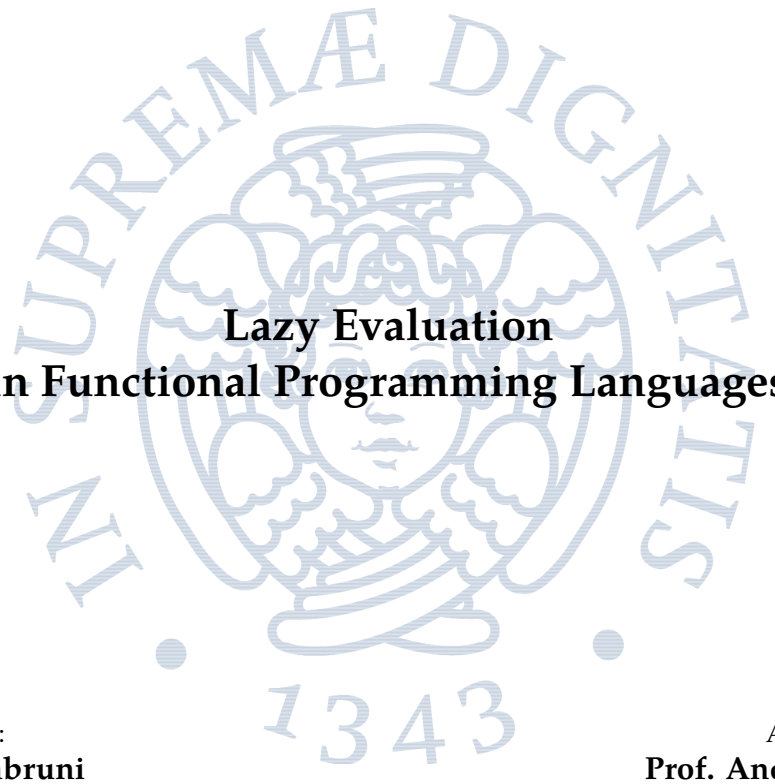
UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA
Corso di Laurea in Informatica

ELABORATO FINALE, CURRICULUM METODOLOGICO



Lazy Evaluation in Functional Programming Languages

Candidate:
Orlando Leombruni

Advisor:
Prof. Andrea Corradini

Anno Accademico 2014/2015

Contents

Ringraziamenti

Sono passati più di 7 anni dalla prima volta in cui ho messo piede nei corridoi di un'università: era un altro ateneo (L'Aquila), un corso di studi totalmente diverso (Medicina e Chirurgia) e soprattutto avevo qualche capello bianco in meno (comunque troppi per un diciannovenne!). In questi anni è successo un po' di tutto, dalla terribile notte del 6 Aprile 2009 ad un "reboot" abbastanza fallimentare come studente di Medicina nell'ateneo di Chieti, fino ad approdare – finalmente! – sul sentiero a me più congruo: quello dell'Informatica, mia vera passione sin dall'infanzia.

Mi piace pensare a questo elaborato come coronamento dell'*intero* percorso da me affrontato, piuttosto che come "semplice" prova finale del corso di laurea in Informatica. Impossibile quindi non ringraziare coloro che mi sono sempre stati vicini, che hanno supportato tutte le mie scelte, aiutato e sorretto nei momenti di difficoltà e gioito con me in quelli di felicità: coloro che, in pratica, hanno permesso tutto questo. Un enorme, gigantesco «grazie» di cuore alla mia bella mamma *rocker* Mara, a Maurizio, a nonna Maria ed a zio Valerio.

A Pisa ho incontrato molte persone, la maggior parte delle quali davvero splendide; non posso però fare a meno di ringraziare in modo particolare la più splendida di tutte, una collega seria e giudiziosa, un'ottima coinquilina, colei che ormai da quattro anni mi sopporta quotidianamente: questo traguardo è anche merito tuo, Alessandra!

Un sentito ringraziamento va anche al Prof. Andrea Corradini, per la pazienza e la cortesia dimostratami non solo durante la stesura di questo lavoro ma anche nel corso degli studi.

Ho un solo rimpianto: non essere riuscito a concludere gli studi in tempo affinché potessi festeggiare questo traguardo insieme al mio caro nonno Vincenzo. Vorrei rimediare, seppur in minima parte, dedicando a lui questo lavoro. Ciao nonno, mi manchi.

Orlando

1 | Introduction

This work concerns the implementation of a simple functional language using the rules and techniques of *lazy evaluation*. Lazy evaluation is an evaluation strategy based on two principles:

- the evaluation of any expression is delayed until its value is needed (*non-strict evaluation*);
- any expression will be evaluated *at most* once (*sharing*).

The lazy evaluation strategy works very well in a purely functional context: both of its principles can be implemented without consequences if there are no side effects to functions. Conversely, imperative languages are better suited for *eager evaluation*, which is the polar opposite: expressions are evaluated as soon as they are used and there's no sharing, since these kind of languages require a tighter control on the order and side effects of operations.

