because they are independent among them.

Another version of the parametric evaluator could be written, where `op2` takes the result returned by `op1` (the `m'`) and `ar2` takes the result returned by `ar1` (the `n'`).

```
type Ev = SLTerm -> SLTerm
eval :: Ev -> Ev -> Ev ->
        Ev -> Ev -> Ev -> Ev
eval la op1 ar1 su op2 ar2 v@(Var s) = v
eval la op1 ar1 su op2 ar2 (Lam v b) =
    let b' = la b
    in Lam v b'
eval la op1 ar1 su op2 ar2 (App m n) =
```

```
let  m' = op1 m
     n' = ar1 n
in case m' of
       (Lam v b) -> su $ subst v b n'
       _         -> let m'' = op2 m'
                        n'' = ar2 n'
                    in App m'' n''
```

The fixed points for the uniform evaluator orders are:

```
aor  = eval aor aor aor aor id id
cbv  = eval id  cbv cbv cbv id id
cbn  = eval id  cbn id  cbn id id
he   = eval he  he  id  he  id id
```

Notice that the definitions have changed. For the evaluation orders where `op1` is not `id`, it is not needed to evaluate in the `op2` place, so `id` is passed as `op2`. The same occurs with the `ar1` and `ar2` parameters. We call this the ***non-orthogonal*** version, because there are dependencies between `op1`, `ar1` and `op2`, `ar2` respectively.

The differences between the orthogonal and the non-orthogonal evaluators are subtle. With the orthogonal version, every possible combination of the parameters leads to a different evaluation order, with a particular denotational semantics. However, the operational semantics are jeopardised, because the evaluation of the `m` and the `n` could occur twice. With the non-orthogonal version, the operational semantics are right, but some evaluation orders are denotationally equivalent. For example, the evaluation order defined as the fixed point (for the non-orthogonal evaluator):

```
aor' = eval aor' aor' aor' aor' aor' aor'
```

is denotationally equivalent to the non-orthogonal `aor` (although it is not operationally equivalent).

A possible solution to this problem could be to replace `op2` and `ar2` by a boolean pointing whether to keep the evaluated expression or whether to return to the unevaluated form. This will avoid reevaluation, but will no longer be consistent with our recursive specification of fixed points. We prefer to keep the type of our parameters and specify in every case which version of the parametric evaluator we are using.

It could be argued that the evaluation orders shadowed by the non-orthogonal version are of little interest, because they perform some work which later is discarded. Nevertheless, we want to show all possibilities, so

we will prefer the orthogonal version, which will be the default unless we point out the opposite.

This parametric evaluator still does not preserve semantics, because as with the naive one, the Haskell *evaluation* will interfere with the intended evaluation order. In Appendix H we present parametric evaluators using pattern matching, strict application operator (`$!`), strict sequencing operator (`seq`), and CPS.

We prefer the evaluator using CPS, because this strategy is not Haskell specific, so it has more generality:

```
type Ev = SLTerm -> (SLTerm -> SLTerm) -> SLTerm
eval :: Ev -> Ev -> Ev -> Ev -> Ev -> Ev -> Ev
eval la op1 ar1 su op2 ar2 v@(Var s) k = k v
eval la op1 ar1 su op2 ar2 (Lam v b) k = la b
                              (\b' -> k $ Lam v b')
eval la op1 ar1 su op2 ar2 (App m n) k =
    op1 m
    (\m' -> case m' of
            (Lam v b) -> ar1 n
                         ( \n' -> su (subst v b n') k)
            _         -> op2 m
                         (\m'' -> ar2 n
                          (\n'' -> k $ App m'' n'')))
```

With continuations, the fixed points can be expressed as well. We need to define a `cId` function which takes an expression and returns the partial left-application of the `$` operator to it.

```
cId :: a -> (a -> a) -> a
cId a = ($ a)
aor  = eval aor aor aor aor aor aor
cbv  = eval cId cbv cbv cbv cbv cbv
cbn  = eval cId cbn cId cbn cbn cId
he   = eval he  he  cId he  he  cId
```

We test the cbv evaluator with the illustrative examples.

```
*ContFix6> (cbv ident) id
\x.x
*ContFix6> (cbv redex) id
x
*ContFix6> (cbv capture) id
\y1.y
```

```
*ContFix6> (cbv free) id
(x y)(z y)
*ContFix6> (cbv varApp) id
x(\x.y)
*ContFix6> (cbv two) id
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*ContFix6> (cbv omega) id
  C-c C-cInterrupted.
*ContFix6> (cbv lamOmega) id
\x.(\x.x x)(\x.x x)
*ContFix6> (cbv constOmega) id
  C-c C-cInterrupted.
*ContFix6> :{
*ContFix6| (runCont $
*ContFix6| cbv $ App (App y fact) three)
*ContFix6| id
*ContFix6| :}
  C-c C-cInterrupted.
*ContFix6> :{
*ContFix6| (cbv $
*ContFix6| App (App (App z fact_d) three) (Var "_")
*ContFix6| ) id
*ContFix6| :}
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
(((\f.\x.x)f)x))f))f))f)
*ContFix6>
```

The parametric evaluator implements the intended semantics.

This evaluator can still be improved in the monadic style, leading to clearer and more flexible code.

```
type Ev m = SLTerm -> m SLTerm
eval :: Monad m => Ev m -> Ev m -> Ev m ->
        Ev m -> Ev m -> Ev m -> Ev m
eval la op1 ar1 su op2 ar2 v@(Var s) = return v
eval la op1 ar1 su op2 ar2 (Lam v b) =
    do b' <- la b
```

```
        return $ Lam v b'
eval la op1 ar1 su op2 ar2 (App m n) =
    do m' <- op1 m
       case m' of
                (Lam v b) -> do n' <- ar1 n
                                su $ subst v b n'
                _         -> do m'' <- op2 m
                                n'' <- ar2 n
                                return $ App m'' n''
```

This time, the id function is replaced by return.

```
aor  = eval aor    aor    aor    aor    aor    aor
cbv  = eval return cbv    cbv    cbv    cbv    cbv
cbn  = eval return cbn    return cbn    return return
he   = eval he     he     return he     return return
```

Similarly, the non-orthogonal version is:

```
type Ev m = SLTerm -> m SLTerm
eval :: Monad m => Ev m -> Ev m -> Ev m ->
        Ev m -> Ev m -> Ev m -> Ev m
eval la op1 ar1 su op2 ar2 v@(Var s) = return v
eval la op1 ar1 su op2 ar2 (Lam v b) =
    do b' <- la b
       return $ Lam v b'
eval la op1 ar1 su op2 ar2 (App m n) =
    do m' <- op1 m
       n' <- ar1 n
       case m' of
                (Lam v b) -> su $ subst v b n'
                _         -> do m'' <- op2 m'
                                n'' <- ar2 n'
                                return $ App m'' n''
```

with the fixed points:

```
aor  = eval aor    aor    aor    aor    return return
cbv  = eval return cbv    cbv    cbv    return return
cbn  = eval return cbn    return cbn    return return
he   = eval he     he     return he     return return
```

We will use the Cont monad as the default. If we run the parametric monadic evaluator

```
*MonadFix6> (runCont $ cbv ident) id
\x.x
*MonadFix6> (runCont $ cbv redex) id
x
*MonadFix6> (runCont $ cbv capture) id
\y1.y
*MonadFix6> (runCont $ cbv free) id
(x y)(z y)
*MonadFix6> (runCont $ cbv varApp) id
x(\x.y)
*MonadFix6> (runCont $ cbv two) id
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*MonadFix6> (runCont $ cbv omega) id
  C-c C-cInterrupted.
*MonadFix6> (runCont $ cbv lamOmega) id
\x.(\x.x x)(\x.x x)
*MonadFix6> (runCont $ cbv constOmega) id
  C-c C-cInterrupted.
*MonadFix6> :{
*MonadFix6| (runCont $
*MonadFix6| evalCBV $ App (App y fact) three)
*MonadFix6| id
*MonadFix6| :}
  C-c C-cInterrupted.
*MonadFix6> :{
*MonadFix6| (runCont $
*MonadFix6| cbv $ App (App y fact) three)
*MonadFix6| id
*MonadFix6| :}
  C-c C-cInterrupted.
*MonadFix6> :{
*MonadFix6| (runCont $
*MonadFix6| cbv $ App (App (App z fact_d) three) (Var "_"))
*MonadFix6| id
*MonadFix6| :}
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
```

```
(((\f.\x.x)f)x))f))f))f)
*MonadFix6>
```

it works as expected.

3.3.2. Parameters and Evaluators

The parametric evaluator of Section 3.3.1 defines a space of six orthogonal dimensions with two values for every dimension (a recursive call or `return` in the fixed point definition). We consider restricting this space. We are interested in doing evaluation (not just single-step reduction) so we can dispense with `su`, and make a recursive call after the substitution. This parameter is just coding either single-step or big-step evaluation. It seems reasonable to dispense with `op1` as well, which specifies if the operator of an application must be evaluated or not, because we always want to evaluate the operator. The problem is that `op1` in conjunction with `op2` allows the definition of hybrid strategies, so we can not get rid of it so easily.

Taking NOR, for example, the operator is first evaluated with CBN. If it is a λ-abstraction, then substitution takes place. Otherwise it must be evaluated with NOR, which evaluates the bodies of λ-abstractions, finding all the existing redexes. The hybrid strategies evaluate the operator in two stages, so we need two different parameters for the operator.

The hybrid strategies may be defined using mutual recursive fixed points, instead of single recursive ones:

```
nor = eval nor cbn return nor nor nor
hn  = eval hn  he  return hn  he  hn
ha  = eval ha  cbv ha     ha  ha  ha
```

Similarly with the non-orthogonal version:

```
nor = eval nor cbn return nor nor nor
ha  = eval ha  cbv ha     ha  ha  ha
hn  = eval hn  cbn return hn  hn  hn
```

NOR must stop evaluating the bodies of λ-abstractions for the operator, otherwise it will potentially evaluate innermost redexes and the resulting strategy will be no longer leftmost-outermost. This lets the definition of recursive functions using the Y combinator. Running with NOR (`fact` is the Haskell representation of the Fac function in Section 2.9 and `one` the Church encoding of 1.):

```
*MonadHybridFix6> y
\f.(\x.f(x x))(\x.f(x x))
*MonadHybridFix6> (runCont $ nor $ App (App y fact) three) id
\f.\x.f(f(f(f(f(f x)))))
```

we get the correct result, but if we define the uniform false NOR as the fixed point

```
fNor = eval fNor fNor return fNor fNor fNor
```

then running

```
*FalseNor> y
\f.(\x.f(x x))(\x.f(x x))
*FalseNor> (runCont $ falseNor $ App (App y fact) three) id
```

makes the evaluator to run forever. The false NOR strategy evaluates the body of λ-abstractions in the operator position, falling in the infinite evaluation of the Y combinator.

From this definition we may realise that something is wrong. By the rules in Section 2.8 the operator is first evaluated with CBN and then the *result* is evaluated with NOR. The above definition does not reflect these rules, because in the second place we are not evaluating the *result*, but the unevaluated operator. We must take this into account and pass the composition of `cbn` and `nor` as the `op2` parameter. In the monadic evaluator this can be done with the standard monadic operator:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
```

resulting the hybrid strategies

```
nor = eval nor cbn return nor (cbn >=> nor) nor
ha  = eval ha  cbv ha     ha  (cbv >=> ha)  ha
hn  = eval hn  he  return hn  (he >=> hn)   hn
```

In this case the two NOR strategies will be equivalent, because for any expression E, evaluating it with CBN and then with NOR leads to the same result than evaluating it directly with NOR. The same happens with HE and HN and with CBV and HA.

The tests can be run over an hybrid evaluation order. Let's take HA as an example.