1.1 The case for lazy evaluation

Lazy evaluation seems a good practice in theory, but what advantages does it bring to the programmer? The immediate answer is that lazy evaluation enables the handling of *infinite data structures and streams*: one can build the (infinite) list of all the Fibonacci numbers with a simple recursive function and then select the *i*-th, and a lazy language will compute only the necessary steps (and only when necessary). Living in an era where fast analysis of enormous quantities of information is crucial (the so-called *big data analysis*), lazy languages can help obtain partial results faster since they can output computed values before the entire operation has finished. Other benefits may include a performance increase (due to avoiding needless computations) and a smaller memory footprint (since values are only created when needed).

To make a real-world example, Facebook's Sigma anti-malware engine core is written in Haskell (a lazy functional language) and serves over one

million requests per second[1], identifying and blocking malicious actions before they affect other people on the social network.

1.2 The case *against* lazy evaluation

Lazy evaluation comes at a price: execution speed. There seems to be no avoiding this; creating, mantaining and working on the data structures required for lazy evaluation introduce a big overhead, which may prevail on the performance increases explained above; moreover, a careless manipulation of infinite data structures can lead to serious space leaks. Some languages tried to reach a good middle-ground by having eager evaluation as default, whilst having constructs that the programmer can use to explicitly request lazy evaluation.

1.3 Introducing Haskell

Haskell is a strongly-typed, purely functional programming language with non-strict semantics. Designed in 1990 (and revised many times) by a committee to be the "open standard" for functional languages ([S. Peyton Jones, 1992]), it follows the steps of Miranda and other proprietary, non-strict functional languages that were popular at the end of the 1980s. The standard implementation of Haskell is called *GHC* (Glasgow Haskell Compiler), noted for its high-performance concurrency and parallelism mechanisms [S. Peyton Jones, 2007].

Being a purely functional language, functions written in Haskell do not have side effects; these are handled with a special construct, called *monad*. Thanks to monads, Haskell can support different kinds of computation like nondeterminism, error handling and parsing.

For our purposes, we'll use a very small subset of Haskell's features: we won't talk about type-checking, monads, list comprehensions and many other things. The aim is to show how a functional language can be implemented, and which techniques to use for non-strict semantics.

1.4 Structure of this work

The first step towards our goal is to translate the Haskell code into an intermediate language, called λ-calculus, which will be introduced in chapter 2 and refined in chapter 3, where we will also show how to make the translation. Chapter 4 details how to mantain and manipulate expressions of the λ-calculus in memory through a technique called *graph reduction*. Finally, in chapter 5 we will discuss about a faster implementation of graph reduction through the use of *supercombinators*.

[1] `https://archive.is/NUsFz`

2 | The λ-calculus

2.1 Overview

In this chapter, we briefly introduce a formal system named *λ-calculus*. Lambda calculus was first formulated by Alonzo Church ([Church, 1941]) and is nowadays used as a basis for functional programming languages; it's a simple language, with few syntactic constructs, but it's sufficiently powerful to express all computable functions ([Turing, 1936]). The seminal work on λ-calculus is [H. Barendregt, 1984].

2.2 A formal description of λ-calculus

We start by introducing a few useful definitions.

Definition 2.1 (Alphabet of the λ-calculus)**.** The *alphabet* of the λ-calculus consists of:

- a countably infinite **set of variables** x_0, x_1, ...;
- an **abstractor** λ;
- the two **parentheses symbols** (and). ■

Definition 2.2 (Set of λ-terms)**.** The *set of λ-terms* (or *terms of the λ-calculus*) $Ter(\lambda)$ is defined inductively as follows:

1. a variable is a term, so $x_0, x_1, \ldots \in Ter(\lambda)$;
2. if $M, N \in Ter(\lambda)$, then $(MN) \in Ter(\lambda)$;
3. if $M \in Ter(\lambda)$ and x is a variable, then $(\lambda x\,.\,M) \in Ter(\lambda)$.

If we enrich the alphabet with a set of constants Γ (with $\alpha, \beta, \ldots \in \Gamma$), we can consider λ-terms over Γ. In that case, we change $Ter(\lambda)$ into $Ter(\lambda\Gamma)$ in the above clauses and add a fourth rule:

4. a constant is a term, so $\alpha, \beta, \ldots \in Ter(\lambda\Gamma)$ (or, more concisely, $\Gamma \subseteq Ter(\lambda\Gamma)$). ■

The λ-calculus essentially gives a formalism to express anonymous (non-named) functions; the λ-term $(\lambda x\,.\,x)$ is short for "the function which takes a parameter x and returns x itself". In the remainder of the chapter, w use lowercase Latin letters for variables $(x, y, z, a, b...)$, lowercase Greek letters for constants $(\alpha, \beta, \gamma...)$ and uppercase Latin letters for arbitrary λ-terms $(M, N, L, P, Q, ...)$. Moreover, we call *application terms* all terms of the form (MN), and *abstraction terms* all terms of the form $(\lambda x\,.\,M)$. The application is a left-associative operation, i.e. $M\,N\,P\,Q = ((M\,N)\,P)\,Q$.