```
*MonadHybridCompFix6> (runCont $ ha ident) id
\x.x
*MonadHybridCompFix6> (runCont $ ha redex) id
x
*MonadHybridCompFix6> (runCont $ ha capture) id
\y1.y
*MonadHybridCompFix6> (runCont $ ha free) id
(x y)(z y)
*MonadHybridCompFix6> (runCont $ ha varApp) id
x(\x.y)
*MonadHybridCompFix6> (runCont $ ha two) id
\f.\x.f(f x)
*MonadHybridCompFix6> (runCont $ ha omega) id
  C-c C-cInterrupted.
*MonadHybridCompFix6> (runCont $ ha lamOmega) id
  C-c C-cInterrupted.
*MonadHybridCompFix6> (runCont $ ha constOmega) id
  C-c C-cInterrupted.
*MonadHybridCompFix6> :{
*MonadHybridCompFix6| (runCont $
*MonadHybridCompFix6| ha $ App (App y fact) three)
*MonadHybridCompFix6| id
*MonadHybridCompFix6| :}
  C-c C-cInterrupted.
*MonadHybridCompFix6> :{
*MonadHybridCompFix6| (runCont $
*MonadHybridCompFix6| ha $ App (App (App z fact_d) three)
*MonadHybridCompFix6| (Var "_")) id
*MonadHybridCompFix6| :}
\f.\x.f(f(f(f(f(f x)))))
*MonadHybridCompFix6>
```

The behaviour is very similar to CBV, except that it yields NF, so the evaluation of `two` and `fact three` yield a Church numeral.

We have already seen that we can dispense the `su` parameter, leaving a space of five dimensions for the evaluators defined by single recursion. A parametric evaluator with orthogonal parameters is

```
type Ev m = SLTerm -> m SLTerm
eval :: Monad m => Ev m -> Ev m -> Ev m ->
        Ev m -> Ev m -> Ev m
eval la op1 ar1 op2 ar2 v@(Var s) = return v
```

```
eval la op1 ar1 op2 ar2 (Lam v b) =
    do b' <- la b
       return $ Lam v b'
eval la op1 ar1 op2 ar2 (App m n) =
    do m' <- op1 m
       case m' of
         (Lam v b) -> let su = eval la op1 ar1 op2 ar2
                      in do n' <- ar1 n
                            su $ subst v b n'
         _         -> do m'' <- op2 m
                         n'' <- ar2 n
                         return $ App m'' n''
```

with the fixed points:

```
triv = eval return return return return return
aor  = eval aor    aor    aor    aor    aor
cbv  = eval return cbv    cbv    cbv    cbv
cbn  = eval return cbn    return return return
he   = eval he     he     return return return
nor  = eval nor    cbn    return nor    nor
ha   = eval ha     cbv    ha     ha     ha
hn   = eval hn     he     return hn     hn
```

We conjecture that those five parameters are orthogonal. If this is true, then we have a space of $2^5 = 32$ evaluation orders which are definable as fixed points using single recursion. It contains, among others, the ones found in the literature, except the hybrid ones. We call this space the β-hypercube. With the mutually recursive fixed points the number of evaluators raise to the infinite. We must consider also the possibility of composite parameters done before. For some hybrid evaluation orders, a denotational equivalent version can be defined without composite parameters, although it will not be operationally equivalent.

The non-orthogonal parametric evaluator with five parameters is:

```
type Ev m = SLTerm -> m SLTerm
eval :: Monad m => Ev m -> Ev m -> Ev m ->
        Ev m -> Ev m -> Ev m
eval la op1 ar1 op2 ar2 v@(Var s) = return v
eval la op1 ar1 op2 ar2 (Lam v b) =
    do b' <- la b
       return $ Lam v b'
```

```
eval la op1 ar1 op2 ar2 (App m n) =
    do m' <- op1 m
       case m' of
         (Lam v b) -> let su = eval la op1 ar1 op2 ar2
                      in do n' <- ar1 n
                            su $ subst v b n'
         _         -> do m'' <- op2 m'
                         n'' <- ar2 n
                         return $ App m'' n''
```

and the fixed points

```
aor  = eval aor    aor    aor    return return
cbv  = eval return cbv    cbv    return return
cbn  = eval return cbn    return return return
he   = eval he     he     return return return
nor  = eval nor    cbn    return nor    nor
ha   = eval ha     cbv    ha     ha     ha
hn   = eval hn     he     return hn     hn
```

The space define by this evaluator is not an hypercube, cause the parameters are non-orthogonal. As before, some of its fixed points are denotationally equivalent. The evaluation order:

```
aor' = eval aor' aor' aor' aor' aor'
```

is equivalent to the `aor` defined above. Thus, evaluation orders are no longer 32.

Five parameters are still a lot, and some subsets of the mutually recursive orders can be stripped out. We want to reduce the number of parameters and to be able to define some hybrid strategies without mutual recursion. We would like to dispense with `op1`, assuming that every order evaluates the operator, although in different ways. This is reasonable because we want to discover opportunities for applying the β-rule. We must ask what are the different orders doing with the operator, and find the differences between uniform and hybrid strategies.

What NOR is not doing when processing the operator is to evaluate under λ-abstractions. Everything else is the same than in the false NOR, which can be uniformly defined. We can factor that out and substitute the parameters `la`, `op1`, `op2` by the parameters `la1` and `la2`, which stand for evaluation under λ-abstraction in the position of the operator and in other positions respectively. Doing something similar with `ar1` and `ar2` leads to an evaluator with just four parameters: b

```
eval :: Monad m => Ev m -> Ev m ->
        Ev m -> Ev m -> Ev m
eval la1 la2 ar1 ar2 v@(Var s)  = return v
eval la1 la2 ar1 ar2 (Lam v b)  = do b' <- la2 b
                                     return $  Lam v b'
eval la1 la2 ar1 ar2 (App
                      (Lam s (Var s'))
                      (Var s'')) = if s == s' then
                                       return $ Var s''
                                   else
                                       return $ Var s'
eval la1 la2 ar1 ar2 (App m n)   =
    do test' <- la1 test
       test'' <- ar1 test
       let la2' = select (test' /= test) la2
       let ar2' = select (test'' /= test) ar2
       m' <- eval la1 la2' ar1 ar2' m
       case m' of
         (Lam v b) ->
            do n' <- ar1 n
               eval la1 la2 ar1 ar2 $ subst v b n'
         _         ->
            do m'' <- eval la1 la2 ar1 ar2 m'
               n'' <- ar2 n
               return $ App m'' n''
```

with the following fixed points:

```
cbn  = eval return return return return
he   = eval he     return return return
cbv  = eval return return cbv    cbv
aor  = eval aor    aor    aor    aor
nor  = eval return nor    return nor
ha   = eval return ha     ha     ha
hn   = eval hn     hn     return hn
```

This evaluator breaks a little with the previous ones. It must test if some of the parameters are significant, this is, different from `return`. In the recursive invocations, the parameters are altered depending in that test.

When evaluating an application, it works in two stages: first it evaluates the operator with the altered parameters and then it pattern-matches over the result. If it is a λ-abstraction, then the substitution occurs. Otherwise

it will go on evaluating with the original parameters. The parameters are altered in the following way: if `la1` and `ar1` are equivalent to `return` then the altered `la2` and `ar2` must be `return`, respectively. This is consistent with the fact that `la1` and `ar1` define the behaviour in the operator position, this is, before doing the substitution.

The following code shows the auxiliary functions:

```
test = App (Lam "x" (Var "x")) (Var "y")
select :: Monad m => Bool -> Ev m -> Ev m
select True  ev = ev
select False ev = return
```

To decide if a parameter is different form `return`, the evaluator applies it to a test expression. The test expression is just the redex $(\lambda x.x)\, y$, which must be reduced to y for every definable evaluation order. A special pattern case `(App (Lam s (Var s')) (Var s''))` is needed. Without this case, the evaluator will loop infinitely trying to evaluate `redex`. The function `select`, which takes a boolean and an evaluator and selects between the evaluator or `return`, is used to select the altered parameters.

We can run the CBV fixed point of the parametric evaluator with four parameters.

```
*MonadFix4> (runCont $ cbv ident) id
\x.x
*MonadFix4> (runCont $ cbv redex) id
x
*MonadFix4> (runCont $ cbv capture) id
\y1.y
*MonadFix4> (runCont $ cbv free) id
(x y)(z y)
*MonadFix4> (runCont $ cbv varApp) id
x(\x.y)
*MonadFix4> (runCont $ cbv two) id
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*MonadFix4> (runCont $ cbv omega) id
  C-c C-cInterrupted.
*MonadFix4> (runCont $ cbv lamOmega) id
\x.(\x.x x)(\x.x x)
*MonadFix4> (runCont $ cbv constOmega) id
  C-c C-cInterrupted.
*MonadFix4> (runCont $ cbv $ App (App y fact) three) id
  C-c C-cInterrupted.
```

```
*MonadFix4> :{
*MonadFix4| (runCont $
*MonadFix4| cbv $ App (App y fact) three)
*MonadFix4| id
*MonadFix4| :}
  C-c C-cInterrupted.
*MonadFix4> :{
*MonadFix4| (runCont $
*MonadFix4| cbv $ App (App (App z fact_d) three) (Var "_"))
*MonadFix4| id
*MonadFix4| :}
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
(((\f.\x.x)f)x))f))f))f)
*MonadFix4>
```

The results are the expected for CBV.

Alternatively, the parameters could have been replaced by booleans:

```
type Ev m = SLTerm -> m SLTerm
eval :: Monad m => Bool -> Bool -> Bool -> Bool -> Ev m
eval la1 la2 ar1 ar2 v@(Var s)          = return v
eval la1 la2 ar1 ar2 (Lam v b)          =
    do b' <- if la2
                then eval la1 la2 ar1 ar2 b
                else return b
       return $  Lam v b'
eval la1 la2 ar1 ar2 (App m n) =
    do let la2' = if not la1 then False else la2
       let ar2' = if not ar1 then False else ar2
       m' <- eval la1 la2' ar1 ar2' m
       case m' of
         (Lam v b) ->
            do n' <- if ar1
                        then eval la1 la2 ar1 ar2 n
                        else return n
               eval la1 la2 ar1 ar2 $ subst v b n'
         _          ->
```

```
do m'' <- eval la1 la2 ar1 ar2 m'
   n'' <- if ar2
             then eval la1 la2 ar1 ar2 n
             else return n
   return $ App m'' n''
```

with the fixed points:

```
cbn  = eval False False False False
he   = eval True  False False False
cbv  = eval False False True  True
aor  = eval True  True  True  True
nor  = eval False True  False True
ha   = eval False True  True  True
hn   = eval True  True  False True
```

We prefer to use the version with parameters as evaluators, since it lets the definition of mutually recursive fixed points and we think it has more generality than the version with the booleans. This has an additional cost, because we must decide the significance of some parameters inside our object language, with the computational overhead this entails. This also makes the code a little more unclear, because of the `select` function and the special pattern case which avoids infinite looping.

This evaluator allows the definition of hybrid evaluation orders seen in Section 2.8, but in an ad hoc way. For the operator position, it should be possible to select the evaluation of the argument both before the substitution and after it. Here the evaluation of the argument before the substitution imposes the evaluation of the argument in the operator in both places, shadowing some evaluation orders which may be interesting. This reflects a pragmatic decision when reducing the number of the arguments while still being able to define some evaluation orders, but leads to an incomplete and ad hoc solution. However, that decision could fairly reflect the peculiarities of the problem and be very profitable, since every one of the foremost evaluation orders can be defined with it.

We have introduced some dependencies between the parameters, so we cannot assert that the parameters are orthogonal. It does not seem that any combination of parameters will be equivalent to any other at first sight, but we will need further research to proof formally one thing or the opposite.

3.4. Compositional Evaluator

In this section we present a rather different approach, where instead of a generic evaluator we write some elemental features that can be combined in multiple ways, giving all the possible evaluation orders. The combination of the elemental features is done via a mechanism which models inheritance. This mechanism is known as mixins.

3.4.1. Mixins

Mixins define a model of inheritance. A ***mixin*** is an abstract subclass that may be used to specialise the behaviour of a variety of parent classes [5]. In a functional programming setting, a mixin is a function taking a parameter which stands for the *super* function. The mixin invokes super or specialises its behaviour as needed. Some mixin models include also another parameter which stands for the function itself, allowing the recursive invocation of that function. This parameter works as the *self* pointer in some object-oriented programming languages. A binary composition operator `(<*>)` and a `zero` mixin exist. The `zero` mixin defines a void behaviour, and along with `(<*>)` induces a monoidal structure for the mixins. A fixed point operator (`mixin`) is used to retrieve an ordinary function given the mixin and its parameters.

In this thesis we use the following definition of mixins, adapted from [6]:

```
type Mixin s = s -> s -> s

(<+>) :: Mixin s -> Mixin s -> Mixin s

f <+> g = \super this -> f (g super this) this

mZero :: Mixin s
mZero super this = super

mixin :: Mixin s -> s
mixin f = let m = mixin f in f m m
```

From within a mixin, `super` allows the access to the ancestor function, whereas `this` accesses the same function. The `mixin` combinator builds up a fixed point which accounts for the 'inheritance' structure. It induces the following graph 3.1.

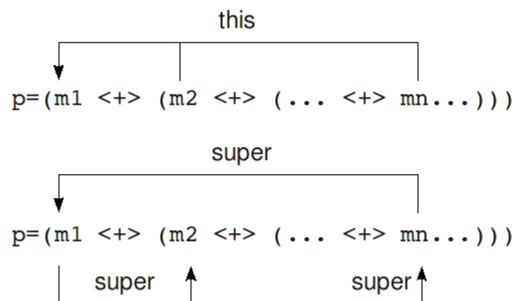

Figure 3.1: Mixin inheritance graph

3.4.2. Uniform Evaluation Orders with Mixins

The elementary mixins which define the uniform evaluation orders are the following:

```
type Ev m = SLTerm -> m SLTerm

betaRed :: Monad m => Mixin (Ev m)
betaRed super this (App (Lam v b) n) = this $ subst v b n
betaRed super this o@_                = return o

la :: Monad m => Mixin (Ev m)
la super this (Lam v b) = do b' <- this b
                             return $ Lam v b'
la super this o@_       = super o

op1 :: Monad m => Mixin (Ev m)
op1 super this (App m n) = do m' <- this m
                              super $ App m' n
op1 super this o@_       = super o

ar1 :: Monad m => Mixin (Ev m)
ar1 super this (App m n) = do n' <- this n
                              super $ App m n'
ar1 super this o@_       = super o

op2 :: Monad m => Mixin (Ev m)
op2 super this a@(App (Lam v b) n) = super a
op2 super this (App m n)           = do m' <- this m
                                        super $ App m' n
```

```
op2 super this o@_                 = super o

ar2 :: Monad m => Mixin (Ev m)
ar2 super this a@(App (Lam v b) n) = super a
ar2 super this (App m n)           = do n’ <- this n
                                        super $ App m n’
ar2 super this o@_                 = super o
```

`betaRed` is the basic mixin from which every evaluator must inherit. It just computes the big-step β-rule, but only in head position. It corresponds to the basics of the parametric evaluator plus the `su` parameter. It must appear as the basic one which is extended (or inherited).

`la` filters λ-abstractions in head position and evaluates its body. Otherwise it invokes `super` to evaluate the expression. It corresponds to the `la` parameter of the parametric evaluator. It does not matter the place where this mixin appears in the inheritance hierarchy, but we will place it, by convention, just under `betaRed`.

`op1` filters applications, evaluates their operator and invoke `super` over the application of the evaluated operator to the argument. For other expressions it invokes `super` over them. It corresponds to the `op1` parameter of the parametric evaluator.

`ar1`, like the previous one, filters applications, evaluates their argument and invokes `super` over the application of the operator to the evaluated argument. Otherwise it invokes `super`. It corresponds to the `ar1` parameter.

`op1` must appear under `ar1` if the intended operational semantics is to evaluate the operand before the argument.

`op2` passes applications of λ-abstractions to an argument, which must be β-reduced, to `super`. For other applications, it evaluates the operand and then invokes `super` over the application of the evaluated operand and the argument. For any other expressions it invokes `super` over them. It corresponds to the `op2` parameter.

`ar2`, like the previous one, delivers applications of λ-abstractions to its ancestor, and for other applications it evaluates its argument and then calls `super` over the application of the operator and the evaluated argument. It corresponds to the `ar2` parameter.

For the last two mixins to work as intended, they must appear over the `op1` and `ar1` component, giving them the opportunity to do their job in first place. Both mixins pass expressions with the pattern `(App (Lam v b) n)` to the ancestor, since they must be β-reduced. They implement also the case of an application whose operand is not a λ-abstraction. As a whole, `op1`, `ar1` reduce the operator and the argument before the substitution, while `op2` and

`ar2` reduce the operator and the argument in the case the operator is not a λ-abstraction. This is equivalent to the way the parametric evaluator works with the parameters `op1`, `ar1`, `op2` and `ar2`.

This elementary mixins can be composed with the `<+>` operator (note that `betaRed` must act as the basic feature), like in

```
cbn :: Monad m => Ev m
cbn = mixin (op1 <+> betaRed)
cbv :: Monad m => Ev m
cbv = mixin (op1 <+> (ar1 <+> (op2 <+> (ar2 <+> betaRed))))
aor :: Monad m => Ev m
aor = mixin (op1 <+> (ar1 <+> (op2 <+> (ar2 <+>
                                        (la <+> betaRed)))))
he :: Monad m => Ev m
he = mixin (op1 <+> (la <+> betaRed))
```

The order of appearance of each component follows the indications gave in the previous paragraph.

The order of the components only determines operational issues. They can be mixed up at will (except for `betaRed`, which is the basic one as it applies the β-rule and acts as the skeleton) and the denotational semantics will not be altered. Likewise, the order of the parameters is not important in the parametric evaluator.

We can check how the mixin CBV works with our tests.

```
*UniformMixin> (runCont $ cbv ident) id
\x.x
*UniformMixin> (runCont $ cbv redex) id
x
*UniformMixin> (runCont $ cbv capture) id
\y1.y
*UniformMixin> (runCont $ cbv free) id
(x y)(z y)
*UniformMixin> (runCont $ cbv varApp) id
x(\x.y)
*UniformMixin> (runCont $ cbv two) id
\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x)
*UniformMixin> (runCont $ cbv omega) id
  C-c C-cInterrupted.
*UniformMixin> (runCont $ cbv lamOmega) id
\x.(\x.x x)(\x.x x)
*UniformMixin> (runCont $ cbv constOmega) id
```

```
  C-c C-cInterrupted.
*UniformMixin> (runCont $ cbv $ App (App y fact) three) id
  C-c C-cInterrupted.
*UniformMixin> :{
*UniformMixin| (runCont $
*UniformMixin| cbv $ App (App y fact) three)
*UniformMixin| id
*UniformMixin| :}
  C-c C-cInterrupted.
*UniformMixin> :{
*UniformMixin| (runCont $
*UniformMixin| cbv $ App (App (App z fact_d) three) (Var "_"))
*UniformMixin| id
*UniformMixin| :}
\f.(\f.\x.f(((\f.\x.f(((\f.\x.f(((\f.\x.x)f)x))f)x))f)x))
((\f.(\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
((\f.(\y.\z.(((\y.\z.(((\f.\x.f(((\f.\x.f(((\f.\x.f
(((\f.\x.x)f)x))f)x))f)x))(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))
(\p.\q.q(p y)))((\p.\q.p)z))(\x.x))((\f.\x.f
(((\f.\x.x)f)x))f))f))f)
*UniformMixin>
```

This is the expected behaviour for CBV.

3.4.3. Hybrid Evaluation Orders with Mixins

The definition of hybrid evaluation orders with mixins requires non-standard mixin combinators. We extend the mixins of Section 3.4.1 with the following combinators:

```
type MixinM s = s -> s -> s -> s
split :: MixinM s -> Mixin s -> Mixin s -> Mixin s
split f g g' = \super this -> f (g super this)
                                (g' super this) this
(<*>) :: Mixin s -> Mixin s -> Mixin s
f <*> g = \super this -> let current = mixin $ f <*> g
                          in f (g super current) current
```

Function `split` takes three components, one special variant of mixins (`MixinM s`) with two different ancestors (`super` and `super'`) and two regular mixins (`Mixin s`) so it lets the definition of inheritance structures following the

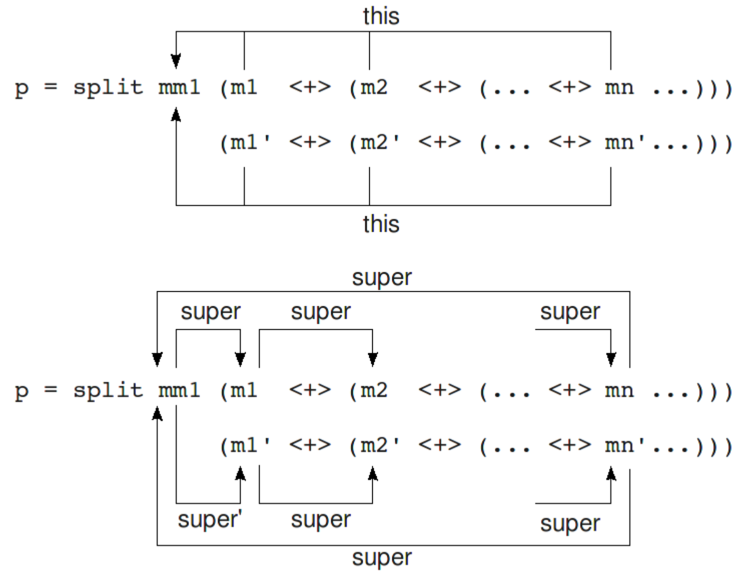

Figure 3.2: Mixin inheritance graph with `split` combinator.

graph in Figure 3.2. `(<*>)` truncates the come back of the `this` to the current component, as it can be seen in Figure 3.3. With this operators we can

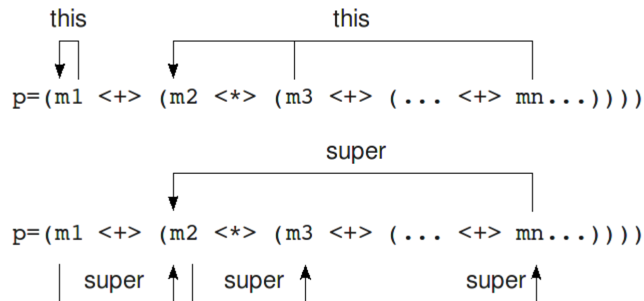

Figure 3.3: Mixin inheritance graph with `(<*>)`.

easily define the hybrid strategies, using two inheritance stacks and one component which selects a particular stack when evaluating the operator before the substitution.