Definition 2.3 (Free variables)**.** Let M be a λ-term. The *set of free variables occurring in M*, $FV(M)$ is defined inductively as follows:

1. for each variable x, $FV(x) = \{\, x \,\}$;
2. $FV(MN) = FV(M) \cup FV(N)$;
3. $FV(\lambda x\,.\,M) = FV(M) - \{\, x \,\}$.

We say that a variable x *occurs free in* M if $x \in FV(M)$. Terms without free variables are called *closed* terms or *combinators*; $Ter_0(\lambda)$ is the set of closed terms. ■

Definition 2.4 (Subterms)**.** Let M be a λ-term. The *set of subterms of M*, $Sub(M)$ is defined inductively as follows:

1. for each variable x, $Sub(x) = \{\, x \,\}$;
2. $Sub(MN) = Sub(M) \cup Sub(N) \cup \{\, MN \,\}$;
3. $Sub(\lambda x.M) = Sub(M) \cup \{\, x \,\} \cup \{\, \lambda x.M \,\}$.

A term N *occurs in* M, $N \subseteq M$, if $N \in Sub(M)$. A variable x *occurs bound in* M if $(\lambda x\,.\,N) \subseteq M$ for some term N. ■

Note that the terms "occurs free" and "occurs bound" refer to *specific occurrences* of the variable in the λ-term: a variable can have both free and bound occurrences in a λ-expression. For example, in $(\lambda y\,.\,x\,y\,(\lambda x\,.\,x\,y\,z))$ the variable x occurs twice: free in $\lambda y\,.\,x\,y\,(\ldots)$ and bound in $\lambda x\,.\,x\,y\,z$ (for reference, y only appears bound and z only appears free).

Definition 2.5 (Substitution)**.** Let M, N be λ-terms and x be a variable such that $x \in FV(M)$ (i.e. x occurs free at least once in M). The *substitution* of N for the free occurrences of x in M, $M[N/x]$, is defined inductively as follows:

1. $x[N/x] \equiv N$;
2. $y[N/x] \equiv y$, for $y \not\equiv x$;
3. $(M_1 M_2)[N/x] \equiv (M_1[N/x])(M_2[N/x])$;
4. $(\lambda y \,.\, M)[N/x] \equiv \lambda y \,.\, (M[N/x])$.

NOTE. $M \equiv N$ denotes the fact that M and N are identical terms. ■

A substitution defined in this way can "break" the λ-expression, changing its semantics: for example, $(\lambda y \,.\, y\, x)[y/x]$ is a perfectly legal substitution – at least according to our definitions – but in the result $(\lambda y \,.\, y\, y)$ the free variable y (the one in $[y/x]$) is now bound by the λ-abstraction. Viceversa, in $(\lambda y \,.\, y)[x/y] \equiv \lambda y \,.\, x$ the substitution "unbinds" the variable in the body of the λ-abstraction. To avoid such situations, we will assume that when writing $M[N/x]$

- no free variables in N become bound after substitution in M, and
- the variable to be substituted x does not occur bound in M.

Both of those assumptions are reasonable; we can always find a suitable renaming of the bound variables in M to achieve the same effect, as we'll show in section 2.3.3.

2.2.1 *Pure* and *applied* λ-calculus

So far, we've introduced λ-calculus in its purest form – there are only variables, application terms and abstraction terms, and maybe constants; this is indeed what Church envisioned in his works. There is no need of the concept of *number* in the pure λ-calculus; the fundamental "block" is the function, and with functions we can represent anything – even numbers. One way to express them in the pure λ-calculus is with **Church numerals**, a system in which numbers are represented with specific λ-abstractions:

- 0 is $\lambda s \,.\, \lambda z \,.\, z$ (the argument s is applied to the argument z 0 times)
- 1 is $\lambda s \,.\, \lambda z \,.\, s\, z$ (s is applied to z 1 time)
- 2 is $\lambda s \,.\, \lambda z \,.\, s\,(s\, z)$ (s is applied to z 2 times)

and so on. It may be quite the verbose way to express numbers, but it works. From there, we can (somewhat) easily build a successor function, which takes a Church numeral and returns the succeeding numeral:
$\lambda m \,.\, \lambda s \,.\, \lambda z \,.\, s\,(m\, s\, z)$.

Example 2.1. The natural number 2 is represented by the Church numeral $\lambda s\,.\,\lambda z\,.\,s\,(s\,z)$; we apply the above function to obtain its successor:

$$\begin{aligned}
&(\lambda m\,.\,\lambda s\,.\,\lambda z\,.\,s\,(m\,s\,z))(\lambda s\,.\,\lambda z\,.\,s\,(s\,z)) \\
\Longrightarrow\ &\lambda s\,.\,\lambda z\,.\,s\,((\lambda s\,.\,\lambda z\,.\,s\,(s\,z))\,s\,z) \\
\Longrightarrow\ &\lambda s\,.\,\lambda z\,.\,s\,((\lambda z\,.\,s\,(s\,z))\,z) \\
\Longrightarrow\ &\lambda s\,.\,\lambda z\,.\,s\,(s\,(s\,z)).
\end{aligned}$$