```
nor :: Monad m => Ev m
nor = mixin (split select
             (op2 <+>(ar2 <+> (la <+> betaRed)))
             (op1 <*> betaRed))
ha :: Monad m =>  Ev m
```

```
ha = mixin (split select
            (op2 <+> (ar2 <+> (ar1 <+> (la <+> betaRed))))
            (op2 <*> (ar2 <+> (op1 <+> (ar1 <+> betaRed)))))
hn :: Monad m => Ev m
hn = mixin (split select
            (op2 <+> (ar2 <+> (la <+> betaRed)))
            (op1 <*> (la <+> betaRed)))
```

The `select` mixin is below every other else in the inheritance hierarchy. This mixin has type `MixinM s`, which means that it has two different ancestors. It evaluates the operator of applications with one ancestor we refer to as the operator branch. The mixin in the bottom of the operator branch uses `(<*>)` to truncate the comeback of `this`, making it to point to the operator branch, instead of coming back to the `select` mixin. It invokes the other ancestor, which we call the regular branch, to evaluate the rest of expressions. The mixins in the regular branch use regular composition (¡+¿), so `this` in this branch points to the `select` mixin, allowing the recursive invocation of the hybrid strategy.

With the non-standard mixin combinators we can define arbitrary inheritance structures. The way the hybrid strategies work let us define them using only one `split` combinator. Other more complex strategies may require more intricate inheritance structures.

If we run the battery test over an hybrid evaluation order, for example HA, we obtain:

```
*HybridMixin> (runCont $ ha ident) id
\x.x
*HybridMixin> (runCont $ ha redex) id
x
*HybridMixin> (runCont $ ha capture) id
\y1.y
*HybridMixin> (runCont $ ha free) id
(x y)(z y)
*HybridMixin> (runCont $ ha varApp) id
x(\x.y)
*HybridMixin> (runCont $ ha varApp) id
(x y)(z y)
*HybridMixin> (runCont $ ha two) id
\f.\x.f(f x)
*HybridMixin> (runCont $ ha omega) id
  C-c C-cInterrupted.
*HybridMixin> (runCont $ ha lamOmega) id
```

```
  C-c C-cInterrupted.
*HybridMixin> (runCont $ ha constOmega) id
  C-c C-cInterrupted.
*HybridMixin> :{
*HybridMixin| (runCont $
*HybridMixin| ha $ App (App y fact) three)
*HybridMixin| id
*HybridMixin| :}
  C-c C-cInterrupted.
*HybridMixin> :{
*HybridMixin| (runCont $
*HybridMixin| ha $ App (App (App z fact_d) three) (Var "_")
*HybridMixin| ) id
*HybridMixin| :}
\f.\x.f(f(f(f(f(f x)))))
*HybridMixin>
```

which is the expected under HA evaluation order.

Chapter 4

Interpreters for the λ-calculus

In this chapter we show and discuss interpreters for the λ-calculus overviewing previous work by Reynolds and commenting on the choice of model. We show interpreters for different evaluation orders that use environments, and also show a parametric interpreter. We discuss the relationships with the β-hypercube.

4.1. Reynolds's Definitional Interpreters

Reynolds [19] proposed a definitional interpreter for a pure functional language (purely applicative language, in Reynold's words). It is definitional because the interpreter is intended to define the semantics for that language. The evaluation order chosen for a language determines the semantics of some programs in that language. He noted that this is the case with an interpreter, and that the evaluation order in the implementation language will affect the semantics of the object language, which is what the interpreter is trying to define.

Reynolds focused on some previous definitions of pure functional languages. McCarthy's definition of LISP [11], Landin's SECD machine [10], Reynold's definition of GEDANKEN [18] and an unpublished work by Morris and Wardsworth. Reynolds classified them attending to two different criteria: the use of higher-order functions in the definition of the interpreter and the dependency with respect to the evaluation order in the implementation language.

Evaluation order dependence	Higher-order	
	Yes	No
Yes	GEDANKEN	McCarthy's LISP
No	Morrish-Wardsworth's method	SECD machine

In his paper he related the classes of definitional interpreters, giving two transformations: *continuation-passing style* (CPS), which makes the semantics implemented by the interpreter independent of the evaluation order in the implementation language and *defunctionalization*, which converts the different uses of a function passed as argument into instances of a data type, turning a higher order interpreter into a first order.

He started with a direct implementation of the interpreter, which used higher-order and was evaluation-order dependent and applied the two transformation to it giving the four different possibilities in the table.

4.2. Meta-Circular Interpreter

The straightforward implementation is a function `interp` which takes an environment and an expression and produces a value. The language has the syntax in:

```
data Exp = Cons Int
         | Bo Bool
         | Var String
         | Lam String Exp
         | App Exp Exp
         | Cond Exp Exp Exp
         | Lrec String String Exp Exp

data Val = C Int
         | B Bool
         | F Fun

type Fun = Val -> Val
```

The language embeds λ-calculus and has also constructs for integer and boolean constants, conditionals and *letrec* definitions. The values are constants or functions from values to values.

The environments are implemented the following way:

```
type Env = String -> Val

emptyEnv :: Env
emptyEnv s = undefined

addVar :: String -> Val -> Env -> Env
addVar s v e = \r -> if r == s then v else e r
```

The environment binds variables to values. It is used to implement substitution and reduction. When looking up a variable which is not bound in the environment the interpreter returns $undefined$. This is because the language does not allow free variables, so open expressions are not valid programs.

The following is the Haskell implementation of the first Reynolds' metacircular interpreter (MCI).

```
interp :: Env -> Exp -> Val
interp e (Cons n)       = C n
interp e (Bo b)         = B b
interp e (Var s)        = e s
interp e (App m n)      =
    apply (interp e m) (interp e n)
interp e (Lam v b)      =
    F $ \t -> interp (addVar v t e) b
interp e (Cond p c a)   = if (interp e p) == (B True) then
                              (interp e c)
                          else
                              (interp e c)
interp e (Lrec d v b i) =
    let e' = addVar d (F $ \t -> interp (addVar v t e') b) e
    in interp e' i

apply :: Val -> Val -> Val
apply (F f)  n    = f n
```

λ-abstractions are interpreted in a peculiar way: a Haskell anonymous function is returned, which invokes `interpret` in its body. This stops evaluation under λ-abstraction because the body is not *evaluated* until some value is

applied to the Haskell function. One may think that the code can be rearranged to evaluate under λ-abstraction too, but the way it works does not allow so. Interpreting a λ-abstraction requires to bound the value passed as argument in the environment, and then interpreting the body of the λ-abstraction with this new environment. This is analogous to the evaluation of the substitution in the parametric evaluators. Interpreting the body of a λ-abstraction implies the interpreter has already done the substitution (in a metaphoric sense, because the interpreter use the environment rather than substitution). The interpretation of the body of λ-abstractions occurs *on the fly*. This mechanism accounts also for β-reduction, since the operator of an application is always a λ-abstraction (otherwise the expression will not be valid). This means that the evaluation order implemented by the interpreter is CBV.

MCI uses an eval-apply schema. The expressions are interpreted before passing them to `apply`. The `apply` function takes values and if the first one is a function, it is applied to the second one. Other cases are not considered because they stand for invalid expressions.

4.2.1. Programing with the MCI

We can interpret λ-expressions with the MCI,

```
*MCI> interp emptyEnv $ ident
<function>
*MCI>
```

but the result will be very often a function, which is rendered by the pretty-printer as the string `<function>`. We must interpret an expression whose result is a constant to appreciate the MCI at work.

```
*MCI> interp emptyEnv $ App ident (Cons 3)
3
v*MCI>
```

Working with constants is very interesting, because it optimises the computation of arithmetic and logical operations and gets rid of Church encoding, making the program less obfuscated. The possibility of working with primitive data (Haskell's data, not Reynolds applicative language data) is also available, using `apply` as a primitive application function.

```
*MCI> apply (interp emptyEnv $ ident) (C 3)
3
*MCI>
```

Notice that we must pass *values* to the `apply` function. Thus the constant 3 is passed as a value `C 3` instead of as an expression `Cons 3` (`C` is the constructor of values whereas `Cons` is the constructor of constant expressions). The same trick applies to functions. We can "implement" a Church decoder applying the interpretation of a natural number to a function which computes the successor and the 0 value.

```
*MCI> let suc = F $ \(C n) -> C $ n + 1
*MCI> apply (apply (interp emptyEnv three) suc) (C 0)
3
*MCI>
```

Alternatively, we can build up an environment which contains primitive functions and constants for the arithmetic and logical operations.

```
arit = addVar "zero" (C 0) $
       addVar "true" (B True) $
       addVar "false" (B False) $
       addVar "suc" (F $ \(C n) -> C $ n + 1) $
       addVar "pre" (F $ \(C n) -> C $ n - 1) $
       addVar "add" (F $ \(C m) -> F $ \(C n) -> C $ m + n) $
       addVar "mult" (F $ \(C m) -> F $ \(C n) -> C $ m * n) $
       addVar "expo" (F $ \(C m) -> F $ \(C n) -> C $ m ^ n) $
       addVar "equal" (F $ \(C m) ->
                         F $ \(C n) -> if m == n then B True
                                       else B False) $
       addVar "isZero" (F $ \(C n) -> if n == 0 then B True
                                     else B False) $
       addVar "neg" (F $ \(B b) -> B $ not b) $
       addVar "&" (F $ \(B p) -> F $ \(B q) -> B $ p && q) $
       addVar "|" (F $ \(B p) -> F $ \(B q) -> B $ p || q) $
       emptyEnv
```

Then we can use those "reserved functions" in any expression.

```
*MCI> interp arit $ App (App three (Var "suc")) (Var "zero")
3
*MCI>
```

With this environment `Cons 0` is denotationally equivalent (but operationally different) to `Var "zero"`.

```
*MCI> interp arit $ App (App three (Var "suc")) (Cons 0)
3
*MCI>
```

Constant expressions are an interpreter's built in feature. With the built in features we don't need to define them again if we change our environment, but the features cannot change for different settings. With the reserved functions we can change them for different setting, but we need to include them in every environment. This raises the question, is it best to include built in features or to let the definition of reserved constants and functions in the environment? In this case the differences are not very significant, but in a production interpreter each feature must be considered individually.

There are some issues regarding the way recursion is implemented. We can just use the λ-calculus fixed point combinators and implement it as we have done in Chapter 3, but we must explicitly manage Church encoding and decoding using primitive values:

```
*MCI> let factThree = App (App (App z fact_d) three) (Var "_")
*MCI> apply (apply (interp emptyEnv factThree) suc) (C 0)
6
*MCI>
```

We are using the Z combinator and the dummy arguments because MCI has an evaluation order which seems to implements strict semantics. However, as this is a naive implementation which is evaluation order dependent, we may have used the Y combinator instead (since Haskell evaluation order implements non-strict semantics).

```
*MCI> let factThree = App (App y fact) three
*MCI> apply (apply (interp emptyEnv factThree) suc) (C 0)
6
*MCI>
```

We are not using the improvement that the *letrec* entails. We can rewrite the *fact* function:

```
factLr = Lrec "f" "n"
          (Cond (App (Var "isZero") (Var "n"))
           (Cons 1)
           (App (App (Var "mult") (Var "n"))
            (App (Var "f") (App (Var "pre") (Var "n")))))
          (Lam "x" (App (Var "f") (Var "x")))
```

and then interpret it with the `arit` environment:

```
*MCI> factLr
letrec f=\n.if isZero n then 1 else (mult n)(f(pre n)) in \x.f x
```

```
*MCI> interp arit $ App factLr (Cons 3)
6
*MCI>
```

Reynolds gave other three versions of the MCI. One which is first-order, one which is evaluation order independent and one which is first-order and evaluation order independent. We are not going to derive those versions here. First we will restrict the interpreter to work with pure untyped λ-calculus.

4.3. MCI for the Pure λ-calculus

The MCI can be easily transformed to work only with λ-calculus. We eliminate constant, conditional and *letrec* expressions, and define a new domain for its values, ranging over λ-expressions and functions from values to values.

```
data Val = T SLTerm
         | F Fun

type Fun = Val -> Val

type Env = String -> Val

interp :: Env -> SLTerm -> Val
interp e (Var s)   = e s
interp e (Lam v b) = F (\t -> interp (addVar v t e) b)
interp e (App m n) = apply (interp e m) (interp e n)

apply :: Val -> Val -> Val
apply (F f)  n    = f n
```

Interpreting λ-expressions is discouraging, because it does not allow free variables, and functions are printed as `<function>`.