The result $\lambda s\,.\,\lambda z\,.\,s\,(s\,(s\,z))$ is the Church numeral representing 3, which is the successor of 2; therefore our successor function is correct.
NOTE. In this example we implicitly used a simplifying rule (β-reduction) that we'll formally define in the next section. ▲

It's not hard to see that the usual arithmetic functions (sum, product…) are obtainable through these simple building blocks. For our purpose, however, the Church numerals notation is very inconvenient; in the remainder of this work we will use a richer version of λ-calculus, named the *applied* λ-calculus, which includes natural numbers as constants and some built-in functions like $+$ (in prefix notation) and even IF. To further lighten the notation, we'll name functions and reuse the names inside terms; so, for example, we can (and will) write λ-terms like

$$\begin{aligned}
\text{add2} &:= \lambda x\,.\,(+\,2\,x), \\
\text{add4} &:= \lambda x\,.\,(\text{add2}\,(\text{add2}\,x)) \\
&\equiv \lambda x\,.\,(+\,2\,(+\,2\,x)).
\end{aligned}$$

2.3 Operational Semantics of the λ-calculus

Now that we've defined what the λ-calculus *is*, we shall direct our attention on *how it works*. In order to obtain some concrete result from a λ-expression we need to *evaluate* it, a process with repeatedly selects a reducible subexpression (a *redex*) and applies some kinds of transformations to it. In this section we show how these transformations work.

2.3.1 Function Application

In the λ-calculus the function application is denoted by an application term, so $(a\,b)$ means "*the function a applied to the argument b*". What about having a function with several arguments, like $(+\,a\,b)$? Instead of using a different notation, say $(+(a,b))$, we can think of applying the function "in steps", one argument at a time: first we apply $+$ to a, obtaining as a result *a new function* that takes one argument and adds a to it, then we apply this intermediate function to b. (It's perfectly normal for a function of

the λ-calculus to return a function as its result.) This way of thinking of all functions as having only one argument was introduced by Schonfinkel [Schönfinkel, 1924], and popularized by Curry [Curry and Feys, 1958], and it's known as *currying*.

2.3.2 β-conversion

We said that λ-abstractions denote functions; we can interpret application terms of the form $(\lambda x\,.\,M)\,N$, called β-redexes for reasons that will become clear shortly, as the application of a certain function denoted by $(\lambda x\,.\,M)$ to the argument N. We can informally say that the result of such application is a λ-term whose body is an instance of M, in which all free occurrences of the formal parameter x are replaced with (copies of) the argument N. More formally, we can introduce a relation between two elements of $Ter(\lambda)$ which helps us evaluate β-redexes.

Definition 2.6 (β-reduction)**.** The relation of *β-reduction*, $\rightarrow_\beta \subseteq Ter(\lambda) \times Ter(\lambda)$, is defined by

$$\rightarrow_\beta = \{\ ((\lambda x\,.\,M)\,N,\ M[N/x])\ |\ M,\ N \in Ter(\lambda)\ \}\,. \qquad \blacksquare$$

Thanks to β-reduction, we can simplify λ-abstractions when they are applied to (at least some) arguments; $(\lambda x\,.\,+\,3\,x)\,5$ becomes $(+\,3\,5)$ and we can use the built-in definition of $+$ to obtain the final result, 8. Moreover, since we require that the formal parameter be a free variable in the abstraction body (see definition 2.5), we avoid troublesome situations that arise when formal parameter names are not unique, as shown in this example.

Example 2.2. We want to reduce a λ-term defined as follows:

$$(\lambda x\,.\,(\lambda x\,.\,+\,(-\,x\,2))\,5\,x)\,7.$$

Applying our definitions of β-reduction and substitution, we substitute 7 in all of the *free* occurrences of x in the abstraction body $(\lambda x\,.\,+\,(-\,x\,2))\,5\,\underline{x}$ (i.e. the underlined $\underline{x}$) and obtain

$$\begin{aligned} & (\lambda x\,.\,+\,(-\,x\,2))\,5\,\underline{7} \\ \rightarrow_\beta\ & +\,(-\,5\,2)\,7 \\ =\ & +\,3\,7 \\ =\ & 10. \end{aligned}$$

Without the free occurrence requirement in the definition of substitution, we could have gotten a simplification like

$$\begin{aligned} & (\lambda x\,.\,+\,(-\,\underline{7}\,2))\,5\,\underline{7} \\ \rightarrow_\beta\ & +\,(-\,7\,2)\,7 \\ =\ & +\,5\,7 \\ =\ & 12 \end{aligned}$$

which is obviously a wrong reduction. ▲

In some cases, it's useful to *introduce* a λ-abstraction where there wasn't one: this is possible using the β-reduction "backwards" like

$$(+\,2\,1) \leftarrow_\beta (\lambda x\,.\,+\,x\,1)\,2.$$

This operation is more formally called *β-abstraction*. When we want to show that two λ-terms are obtainable from each other through β-reduction and β-abstraction, we use the *β-conversion* symbol $\underset{\beta}{\longleftrightarrow}$:

$$(+\,2\,1) \underset{\beta}{\longleftrightarrow} (\lambda x\,.\,+\,x\,1)\,2.$$

We say that β-conversion is the *equivalence relation generated by* $\rightarrow_\beta$. Since the other relations that we'll introduce in this section are almost always used in a symmetric way (like β-conversion), we use undecorated arrows $\leftarrow$ and $\rightarrow$ to express β-abstraction and β-reduction respectively. An undecorated double arrow $\longleftrightarrow$ will be used to denote zero or more conversions of any kind.