```
*MCILCalc> interp emptyEnv ident
<function>
*MCILCalc> interp emptyEnv redex
*** Exception: Prelude.undefined
*MCILCalc> interp emptyEnv capture
<function>
*MCILCalc> interp emptyEnv free
*** Exception: Prelude.undefined
```

```
*MCILCalc> interp emptyEnv varApp
*** Exception: Prelude.undefined
*MCILCalc> interp emptyEnv two
<function>
*MCILCalc> interp emptyEnv omega
  C-c C-cInterrupted.
*MCILCalc> interp emptyEnv lamOmega
<function>
*MCILCalc> interp emptyEnv constOmega
*** Exception: Prelude.undefined
*MCILCalc> interp emptyEnv $ App (App y fact) three
<function>
*MCI> :{
*MCI| interp emptyEnv $
*MCI| App (App (App z fact_d) three) (Var "_")
*MCI| :}
<function>
*MCI>
```

A valid result is displayed as `<function>`, because the λ-expressions does not contain constants. Open expressions are not allowed, so `redex` and `varApp` fail. `omega` yields undefined and `lamOmega` terminates, because MCI stops interpretation under λ-abstraction (what happens is that λ-abstractions are interpreted on the fly, as seen before). `constOmega` returns (the MCI is the naive version and does not implement evaluation-order independence) but an invalid result, because it has free variables. Recursion works both with the Y and Z combinators, because Haskell laziness interferes the semantics.

It seems we can do very little with this interpreter, but this is not true. We must resort to primitive application (`apply`) and primitive values

```
*MCILCalc> let intIdent = interp emptyEnv ident
*MCILCalc> apply ident (T $ Var "x")
x
*MCILCalc>
```

or to environments with primitive values.

```
*MCILCalc> let newEnv = addVar "x" (T $ Var "x") emptyEnv
*MCILCalc> interp newEnv $ App ident (Var "x")
x
*MCILCalc>
```

A version can be written which works fancier with free variables, using a different environment.

4.4. Interpreter with Free Variables

When a variable which is looked up does not appear in the environment, it returns `undefined`, because this is interpreted as an invalid case. We can modify the environment to return the variable being looked-up when it is not found. This make the language definition slightly different, because a free variable is not an exceptional situation. A free variable can now be applied to some expression. We must extend also the cases in the `apply` function to manage this scenario.

```
data Val = T SLTerm
         | F Fun

type Fun = Val -> Val

instance Show Val where
    show (T t) = show t
    show (F f) = "<function>"

type Env = String -> Val

emptyEnv :: Env
emptyEnv s = (T $ Var s)

addVar :: String -> Val -> Env -> Env
addVar s v e = \r -> if r == s then
                         v
                     else
                         e r

interp :: Env -> SLTerm -> Val
interp e (Var s)   = e s
interp e (Lam v b) = F (\t -> interp (addVar v t e) b)
interp e (App m n) = apply (interp e m) (interp e n)

apply :: Val -> Val -> Val
apply (F f)  n    = f n
apply (T m) (T n) = T $ App m n
```

Putting it to work with some λ-calculus expressions we get:

```
*MCIFreeVars> interp emptyEnv ident
```

```
<function>
*MCIFreeVars> interp emptyEnv redex
x
*MCIFreeVars> interp emptyEnv capture
<function>
*MCIFreeVars> interp emptyEnv free
(x y)(z y)
*MCIFreeVars> interp emptyEnv varApp
*** Exception: Interp/MCIFreeVars.lhs:(36,2)-(37,32):
Non-exhaustive patterns in function apply

*MCIFreeVars> interp emptyEnv two
<function>
*MCIFreeVars> interp emptyEnv omega
  C-c C-cInterrupted.
*MCIFreeVars> interp emptyEnv lamOmega
<function>
*MCIFreeVars> interp emptyEnv constOmega
y
*MCIFreeVars> interp emptyEnv $ App (App y fact) three
<function>
*MCIFreeVars> :{
*MCIFreeVars| interp emptyEnv $
*MCIFreeVars| App (App (App z fact_d) three) (Var "_")
*MCIFreeVars| :}
<function>
*MCIFreeVars>
```

This interpreter is as powerful as the previous one but it lets an easier manipulation of open expressions, since the environment can deal with them. It still lacks the ability to visualise values which are functions. We would like to know the irreducible expression denoting the value returned by the interpret, but this is not possible because function values discard the information of the λ-abstractions denoting them. It also does not support functions in the place of the argument where the operand is a free variable.

This does not fulfil with the concept of evaluation we have seen in Chapter 3. We need a way to interpret to values which contain the normal form as a λ-expression.

4.5. MCI with Applications of Free Variables

We want our values to contain also irreducible λ-expressions (no matter which notion of irreducible expression) which denote that value. Thus, we may compare the interpreter and its evaluation order with the evaluators in Chapter 3.

When interpreting a λ-abstraction we translate it to an anonymous Haskell function, loosing forever the information about its formal parameter and his body. This is not completely true. Inside the Haskell function there is an invocation to `interp` where the body of the λ-abstraction is interpreted (remember the on the fly interpretation of λ-abstractions, in Section 4.2), taking an environment which binds the formal parameter. We can implement a `toSLTerm` function taking a value and building a λ-abstraction from it. To do so we just pick a variable `x` and return the λ-abstraction which has this variable as formal parameter and the application of the function value to `Var "x"` as body (where this body is recursively *translated* into `SLTterm`).

```
toSLTerm (T t) = t
toSLTerm (F f) = (Lam "x" (toSLTerm $ f (Var "x"))
```

This gives us a λ-expression which resembles the irreducible expression denoting the value, except for the formal parameter, which is always the same. This cause a loss of information and possibly captures the free variables inside the body. We can cheat a little (for a fair purpose) and annotate every function with the name of the formal parameter of the λ-abstraction which gave it to life.

```
data Val = T SLTerm
         | F Fun String

type Fun = Val -> Val

instance Show Val where
    show = show . toSLTerm

toSLTerm :: Val -> SLTerm
toSLTerm (T t)   = t
toSLTerm (F f v) = let b = f $ T $ Var v
                   in Lam v $ toSLTerm b


type Env = String -> Val
```

```
emptyEnv :: Env
emptyEnv s = T $ Var s

addVar :: String -> Val -> Env -> Env
addVar s v e = \r -> if r == s then
                         v
                     else
                         e r

interp :: Env -> SLTerm -> Val
interp e (Var s)   = e s
interp e (Lam v b) =
   F (\t -> interp (addVar v t e) b) v
interp e (App m n) = apply (interp e m) (interp e n)

apply :: Val -> Val -> Val
apply (F f v) n = f n
apply (T m)   n = T $ App m $ toSLTerm n
```

We would need some kind of α-conversion to avoid capturing free variables, but this cannot be done easily, because we do not know the free and bound variables of a function values. This problem becomes apparent only when translating a vale to a `SLTerm` and showing it. The interpreter will compute the right thing, but when we show that thing we may give a term where capture of variables may occurs. We present the results of the interpreter but with this consideration:

```
*MCIApp> interp emptyEnv ident
\x.x
*MCIApp> interp emptyEnv redex
x
*MCIApp> interp emptyEnv capture
\y.y
*MCIApp> interp emptyEnv free
(x y)(z y)
*MCIApp> interp emptyEnv varApp
x(\x.y)
*MCIApp> interp emptyEnv two
\f.\x.f(f x)
*MCIApp> interp emptyEnv omega
  C-c C-cInterrupted.
```

```
*MCIApp> interp emptyEnv lamOmega
\x.  C-c C-cInterrupted.
*MCIApp> interp emptyEnv constOmega
y
*MCIApp> interp emptyEnv $ App (App y fact) three
\f.\x.f(f(f(f(f(f x))))))
*MCIApp> :{
*MCIApp| interp emptyEnv $
*MCIApp| App (App (App z fact_d) three) (Var "_")
*MCIApp| :}
\f.\x.f(f(f(f(f(f x))))))
*MCIApp>
```

We can observe the issue with variable capturing when interpreting `capture`. The function returned by the interpreter is right, but when we try to visualise we cannot apply the appropriate α-conversion to show a λ-expression denoting it accurately. Noticed when interpreting `lamOmega`, the `show` variable tries to do their job and it runs indefinitely.

Now we must ask what is the evaluation order implemented by the MCI. First we need a version which is evaluation order independent. We are not following Reynolds completely here. We provide Reynolds CPS version, and then we apply the strategies we saw in Chapter 3 to provide our own versions. We can derive interpreters for every semantics preserving strategy, but we have already done this exercise for the evaluators. We explore only the CPS and the monadic approach.

4.6. CPS MCI for Untyped λ-calculus

Following Reynolds' implementation of CPS we give the MCI with CPS.

```
data Val = T SLTerm
         | F Fun

type Fun = Val -> Cont -> Val

type Cont = Val -> Val

type Env = String -> Val

emptyEnv :: Env
emptyEnv s = undefined
```

```
addVar :: String -> Val -> Env -> Env
addVar s v e = \r -> if r == s then
                         v
                     else
                         e r

interp :: Env -> SLTerm -> Cont -> Val
interp e (Var s)   c = c $ e s
interp e (Lam v b) c = c $ F $ \a c ->
                               interp (addVar v a e) b c
interp e (App m n) c = interp e m (\(F f) ->
                                   interp e n (\a -> f a c))
```

This is an eval-case interpreter (the case expression is obviated by pattern matching in the anonymous function). An eval-apply equivalent interpreter is possible, but it will obfuscate the CPS.

As we discussed in Section 4.2 this interpreter (now fixed with the CPS) implements an evaluation order which leads to strict semantics, because the continuation in the `(App m n)` case is passed to the interpretation of the `n`, and this in turn is passed as a continuation to the interpretation of the `m`. Both `m` and `n` are interpreted before executing the continuation in the `(App m n)` case.

If we run the tests:

```
*MCILCalcCont> (interp emptyEnv ident) id
<function>
*MCILCalcCont> (interp emptyEnv redex) id
*** Exception: Prelude.undefined
*MCILCalcCont> (interp emptyEnv capture) id
<function>
*MCILCalcCont> (interp emptyEnv free) id
*** Exception: Prelude.undefined
*MCILCalcCont> (interp emptyEnv varApp) id
*** Exception: Prelude.undefined
*MCILCalcCont> (interp emptyEnv two) id
<function>
*MCILCalcCont> (interp emptyEnv omega) id
  C-c C-cInterrupted.
*MCILCalcCont> (interp emptyEnv lamOmega) id
<function>
*MCILCalcCont> (interp emptyEnv constOmega) id
```

```
  C-c C-cInterrupted.
*MCILCalcCont> (interp emptyEnv $ App (App y fact) three) id
  C-c C-cInterrupted.
*MCILCalcCont> :{
*MCILCalcCont| (interp emptyEnv $
*MCILCalcCont| App (App (App z fact_d) three) (Var "_")
*MCILCalcCont| ) id
*MCILCalcCont| :}
<function>
*MCILCalcCont>
```

The interpreter stops interpretation under λ-abstractions, because they are "translated" to Haskell on the fly. When interpreting `omega` the interpreter never stops. This is what we want, because any evaluation order must not terminate with `omega`. When interpreting `lamOmega` it returns a function, which will interpret `omega` when passing some argument to it. We must use the Z combinator and dummy arguments. The interpreter implements an eager strategy and will run forever with the Y combinator.

We can apply the value resulting when interpreting `lamOmega` to some other value. We do not have the primitive `apply` function, so we must build up a continuation.

```
*MCILCalcCont> let dummy = interp emptyEnv (Var "_")
*MCILCalcCont> let cont = (\(F f) -> dummy (\a -> f a id))
*MCILCalcCont> (interp emptyEnv lamOmega) cont
```

This runs forever, because now the body of `Lam "x" omega` is being interpreted.

4.7. Monadic MCI for Untyped λ-calculus

The MCI for λ-calculus in Section 4.3 can be improved by the use of monads.

```
data Val = T SLTerm
         | F Fun

type Fun = Monad m => Val -> m Val

type Env = String -> Val

emptyEnv :: Env
```

```haskell
emptyEnv s = undefined

addVar :: String -> Val -> Env -> Env
addVar s v e = \r -> if r == s then
                         v
                     else
                         e r


interp :: Monad m => Env -> SLTerm -> m Val
interp e (Var s)   = return $ e s
interp e (Lam v b) =
    return $ F $ \t -> interp (addVar v t e) b
interp e (App m n) = apply (interp e m) (interp e n)

apply :: Monad m => m Val -> m Val -> m Val
apply op ar = do m <- op
                 n <- ar
                 case m of
                   (F f) -> f n
```

Everything turns into monadic, including `apply` and the function inside the `Value` data type. This forces to call `apply` with a non-monadic style and to do the monadic application of the operator to the argument inside `apply`. This does not alter the CPS schema, it just delays the continuations to take effect inside `apply` instead of inside `interp`. For the body of lambda expressions we just invoke the interpreter recursively (we do not need to interpret a piece of the expression and then build the whole value, because the MCI interprets λ-abstractions on the fly).