2.3.3 α-conversion

Consider the two λ-terms

$$A := (\lambda x\,.\,+\,x\,1) \qquad B := (\lambda y\,.\,+\,y\,1).$$

According to what we have said, A and B are different (i.e. $A \not\equiv B$). It should be clear, however, that their "effect" is the same when they are applied to other λ-terms – they β-reduce to the same term, since all that differs between the two is just the name of the formal parameter. Luckily, λ-calculus provides a relation between these "equivalent" terms which allows us to change the parameter's name at will, provided that the new name doesn't occur free in the body of the λ-abstraction.

Definition 2.7 (α-reduction)**.** The relation of *α-reduction*, $\rightarrow_\alpha \subseteq \mathit{Ter}(\lambda) \times \mathit{Ter}(\lambda)$, is defined by

$$\rightarrow_\alpha = \{\ ((\lambda x\,.\,M),\ (\lambda y\,.\,M[y/x])) \mid y \notin FV(\lambda x\,.\,M)\ \}\,. \qquad \blacksquare$$

The equivalence relation generated by $\to_\alpha$ is denoted by $\underset{\alpha}{\longleftrightarrow}$ and it's called α-conversion; thanks to it, we can rename formal paramers at will.

Example 2.3. Let's return to the troublesome λ-term we've shown in example 2.2:

$$(\lambda x\,.\,(\lambda x\,.\,+\,(-\,x\,2))\,5\,x)\,7.$$

Thanks to α-conversion, we can rename one of the formal parameters and avoid any confusion before applying β-reduction:

$$(\lambda x\,.\,(\lambda x\,.\,+\,(-\,x\,2))\,5\,x)\,7 \underset{\alpha}{\longleftrightarrow} (\lambda y\,.\,(\lambda x\,.\,+\,(-\,x\,2))\,5\,y)\,7. \qquad \blacktriangle$$

It should be clear that α-conversion's only practical use is to avoid name clashes like the one in the previous examples. Nevertheless, its use is sometimes essential.

2.3.4 η-conversion

We said that λ-terms in the same α-conversion equivalence class have the same "effect" when applied to the same term; the λ-calculus has another relation which induces a different equivalence class with the same property. Consider the two expressions

$$A := (\lambda x\,.\,+\,5\,x) \qquad B := (+5);$$

clearly $A \not\equiv B$ and $A \underset{\alpha}{\not\longleftrightarrow} B$, but both A and B reduce to the same term when applied to the same argument.

Definition 2.8 (η-reduction)**.** The relation of *η-reduction*, $\to_\eta \subseteq \mathit{Ter}(\lambda) \times \mathit{Ter}(\lambda)$, is defined by

$$\to_\eta = \{\, ((\lambda x\,.\,M\,x),\,(M)) \mid x \notin \mathit{FV}(M) \,\}\,. \qquad \blacksquare$$

Once again we consider the equivalence relation generated by $\to_\eta$, the η-conversion $\underset{\eta}{\longleftrightarrow}$. Thanks to η-conversion, we can remove unneeded λ-abstractions from our terms: for example,

$$\begin{aligned} &(\lambda x\,.\,+\,5\,x) \\ \underset{\eta}{\longleftrightarrow}\;&(+5) \end{aligned}$$

and both terms evaluate to 10 when applied to the argument 5.

2.4 Recursion

Programs written in one of the many functional programming languages vastly use recursion; λ-calculus, on the other hand, lacks it. It is possible, though, to "convert" a recursive function into a λ-term that doesn't use recursion.

2.4.1 The Y Function

Let's take the *poster child* function for recursion: the factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1) & \text{otherwise} \end{cases}$$

rewritten to be more like our λ-terms:

$$FAC := \lambda x\,.\,\text{IF}\,(=\,x\,0)\,1\,(*\,x\,(FAC\,(-\,x\,1)))$$

At first glance, there's nothing wrong with this expression; however, names like *FAC* are just placeholders for the expressions they define, so it's impossible to reuse *FAC* in its very definition. (Think of our names like C macros.) Let's perform a β-abstraction to "bring out" the name:

$$\begin{aligned} FAC &:= \lambda x\,.\;\ldots\,(FAC\,(-\,x\,1)) \\ \leftarrow FAC &:= \lambda f\,.\,(\lambda x\,.\;\ldots\,(f\,(-\,x\,1)))\,FAC \\ = FAC &:= H\,FAC \end{aligned}$$

where $H := \lambda f\,.\,(\lambda x\,.\,\text{IF}\,(=\,x\,0)\,1\,(*\,x\,(f\,(-\,x\,1))))$. If we forget for a moment that things like *FAC* are names, we can draw a parallel between our expression $FAC := H\,FAC$ and the mathematical notion of *fixed point*: x is a fixed point of f if $f(x) = x$. (The fixed point isn't necessarily unique.) In other words, we can consider *FAC* as the fixed point of the function H, and this fixed point depends only on H.

Imagine now having a nonrecursive λ-expression named Y, which represents a function that takes another function and returns its fixed point (a *fixed point combinator*); we could apply it to H and obtain

$$\begin{aligned} &Y\,H \\ =\,&H(Y\,H) \end{aligned}$$

since $Y\,H$ is a fixed point of H. Our factorial function is now simply written

$$FAC := Y\,H;$$

to prove it, let's show that $FAC\,5 = *\,5\,(FAC\,4)$.

$$\begin{aligned} &FAC\,5 \\ =\,&(Y\,H)\,5 \\ =\,&H\,(Y\,H)\,5 \\ =\,&\lambda f\,.\,(\lambda x\,.\,\text{IF}\,(=\,x\,0)\,1\,(*\,x\,(f\,(-\,x\,1))))\,(Y\,H)\,5 \\ \rightarrow\,&\lambda x\,.\,\text{IF}\,(=\,x\,0)\,1\,(*\,x\,((Y\,H)\,(-\,x\,1)))\,5 \\ \rightarrow\,&\text{IF}\,(=\,5\,0)\,1\,(*\,x\,((Y\,H)\,(-\,5\,1))) \\ \rightarrow\,&(*\,5\,((Y\,H)\,(-\,5\,1))) \\ \rightarrow\,&(*\,5\,((Y\,H)\,4)) \\ =\,&(*\,5\,(FAC\,4)) \end{aligned}$$

The good news is that Y exists and is indeed a nonrecursive λ-term, defined as follows:

$$Y := (\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x))).$$

Let's show that $Y\,H = H\,(Y\,H)$ with this definition:

$$\begin{aligned}
&Y\,H\\
&= (\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)))\,H\\
&\to (\lambda x.\,H\,(x\,x))\,(\lambda x.\,H\,(x\,x))\\
&\to H\,(\lambda x.\,H\,(x\,x))\,(\lambda x.\,H\,(x\,x))\\
&\leftarrow H\,((\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)))\,H)\\
&= H\,(Y\,H).
\end{aligned}$$

It can be demonstrated that the fixed point generated by Y is the *least* fixed point of the function [Stoy, 1981]. Altough this theoretical result is satisfactory, implementing Y as a λ-abstraction is quite inefficient, so most implementations provide it as a built-in function.

2.5 Reduction order

An expression is fully evaluated when there are no more redexes; such expressions are said to be in *normal form*. Evaluation is thus the process of successively reducing redexes until the expression is in normal form; what if an expression contains more than one redex? Reduction can proceed by alternative routes:

$$\begin{aligned}
&(+\,\underline{(*\,3\,4)}\,(*\,7\,8))\\
&\to (+\,12\,\underline{(*\,7\,8)})\\
&\to \underline{(+\,12\,56)}\\
&\to 68
\end{aligned}
\qquad\qquad
\begin{aligned}
&(+\,(*\,3\,4)\,\underline{(*\,7\,8)})\\
&\to (+\,\underline{(*\,3\,4)}\,56)\\
&\to \underline{(+\,12\,56)}\\
&\to 68
\end{aligned}$$

Not every expression has a normal form; there are expressions whose evaluation doesn't end. Consider the abstraction $A := \lambda x.\,x\,x$ and the expression $(A\,A)$. Let's try to evaluate this last one:

$$\begin{aligned}
&A\,A\\
&= (\lambda x.\,x\,x)\,(\lambda x.\,x\,x)\\
&\to (\lambda x.\,x\,x)\,(\lambda x.\,x\,x)\\
&= A\,A.
\end{aligned}$$

$(A\,A)$ evaluates to itself, so it doesn't have a normal form. This is analoguous to an imperative program going into an infinite loop.

2.5.1 Normal Order

There is another complication that arises with evaluation: for a given expression, some reduction sequences may reach normal form while others do not. For example, the expression

$$(\lambda x \,.\, 1)\,(A\,A)$$

where A is the same term defined previously, evaluates to normal form only if we eventually reduce the application of $(\lambda x \,.\, 1)$ to $(A\,A)$; if we continue reducing the application of A to A – obtaining again $(A\,A)$ as the argument for $(\lambda x \,.\, 1)$ – we'll never reach normal form. Fortunately, it isn't possible for two reduction sequences to lead to different normal forms; this is guaranteed by the following theorem (and its corollary).

Theorem 2.1 (Church-Rosser I)**.** *If $E_1 \longleftrightarrow E_2$ there exists an expression E such that $E_1 \to E$ and $E_2 \to E$.*

Corollary. *No expression can be reduced to two different normal forms, not convertible through α-conversion.*

We can informally say that, starting from the same expression, all reduction sequences which terminate will lead to the same result. A second theorem, again by Church and Rosser, assures that for all expressions that evaluate to a normal form there exists a particular reduction order which guarantees that evaluation.