```
*MonadMCILCalc> (runCont $ interp emptyEnv ident) id
<function>
*MonadMCILCalc> (runCont $ interp emptyEnv redex) id
*** Exception: Prelude.undefined
*MonadMCILCalc> (runCont $ interp emptyEnv capture) id
<function>
*MonadMCILCalc> (runCont $ interp emptyEnv free) id
*** Exception: Prelude.undefined
*MonadMCILCalc> (runCont $ interp emptyEnv varApp) id
*** Exception: Prelude.undefined
*MonadMCILCalc> (runCont $ interp emptyEnv two) id
```

```
<function>
*MonadMCILCalc> (runCont $ interp emptyEnv omega) id
  C-c C-cInterrupted.
*MonadMCILCalc> (runCont $ interp emptyEnv lamOmega) id
<function>
*MonadMCILCalc> (runCont $ interp emptyEnv constOmega) id
  C-c C-cInterrupted.
*MonadMCILCalc> (runCont $ interp emptyEnv $ App (App y fact) three) id
  C-c C-cInterrupted.
*MonadMCILCalc> :{
*MonadMCILCalc| (runCont $ interp emptyEnv $
*MonadMCILCalc| App (App (App z fact_d) three) (Var "_")
*MonadMCILCalc| ) id
*MonadMCILCalc| :}
<function>
*MonadMCILCalc>
```

The behaviour of this interpreter is similar to the CPS one. We have an eager evaluation order which is strict on applications. The evaluation order is CBV. This evaluation order implements strict semantics and is suitable for recursion with the Z combinator.

Both the CPS and the monadic MCI could be extended with the free variables (Section 4.4) and the applications of free variables (Section 4.5) mechanisms. Nevertheless we will focus on the basic version to give the parametric interpreter.

4.8. Parametric MCI

There are three places where the MCI recursively calls itself. Applying the ideas in Section 3.3 the evaluation order can be abstracted out using a MCI which takes interpreters as parameters, and applies them in the places mentioned above. This leads to a parametric MCI with three parameters.

```
data Val = T SLTerm
         | F Fun

type Fun = Val -> Val

type Env = String -> Val

emptyEnv :: Env
```

```
emptyEnv s = undefined

addVar :: String -> Val -> Env -> Env
addVar s v e = \r -> if r == s then
                         v
                     else
                         e r

type Interp = Env -> SLTerm -> Val
interp :: Interp -> Interp -> Interp -> Interp
interp la op ar e (Var s)   = e s
interp la op ar e (Lam v b) = F (\t -> la (addVar v t e) b)
interp la op ar e (App m n) = apply (op e m) (ar e n)

apply :: Val -> Val -> Val
apply (F f) n = f n
```

We consider the fixed points:

```
intId :: Interp
intId e t = (T t)

cbv = interp cbv cbv cbv
cbn = interp cbn cbn intId
```

We must reduce the number of effective parameters only to one. The `la` is involved in β-reduction and `op` determines if the operator is interpreted (which we want always to happen). This seem limiting, but still we can define two different orders of evaluation. They do not interpret under λ-abstraction. They correspond to CBV and CBN.

This generic interpreter does not implement any semantics preserving mechanism. We give the monadic version with the `Cont` monad, which is our reference solution.

```
data Val = T SLTerm
         | F Fun

type Fun = Monad m => Val -> m Val

type Env = String -> Val

type Interp = Monad m => Env -> SLTerm -> m Val
```

```
interp :: Interp -> Interp -> Interp -> Interp
interp la op ar e (Var s)   = return $ e s
interp la op ar e (Lam v b) =
    return $ F $ \t -> la (addVar v t e) b
interp la op ar e (App m n) = do m' <- op e m
                                 n' <- ar e n
                                 apply (op e m) (ar e n)

apply :: Monad m => m Val -> m Val -> m Val
apply op ar = do m <- op
                 n <- ar
                 case m of
                   (F f) -> f n
```

The fixed points for this interpreter are:

```
mId :: Interp
mId e t = return $ T t

cbv = interp cbv cbv cbv
ha  = interp cbn cbn mId
```

We run the tests for CBV:

```
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv ident) id
<function>
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv redex) id
*** Exception: Prelude.undefined
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv capture) id
<function>
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv free) id
*** Exception: Prelude.undefined
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv varApp) id
*** Exception: Prelude.undefined
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv two) id
<function>
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv omega) id
  C-c C-cInterrupted.
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv lamOmega) id
<function>
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv constOmega) id
  C-c C-cInterrupted.
```

```
*MonadMCI3LCalcFix> (runCont $ cbv emptyEnv $ App (App y fact) three) id
  C-c C-cInterrupted.
*MonadMCI3LCalcFix> :{
*MonadMCI3LCalcFix| (runCont $ cbv emptyEnv $
*MonadMCI3LCalcFix| App (App (App z fact_d) three) (Var "_")
*MonadMCI3LCalcFix| ) id
*MonadMCI3LCalcFix| :}
<function>
*MonadMCI3LCalcFix>
```

Chapter 5

Results and conclusions

The most important novel result of this thesis has been the development of generic evaluators and interpreters for the λ-calculus. The parametric evaluators led us to the discovery of a space (we called the β-hypercube) where every evaluation order can be found. This could be the first systematic implementation of every evaluation order which is available.

We have shown how to instantiate the parametric evaluator into uniform (fixed points with single recursion) and hybrid (fixed points with mutual recursion) evaluation orders. Hybrid ones are interesting because they apply several strategies in different stages of the evaluation process. As an example, the HA evaluation order is an eager strategy, with very good performance, that always returns a normal form, but still retains the ability to work with recursion, like some weaker (in the sense of irreducibility) evaluation orders.

In the latest version of the parametric evaluator we minimised the number of parameters to four. This induces an easy-to-manipulate space where even the hybrid strategies can be defined with single recursion. This evaluator prunes the β-hypercube shadowing some evaluation orders which may seem reasonable. Nevertheless, every previously described evaluation orders fits in.

In the compositional approach we exploit mixins to isolate elementary features of the evaluators that can be mixed up resulting every uniform evaluation order in the β-hypercube. We also proposed non-standard mixin combinators that allow more intricate inheritance graphs. With this combinators is possible to define the hybrid evaluation orders as well.

We collect the results in [19], and put them into relation to the β-hypercube, revealing the peculiarities of the evaluation orders of Reynolds' interpreters, where the *on the fly* interpretation of λ-abstractions leads naturally to strategies that do stop interpreting when they found a λ-abstraction.

We also developed a generic (parametric) interpreter from it, which maps

a subset of the β-hypercube.

We reiterate in detail our work and results:

1. We have discussed and implemented in Haskell *evaluators* for the untyped λ-calculus that realise the foremost evaluation orders (AOR, NOR, CBN, and CBV among others). We started with naive (and inaccurate) versions, discussing issues of free variables and independence from Haskell's evaluation order. We then presented classic (non-monadic, eval-case) as well as monadic evaluators that really implement appropriate evaluation orders by dint of special features such as strict application, continuations, and monads. We have addressed and discussed the difficulties of overriding the implementation language's evaluation order when implementing an evaluator for a higher-order language, particularly in the context of non-strict Haskell.

2. We have presented novel generic evaluators which generalise all specific ones and are capable of implementing all evaluation orders. Genericity has been achieved by parametrisation. Generic evaluators are higher-order functions whose parameters factor-out the distinctive bits of behaviour of specific evaluators (evaluation under lambda, evaluation of parameters in application, etc). Particular evaluators have been defined as fixed points of generic ones. We have presented monadic and non-monadic versions of generic evaluators and investigate the parameter space or β-hypercube, as we like to call it. A finite definition of this space contains the salient evaluation orders. An infinite definition of a similar space is given by means of mutually recursive fixed points. It is possible to define evaluators that act on particular subexpressions only. Hybrid evaluators have been presented as combinations of the former. We have discussed parameter minimalisation and its impact on expressiveness.

3. We have presented a novel generic evaluator where genericity has been achieved by means of mixins [5]. We have defined elementary components which isolate distinctive bits of behaviour so that particular evaluation orders have been defined by composing particular components.

4. We present versions of Reynolds *interpreters* [19] for the pure λ-calculus and a generic-parametric version that abstracts out the evaluation order.

Chapter 6

Future Work

We outline the main lines of future work on this thesis:

- To develop formal proofs of correctness for evaluators and interpreters and to study the general proof for the generic versions in relation to the proofs for each specific version, which would show the benefits of genericity in action. We should also develop proofs in the context of mixins, which are not that studied.
- To extend our generic evaluators to typed and extended lambda calculus, and go into the λ-cube [2].
- To study possible changes of representation and substitution: DeBruijn indices, closures, etc.
- Reynolds' interpreters do not force evaluation under λ-abstraction. We should explore possible mechanisms to achieve this. The interpreters would be able to implement more evaluation orders than CBN and CBV. We should study also interpreters with hybrid evaluation orders and revise the number of parameters of the parametric interpreter. We should also explore the mixin road.
- The final stroke is to apply Danvy's work and study whether a generic (or "pluggable" with smaller machines) abstract machine can be obtained from generic interpreters.

Appendix A

Illustrative λ-expressions

This is the code of the λ-expressions used in this thesis as tests for the evlauators and the interpreters.

```
suc = Lam "n"
       (Lam "f"
        (Lam "x"
         (App (Var "f")
          (App (App (Var "n") (Var "f"))
           (Var "x")))))

pre = Lam "x"
       (Lam "y"
        (Lam "z"
         (App (App (App (Var "x") inter)
                       (App true (Var "z")))
          (Lam "x" (Var "x")))))
       where inter = Lam "p"
                      (Lam "q"
                       (App (Var "q")
                        (App (Var "p") (Var "y"))))

cons = Lam "x" (Lam "y" (Var "x"))

add = Lam "m" (Lam "n" (App (App (Var "n") suc) (Var "m")))
mult = Lam "m" (Lam "n"
                (Lam "f" (App (Var "m")
                          (App (Var "n") (Var "f")))))
```

```
expo = Lam "m" (Lam "n" (App (App (Var "n")
                              (App mult (Var "m"))) one))

true = Lam "p" (Lam "q" (Var "p"))
false = Lam "p" (Lam "q" (Var "q"))

cond = Lam "c"
       (Lam "t"
        (Lam "f"
         (App (App (Var "c") (Var "t")) (Var "f"))))
chNot = Lam "p" (App (App (App cond (Var "p")) false) true)
chAnd = Lam "p" (Lam "q" (App (App (App cond (Var "p"))
                               (Var "q")) false))
chOr = Lam "p" (Lam "q" (App (App (App cond (Var "p")) true)
                         (Var "q")))
isZero = Lam "n" (App (App (Var "n") (App cons false)) true)

ident = Lam "x" (Var "x")

redex = App (Lam "x" (Var "x")) (Var "x")

omega      = App twice twice
    where twice = Lam "x" (App (Var "x") (Var "x"))
lamOmega   = Lam "x" omega
constOmega = App (Lam "x" (Var "y")) omega

capture = App (Lam "x" (Lam "y" (Var "x"))) (Var "y")

free = App (App (Lam "x" (App (Var "x") (Var "y")))
            (Var "x"))
       (App  (Lam "y" (App (Var "z") (Var "y")))
        (Var "y"))

varApp = App (Var "x") (Lam "x" (Var "y"))

zero  = Lam "f" (Lam "x" (Var "x"))
one   = App suc zero
two   = App suc one
three = App suc two
four  = App suc three
five  = App suc four
```