Theorem 2.2 (Church-Rosser II)**.** *If $E_1 \to E_2$ and E_2 is in normal form, there exists a* normal order *reduction sequence from E_1 to E_2.*

Definition 2.9 (Normal order reduction sequence)**.** A reduction sequence is said to be in *normal order* if at every step the redex selected for reduction is the leftmost, outermost one. ■

Together, theorems 2.1 and 2.2 say that there is at most one possible result and that normal order reduction will surely find it. Furthermore, there is no reduction sequence that can give a "wrong" result; the worst possible scenario is nontermination. Note that normal order reduction doesn't guarantee optimality; fortunately, for graph reduction – the representation for expressions that we'll develop in the following chapters – normal order is "almost optimal", i.e. it probably takes more time to find the optimal redex than to use normal order. In the remainder of the work we'll always use normal order, for reasons that will become clear in section 4.2.

2.6 Denotational Semantics of the λ-calculus

Although the three conversion rules shown in the previous sections formally define *purely synctactic transformations* on λ-expressions, the way we've reasoned about them in terms of abstract functions allows us to use them as rules for an operational semantics. To prove that λ-calculus is effectively a good representation for abstract functions, we need to give a denotational semantics for it, i.e. giving an `eval` function which assigns a *value* (or meaning) to every *expression* of the language [Tennent, 1994].

2.6.1 Building the `eval` function

From a mathematical point of view, the `eval` function is a map from expressions to values:

$$\texttt{eval} : \mathit{expr} \to \mathit{val}.$$

For example, we can write $\texttt{eval}[\![+\,1\,2]\!] = 3$; note that 3 is not "the λ-term comprised only by the constant symbol 3" but the abstract numerical value 3. The argument of `eval` is enclosed by double square brackets, emphasizing the fact that it's a purely synctactical object; we might say that in λ-calculus the expression $(+\,1\,2)$ is a *denotation* of the value 3 (hence the name *denotational semantics*).

In order to give a complete `eval` function, we need another ingredient. Which is the value denoted by a variable x (i.e. $\texttt{eval}[\![x]\!]$)? The answer can't clearly be fixed, since the same variable can hold a value in one λ-term and another value in a different one. We need to give the `eval` function some more "contextual" information; namely, a function which maps variables to values, the *environment* ρ. Thus, expressions with variables are evaluated with the help of the environment:

$$\texttt{eval}_\rho[\![x]\!] = \rho(x).$$

An environment can be extended with more bindings; this can be useful when dealing with λ-abstractions. We use the following notation: given an existing environment ρ, its extension with the binding a/x is

$$\rho[a/x](t) = \begin{cases} a & \text{if } t = x \\ \rho(t) & \text{otherwise.} \end{cases}$$

We can thus give the evaluation for a λ-abstraction when it's applied to a generic argument a:

$$\texttt{eval}_\rho[\![(\lambda x\,.\,E)\,a]\!] = \texttt{eval}_{\rho[v/x]}[\![E]\!]$$

where $v = \texttt{eval}_\rho[\![a]\!]$. This is coerent with our intuition that the value of a λ-abstraction applied to an argument should be the value of the body in a

context where the argument is bound to the formal parameter. That leaves only the evaluation for application terms, which is simply

$$\texttt{eval}_\rho[\![\,(E_1\,E_2)\,]\!] = (\texttt{eval}_\rho[\![\,E_1\,]\!])(\texttt{eval}_\rho[\![\,E_2\,]\!]).$$

The way we've built the `eval` function has an important property: if $E_1 \longleftrightarrow E_2$ then surely $\texttt{eval}_\rho[\![\,E_1\,]\!] = \texttt{eval}_\rho[\![\,E_2\,]\!]$. The reverse isn't necessarily true.

2.6.2 The $\perp$ value and strictness

One thing that we're missing is a more specific description of the possible values which `eval` can produce; this collection is called a *domain*. A domain includes all the functions and data values that can be denoted by a λ-expression; that includes the domain's own function space (i.e. functions in the domain can be applied to themselves; this is required by the self-application introduced in the λ-abstraction for Y). A complete study on the theory behind domains is outside the scope of this work, and the interested reader can find it in [Scott, 1982; Abramsky and Jung, 1994]; we take the theory's results for granted. We shall introduce just one important value, which is the result of the evaluation of *all* the expressions that don't have a normal form: the $\perp$ value (pronounced 'bottom'). If E doesn't have a normal form (i.e. the evaluation of E doesn't terminate), then we write $\texttt{eval}_\rho[\![\,E\,]\!] = \perp$.

The $\perp$ value introduces a fair share of problems, and part of this work concerns how to handle them. What if we have a λ-abstraction applied to a non-terminating argument? Thinking outside the λ-calculus, what if we apply a function to the $\perp$ value? The answer is: it depends from the function. It turns out that there are *strict* functions, which always return $\perp$ when their argument is $\perp$, and *non-strict* functions, which may return a non-$\perp$ argument even when their argument is $\perp$[1]; $f(x) = x + 3$ is a strict function, while $g(x) = 2$ is a non-strict one. Unless noted otherwise, we consider built-in functions of our §ambdac as non-strict; for example, our *AND* returns *false* when the first argument evaluates to *false*, skipping the evaluation of the second argument altogether – so even if the second argument is $\perp$ we obtain *false* nevertheless.