```
y = Lam "f" (App fTwice fTwice)
    where fTwice = Lam "x"
                   (App (Var "f")
                    (App (Var "x") (Var "x")))
fact = Lam "f"
       (Lam "n"
        (App (App (App cond (App isZero (Var "n")))
                  one)
         (App (App mult (Var "n"))
          (App (Var "f") (App pre (Var "n"))))))

z = Lam "f" (App fYTwice fYTwice)
    where fYTwice = Lam "x"
                    (Lam "y"
                     (App (App (Var "f")
                           (App (Var "x") (Var "x")))
                      (Var "y")))
fact_d = Lam "f"
         (Lam "n"
          (App (App (App cond (App isZero (Var "n")))
                (Lam "d" one))
           (Lam "d" (App (App mult (Var "n"))
                     (App (App (Var "f")
                           (App pre (Var "n")))
                      (Var "d"))))))
```

Appendix B

Naive Evaluators

The following evaluators are naive in the sense that they do not implement the proper evaluation orders. Haskell's evaluation order gets in the way.

CBV Naive Evaluator

```
evalCBV :: SLTerm -> SLTerm
evalCBV v@(Var s)   = v
evalCBV l@(Lam v b) = l
evalCBV (App m n)   =
    let  m' = evalCBV m
         n' = evalCBV n
    in case m' of
           (Lam v b) -> evalCBV $ subst v b n'
           _         -> App m' n'
```

AOR Naive Evaluator

```
evalAOR :: SLTerm -> SLTerm
evalAOR v@(Var s) = v
evalAOR (Lam v b) = let b' = evalAOR b
                    in Lam v b'
evalAOR (App m n) =
    let  m' = evalAOR m
         n' = evalAOR n
    in case m' of
           (Lam v b) -> evalAOR $ subst v b n'
           _         -> App m' n'
```

CBN Naive Evaluator

```
evalCBN :: SLTerm -> SLTerm
evalCBN v@(Var s)   = v
evalCBN l@(Lam v b) = l
evalCBN (App m n)   =
    let  m’ = evalCBN m
    in case m’ of
           (Lam v b) -> evalCBN $ subst v b n
                        in App m’ n
```

HE Naive Evaluator

```
evalHE :: SLTerm -> SLTerm
evalHE v@(Var s) = v
evalHE (Lam v b) = let b’ = evalHE b
                    in $ Lam v b’
evalHE (App m n) =
    let  m’ = evalHE m
    in case m’ of
           (Lam v b) -> evalHE $ subst v b n
           _         -> App m’ n
```

NOR Naive Evaluator

```
evalNOR :: SLTerm -> SLTerm
evalNOR v@(Var s) = v
evalNOR (Lam v b) = let b' = evalNOR b
                    in Lam v b'
evalNOR (App m n) =
    let  m' = evalCBN m
    in case m' of
           (Lam v b) -> evalNOR $ subst v b n
           _         -> let m'' = evalNOR m'
                               n' = evalNOR n'
                           in App m'' n'
```

HA Naive Evaluator

```
evalHA :: SLTerm -> SLTerm
evalHA v@(Var s) = v
evalHA (Lam v b) = let b' = evalHA b
                   in Lam v b'
evalHA (App m n) =
    let  m' = evalCBV m
         n' = evalHA n
    in case m' of
           (Lam v b) -> evalHA $ subst v b n'
           _         -> App m' n'
```

HN Naive Evaluator

```
evalHN :: SLTerm -> SLTerm
evalHN v@(Var s) = v
evalHN (Lam v b) = let b' = evalHN b
                   in Lam v b'
evalHN (App m n) =
    let  m' = evalHE m
    in case m' of
           (Lam v b) -> evalHN $ subst v b n
           _         -> let m'' = evalHN m'
                               n' = evalHN n'
                           in App m'' n'
```

Appendix C

Evaluators Using Pattern Matching

The following evaluators use pattern matching to force the *evaluation* of the expressions at certain points, thus preserving the intended semantics for the defined language.

CBV Evaluator with Pattern Matching

```
evalCBV :: SLTerm -> SLTerm
evalCBV v@(Var s)   = v
evalCBV l@(Lam v b) = l
evalCBV (App m n)   =
    let m’ = evalCBV m
        n’ = evalCBV n
    in case m’ of
           (Lam v b) -> case n’ of
                          (Var _) -> evalCBV $ subst v b n’
                          _       -> evalCBV $ subst v b n’
           _         -> case n’ of
                          (Var _) -> App m’ n’
                          _       -> App m’ n’
```

AOR Evaluator with Pattern Matching

```
evalAOR :: SLTerm -> SLTerm
evalAOR v@(Var s) = v
evalAOR (Lam v b) = let b' = evalAOR b
                    in case b' of
                        (Var _) -> Lam v b'
                        _       -> Lam v b'
evalAOR (App m n) =
    let m' = evalAOR m
        n' = evalAOR n
    in case m' of
           (Lam v b) -> case n' of
                        (Var _) -> evalAOR $ subst v b n'
                        _       -> evalAOR $ subst v b n'
           _         -> case n' of
                        (Var _) -> App m' n'
                        _       -> App m' n'
```

CBN Evaluator with Pattern Matching

```
evalCBN :: SLTerm -> SLTerm
evalCBN v@(Var s  ) = v
evalCBN l@(Lam v b) = l
evalCBN (App m n)   =
    let m' = evalCBN m
    in case m' of
           (Lam v b) -> evalCBN $ subst v b n
           _         -> App m' n
```

HE Evaluator with Pattern Matching

```
evalHE :: SLTerm -> SLTerm
evalHE v@(Var s  ) = v
evalHE l@(Lam v b) = let b’ = evalHE b
                     in Lam v $! b
evalHE (App m n)   =
    let m’ = evalHE m
    in case m’ of
           (Lam v b) -> evalHE $ subst v b n
           _         -> App m’ n
```

NOR Evaluator with Pattern Matching

```
evalNOR :: SLTerm -> SLTerm
evalNOR v@(Var s) = v
evalNOR (Lam v b) = let b’ = evalNOR b
                    in case b’ of
                        (Var _) -> Lam v b’
                        _       -> Lam v b’
evalNOR (App m n) =
    let m’ = evalCBN m
    in case m’ of
           (Lam v b) -> evalNOR $ subst v b n
           _         -> let n’ = evalNOR n
                        in case n’ of
                               (Var _) -> App m’ n’
                               _       -> App m’ n’
```

HA Evaluator with Pattern Matching

```
evalHA :: SLTerm -> SLTerm
evalHA v@(Var s) = v
evalHA (Lam v b) = let b' = evalHA b
                   in case b' of
                        (Var _) -> Lam v b'
                        _       -> Lam v b'
evalHA (App m n) =
    let m' = evalCBV m
        n' = evalHA n
    in case m' of
           (Lam v b) -> case n' of
                        (Var _) -> evalHA $ subst v b n'
                        _       -> evalHA $ subst v b n'
           _         -> case n' of
                        (Var _) -> App m' n'
                        _       -> App m' n'
```

HN Evaluator with Pattern Matching

```
evalHN :: SLTerm -> SLTerm
evalHN v@(Var s) = v
evalHN (Lam v b) = let b' = evalHN b
                   in case b' of
                        (Var _) -> Lam v b'
                        _       -> Lam v b'
evalHN (App m n) =
    let m' = evalHE m
    in case m' of
           (Lam v b) -> evalHN $ subst v b n
           _         -> let n' = evalHN n
                        in case n' of
                               (Var _) -> App m' n'
                               _       -> App m' n'
```

Appendix D

Evaluators using the Strict Application Operator ($!)

The following evaluators use the strict application operator ($!)to force the evaluation of the expressions at certain points, thus preserving the intended semantics for the defined language.

CBV Evaluator using ($!)

```
evalCBV :: SLTerm -> SLTerm
evalCBV v@(Var s)   = v
evalCBV l@(Lam v b) = l
evalCBV (App m n)   =
    let  m' = evalCBV m
         n' = evalCBV n
    in case m' of
           (Lam v b) -> evalCBV $ subst v b $! n'
           _         -> App m' $! n'
```

AOR Evaluator using ($!)

```
evalAOR :: SLTerm -> SLTerm
evalAOR v@(Var s) = v
evalAOR (Lam v b) = let b' = evalAOR b
                    in Lam v $! b'
evalAOR (App m n) =
    let  m' = evalAOR m
         n' = evalAOR n
    in case m' of
           (Lam v b) -> evalAOR $ subst v b $! n'
           _         -> App m' $! n'
```

CBN Evaluator using ($!)

```
evalCBN :: SLTerm -> SLTerm
evalCBN v@(Var s)   = v
evalCBN l@(Lam v b) = l
evalCBN (App m n)   =
    let  m' = evalCBN m
    in case m' of
           (Lam v b) -> evalCBN $ subst v b n
           _         -> App m' n
```

HE Evaluator using ($!)

```
evalHE :: SLTerm -> SLTerm
evalHE v@(Var s) = v
evalHE (Lam v b) = let b' = evalHE b
                   in Lam v $! b'
evalHE (App m n) =
    let  m' = evalHE m
    in case m' of
           (Lam v b) -> evalHE $ subst v b n
           _         -> App m' n
```

NOR Evaluator using ($!)

```
evalNOR :: SLTerm -> SLTerm
evalNOR v@(Var s) = v
evalNOR (Lam v b) = let b' = evalNOR b
                    in Lam v $! b'
evalNOR (App m n) =
    let  m' = evalCBN m
    in case m' of
           (Lam v b) -> evalNOR $ subst v b n
           _         -> let n' = evalNOR n
                           in App m' $! n'
```

HA Evaluator using ($!)

```
evalHA :: SLTerm -> SLTerm
evalHA v@(Var s) = v
evalHA (Lam v b) = let b' = evalHA b
                   in Lam v $! b'
evalHA (App m n) =
    let  m' = evalCBV m
         n' = evalHA n
    in case m' of
           (Lam v b) -> evalHA $ subst v b $! n'
           _         -> App m' $! n'
```

HN Evaluator using ($!)

```
evalHN :: SLTerm -> SLTerm
evalHN v@(Var s) = v
evalHN (Lam v b) = let b' = evalHN b
                   in Lam v $! b'
evalHN (App m n) =
    let  m' = evalHE m
    in case m' of
           (Lam v b) -> evalHN $ subst v b n
           _         -> let n' = evalHN n
                           in App m' $! n'
```

Appendix E

Evaluators using the Strict Sequencing Operator (seq)

The following evaluators use the strict sequencing operator seq to force the evaluation of the expressions at certain points, thus preserving the intended semantics for the defined language.

CBV Evaluator using seq

```
evalCBV :: SLTerm -> SLTerm
evalCBV v@(Var s)   = v
evalCBV l@(Lam v b) = l
evalCBV (App m n)   =
    let  m’ = evalCBV m
         n’ = evalCBV n
    in case m’ of
           (Lam v b) -> n’ ‘seq‘ (evalCBV $ subst v b n’)
           _         -> n’ ‘seq‘ (App m’ n’)
```

AOR Evaluator using seq

```
evalAOR :: SLTerm -> SLTerm
evalAOR v@(Var s) = v
evalAOR (Lam v b) = let b’ = evalAOR b
                    in b’ ‘seq‘ (Lam v b’)
evalAOR (App m n) =
    let  m’ = evalAOR m
         n’ = evalAOR n
    in case m’ of
           (Lam v b) -> n’ ‘seq‘ (evalAOR $ subst v b n’)
           _         -> n’ ‘seq‘ (App m’ n’)
```

CBN Evaluator using seq

```
evalCBN :: SLTerm -> SLTerm
evalCBN v@(Var s)   = v
evalCBN l@(Lam v b) = l
evalCBN (App m n)   =
    let  m’ = evalCBN m
    in case m’ of
           (Lam v b) -> evalCBN $ subst v b n
           _         -> let n’ = evalCBN n
                        in n’ ‘seq‘ (App m’ n’)
```

HE Evaluator using seq

```
evalHE :: SLTerm -> SLTerm
evalHE v@(Var s)   = v
evalHE l@(Lam v b) = l
evalHE (App m n)   =
    let  m' = evalHE m
    in case m' of
           (Lam v b) -> evalHE $ subst v b n
           _         -> let n' = evalHE n
                           in n' `seq` (App m' n')
```

NOR Evaluator using seq

```
evalNOR :: SLTerm -> SLTerm
evalNOR v@(Var s) = v
evalNOR (Lam v b) = let b' = evalNOR b
                    in b' `seq` (Lam v b')
evalNOR (App m n) =
    let  m' = evalCBN m
    in case m' of
           (Lam v b) -> evalNOR $ subst v b n
           _         -> let n' = evalNOR n
                           in n' `seq` (App m' n')
```

HA Evaluator using seq

```
evalHA :: SLTerm -> SLTerm
evalHA v@(Var s) = v
evalHA (Lam v b) = let b' = evalHA b
                   in b' `seq` (Lam v b')
evalHA (App m n) =
    let  m' = evalCBV m
         n' = evalHA n
    in case m' of
           (Lam v b) -> n' `seq` (evalHA $ subst v b n')
           _         -> n' `seq` (App m' n')
```

HN Evaluator using seq

```
evalHN :: SLTerm -> SLTerm
evalHN v@(Var s) = v
evalHN (Lam v b) = let b' = evalHN b
                   in b' `seq` (Lam v b')
evalHN (App m n) =
    let  m' = evalHE m
    in case m' of
           (Lam v b) -> evalHN $ subst v b n
           _         -> let n' = evalHN n
                           in n' `seq` (App m' n')
```

Appendix F

Evaluators using CPS

The following evaluators use CPS to force the evaluation of the expressions at certain points, thus preserving the intended semantics for the defined language.

CBV Evaluator using CPS

```
evalCBV :: SLTerm -> (SLTerm -> r) -> r
evalCBV v@(Var s)   c = c v
evalCBV l@(Lam v b) c = c l
evalCBV (App m n)   c =
    evalCBV m
    (\m' -> evalCBV n
            (\n' -> case m' of
                    (Lam v b) -> evalCBV (subst v b n') c
                    _         -> c $ App m' n'))
```

AOR Evaluator using CPS

```
evalAOR :: SLTerm -> (SLTerm -> r) -> r
evalAOR v@(Var s) c = c v
evalAOR (Lam v b) c = evalAOR b
                       (\b' -> c $ Lam v b')
evalAOR (App m n) c =
    evalAOR m
    (\m' -> evalAOR n
            (\n' -> case m' of
                    (Lam v b) -> evalAOR (subst v b n') c
                    _         -> c $ App m' n'))
```

CBN Evaluator using CPS

```
evalCBN :: SLTerm -> (SLTerm -> r) -> r
evalCBN v@(Var s)   c = c v
evalCBN l@(Lam v b) c = c l
evalCBN (App m n)   c =
    evalCBN m
    (\m' -> case m' of
            (Lam v b) -> evalCBN (subst v b n) c
            _         -> evalCBN n
                            (\n' -> c $ App m' n'))
```

HE Evaluator using CPS

```
evalHE :: SLTerm -> (SLTerm -> r) -> r
evalHE v@(Var s)   c = c v
evalHE l@(Lam v b) c = c l
evalHE (App m n)   c =
    evalHE m
    (\m' -> case m' of
            (Lam v b) -> evalHE (subst v b n) c
            _         -> evalHE n
                            (\n' -> c $ App m' n'))
```

NOR Evaluator using CPS

```
evalNOR :: SLTerm -> (SLTerm -> r) -> r
evalNOR v@(Var s)   c = c v
evalNOR l@(Lam v b) c = evalNOR b
                        (\b' -> c $ Lam v b')
evalNOR (App m n)   c =
    evalCBN m
    (\m' -> case m' of
            (Lam v b) -> evalNOR (subst v b n) c
            _         -> evalNOR n
                            (\n' -> c $ App m' n'))
```

HA Evaluator using CPS

```
evalHA :: SLTerm -> (SLTerm -> r) -> r
evalHA v@(Var s) c = c v
evalHA (Lam v b) c = evalHA b
                     (\b' -> c $ Lam v b')
evalHA (App m n) c =
    evalCBV m
    (\m' -> evalHA n
            (\n' -> case m' of
                    (Lam v b) -> evalHA (subst v b n') c
                    _         -> c $ App m' n'))
```

HN Evaluator using CPS

```
evalHN :: SLTerm -> (SLTerm -> r) -> r
evalHN v@(Var s)   c = c v
evalHN l@(Lam v b) c = evalHN b
                        (\b' -> c $ Lam v b')
evalHN (App m n)   c =
    evalHE m
    (\m' -> case m' of
            (Lam v b) -> evalHN (subst v b n) c
            _         -> evalHN n
                           (\n' -> c $ App m' n'))
```

Appendix G

Monadic Evaluators

The following evaluators are implemented using monadic style. They will preserve the intended semantics when instantiated with some particular monads.

CBV Monadic Evaluator

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

AOR Monadic Evaluator

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

CBN Monadic Evaluator

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

HE Monadioc Evaluator

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

NOR Monadic Evaluator

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

HA Monadic Evaluator

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

HN Monadic Evaluator

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

Appendix H

Parametric Evaluators

The following are parametric evaluators, for them different fix points can be written, which implement different reduction orders. We give the evaluators according to every strategy in Section 2.8.

Naive Parametric Evaluator

```
type Ev = SLTerm -> SLTerm
eval :: Ev -> Ev -> Ev ->
        Ev -> Ev -> Ev -> Ev
eval la op1 ar1 su op2 ar2 v@(Var s) = v
eval la op1 ar1 su op2 ar2 (Lam v b) =
    let b' = la b
    in Lam v b'
eval la op1 ar1 su op2 ar2 (App m n) =
    let  m' = op1 m
    in case m' of
           (Lam v b) -> let n' = ar1 n
                        in su $ subst v b n'
           _         -> let m'' = op2 m
                            n'' = ar2 n
                        in App m'' n''
```

Pattern Parametric Evaluator

```
type Ev = SLTerm -> SLTerm
eval :: Ev -> Ev -> Ev ->
        Ev -> Ev -> Ev -> Ev
eval la op1 ar1 su op2 ar2 v@(Var s) = v
eval la op1 ar1 su op2 ar2 (Lam v b) =
                                       let b' = la b
                   in case b' of
                        (Var _) -> Lam v b'
                        _       -> Lam v b'
eval la op1 ar1 su op2 ar2 (App m n) =
    let m' = op1 m
    in case m' of
           (Lam v b) -> let n' = ar1 n
                        in case n' of
                               (Var _) -> su $ subst v b n'
                               _       -> su $ subst v b n'
           _         -> let m'' = op2 m
                            n'' = ar2 n
                        in case n'' of
                               (Var _) -> App m'' n''
                               _       -> App m'' n''
```

Strict Application Operator Parametric Evaluator

```
type Ev = SLTerm -> SLTerm
eval :: Ev -> Ev -> Ev ->
        Ev -> Ev -> Ev -> Ev
eval la op1 ar1 su op2 ar2 v@(Var s) = v
eval la op1 ar1 su op2 ar2 (Lam v b) =
    let b' = la b
    in Lam v $! b'
eval la op1 ar1 su op2 ar2 (App m n) =
    let  m' = op1 m
    in case m' of
           (Lam v b) -> let n' = ar1 n
                        in su $ subst v b $! n'
           _         -> let m'' = op2 m
                            n'' = ar2 n
                        in (\mx -> App mx $! n'') $! m''
```

Strict Sequencing Operator Parametric Evaluator

```
type Ev = SLTerm -> SLTerm
eval :: Ev -> Ev -> Ev ->
        Ev -> Ev -> Ev -> Ev
eval la op1 ar1 su op2 ar2 v@(Var s) = v
eval la op1 ar1 su op2 ar2 (Lam v b) =
    let b' = la b
    in b' `seq` (Lam v b')
eval la op1 ar1 su op2 ar2 (App m n) =
    let  m' = op1 m
    in case m' of
           (Lam v b) -> let n' = ar1 n
                        in n' `seq` (su $ subst v b n')
           _         -> let m'' = op2 m
                            n'' = ar2 n
                        in m'' `seq` (n'' `seq` (App m'' n''))
```

CPS Parametric Evaluator

```
type Ev = SLTerm -> (SLTerm -> SLTerm) -> SLTerm
eval :: Ev -> Ev -> Ev -> Ev -> Ev -> Ev -> Ev
eval la op1 ar1 su op2 ar2 v@(Var s) k = k v
eval la op1 ar1 su op2 ar2 (Lam v b) k = la b
                             (\b' -> k $ Lam v b')
eval la op1 ar1 su op2 ar2 (App m n) k =
    op1 m
    (\m' -> case m' of
             (Lam v b) -> ar1 n
                          ( \n' -> su (subst v b n') k)
             _         -> op2 m
                          (\m'' -> ar2 n
                           (\n'' -> k $ App m'' n'')))
```

Monadic Parametric Evaluator

This evaluator may be used with the `Cont` monad..

```
type Ev m = SLTerm -> m SLTerm
eval :: Monad m => Ev m -> Ev m -> Ev m ->
        Ev m -> Ev m -> Ev m -> Ev m
eval la op1 ar1 su op2 ar2 v@(Var s) = return v
eval la op1 ar1 su op2 ar2 (Lam v b) =
    do b' <- la b
       return $ Lam v b'
eval la op1 ar1 su op2 ar2 (App m n) =
    do m' <- op1 m
       case m' of
                (Lam v b) -> do n' <- ar1 n
                                su $ subst v b n'
                _         -> do m'' <- op2 m
                                n'' <- ar2 n
                                return $ App m'' n''
```

Fix Points for the Four First Parametric Evaluators

```
aor  = eval aor aor aor aor aor aor
cbv  = eval id  cbv cbv cbv cbv cbv
cbn  = eval id  cbn id  cbn cbn id
he   = eval he  he  id  he  he  id
```

Fix Points for the CPS Parametric Evaluator

```
cId :: a -> (a -> a) -> a
cId a = ($ a)
aor  = eval aor aor aor aor aor aor
cbv  = eval cId cbv cbv cbv cbv cbv
cbn  = eval cId cbn cId cbn cbn cId
he   = eval he  he  cId he  he  cId
```

Fix Points for the Monadic Parametric Evaluator

```
aor  = eval aor    aor    aor    aor    aor    aor
cbv  = eval return cbv    cbv    cbv    cbv    cbv
cbn  = eval return cbn    return cbn    return return
he   = eval he     he     return he     return return
```

Bibliography

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 8–19. ACM Press, 2003.

[2] Henk P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North Holland, 1984.

[3] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[4] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[5] W. R. Cook. *A denotational semantics of inheritance*. PhD thesis, Providence, RI, USA, 1989.

[6] Bruno C. d. S. Oliveira. The different aspects of monads and mixins. 2007.

[7] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Principles of Programming Languages*, pages 458–471, 1994.

[8] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–55. ACM, 2007.

[9] P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1963.

[10] Peter J. Landin. A λ-calculus approach. In Leslie Fox, editor, *Advances in Programming and Non-Numerical Computation*, pages 97–141, Oxford, 1966. Oxford University Computing Laboratory and Delegacy for Extra-Mural Studies, Pergammon Press.

[11] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Comm. Assoc. Comput. Mach.*, 3(3):184–195, 1960.

[12] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[13] J. Newbern and A. Palamarchuk. Module control.monad.cont. Haskell-Wiki, 2007.

[14] Simon Peyton Jones, editor. *The Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, Cambridge, England, 2003.

[15] Simon Peyton Jones and Satnam Singh. Multi-core is here: Can you program for it? *Dr Dobb's Journal*, December 30 2008.

[16] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

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

[18] J. C. Reynolds. GEDANKEN - A simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13:308–319, 1970.

[19] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[20] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Report PRG-6, Oxford University Computer Laboratory, 1971.

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

[22] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[23] P. Wadler. The essence of functional programming. In *Proc. POPL'92*, 1992.