[1] Functions that take more than one argument may be strict on some arguments and non-strict on others.

3 | Pattern Matching and the Enriched λ-calculus

3.1 Overview

Taking a step further towards a sufficiently powerful intermediate language, we expand upon the foundations of the λ-calculus introduced in the previous chapter. Since our goal is to translate a subset of the Haskell programming language, we will introduce useful constructs for `let` expressions and pattern matching; a λ-calculus with these constructs is called *enriched λ-calculus*.

3.2 Pattern Matching

The term *pattern matching* refers to the act of checking if a given sequence of tokens follows a precise pattern, and possibly binding some of these tokens into variables. Pattern matching is an useful component of functional programming languages and it's necessary to properly handle *structured data types*.

3.2.1 Structured Data Types

Let's suppose that our applied λ-calculus, defined in the previous chapter, is able to express lists: it has the constant *NIL*, representing the empty list, a function to check if a list is *NIL* (aptly named *ISNIL*), and the function *CONS a b*, representing the concatenation of item *a* to the list *b*[1]. Both of them can be defined in pure λ-calculus (as shown in [S. L. Peyton Jones, 1987] or [Pierce, 2002], so their use in our applied λ-calculus is justified. It turns out that *NIL* and *CONS* by themselves are pretty useless; it's impossible to make even a simple *TAIL* function – that is, a function that removes

[1] In this work we won't discuss about types beyond data structures. We will suppose that functions are "properly" used, i.e. the second argument of *CONS* is always a list. The interested reader can consult [S. L. Peyton Jones, 1987] for an introduction to type checking.

the first element of the list – using only them.

$$TAIL := \lambda x\,.\,\text{IF}\,(ISNIL\,x)\,NIL\,(\text{tail of } x?)$$

The λ-calculus sees x as a monolithic block; it can check wherever $x\,ISNIL$, but it cannot "see" how x is composed. If we could specify the structure of the variables in λ-abstractions, the *TAIL* function would become

$$TAIL := \lambda x\,.\,\text{IF}\,(ISNIL\,x)\,NIL\,((\lambda\,(CONS\,s\,t)\,.\,t)\,x)$$

Here, we say that if x is not *NIL* then surely is a *CONS* of two arguments, say s and t, so we *match* the variable x into the *pattern* $(CONS\,s\,t)$. Patterns will be formally introduced in the next section.

Lists are a kind of *structured type*, a mechanism for defining new data types in many functional languages. *NIL* and *CONS* are called *constructors* of the type; *NIL* has arity 0 (it doesn't take any argument), while *CONS* has arity 2. For structured types, constructors – together with their arguments – are what variables denote; the only way to "unpack" the arguments of a constructor is through pattern matching. We use the standard Haskell notation for defining new structured types[2]:

```
data List a = Nil | Cons a (List a)
```

This statement declares that a `List` (whose elements are of a generic type `a`) is either a `Nil` or a `Cons` of an element and another `List`.

A general structured type `T` can be defined as follows:

$$\begin{aligned}\texttt{data T} = {} & \texttt{c}_1\,\texttt{T}_{1,1}\,\texttt{T}_{1,2}\,\ldots\,\texttt{T}_{1,\texttt{r}_1} \\ \mid {} & \texttt{c}_2\,\texttt{T}_{2,1}\,\texttt{T}_{2,2}\,\ldots\,\texttt{T}_{2,\texttt{r}_2} \\ \mid {} & \ldots \\ \mid {} & \texttt{c}_\texttt{n}\,\texttt{T}_{\texttt{n},1}\,\texttt{T}_{\texttt{n},2}\,\ldots\,\texttt{T}_{\texttt{n},\texttt{r}_\texttt{n}}\end{aligned}$$

where $\texttt{c}_\texttt{i}$ $(1 \leq i \leq n)$ are the constructors, each of arity r_i, and $\texttt{T}_{\texttt{i},\texttt{p}}$ $(1 \leq p \leq r_i)$ are the types of the constructors' arguments. In type theory, we say that the type `T` is the *sum* (or *discriminate union*)

$$T = T_1 + T_2 + \ldots + T_n$$

where every T_i is a *product*

$$T_i = T_{i,1} \cdot T_{i,2} \cdot \ldots \cdot T_{i,r_i}.$$

To better distinguish between the two cases, we call `t` the only constructor of a product type (from tuple, the most common product type) and $\texttt{s}_\texttt{i}$ the i-th constructor of a sum type.

[2]Haskell – and many other functional languages – actually have a special, built-in way to express lists and other commonly-used types such as tuples.

3.2.2 Patterns in λ-abstractions

In the preceding section, in order to solve our problem of defining a *TAIL* function, we intuitively introduced terms of the form $(\lambda p\,.\,E)$ where p is a *pattern*. We now give a formal definition of patterns.

Definition 3.1 (Patterns)**.** A *pattern* is either

- a variable x;
- a constant k (a number, a boolean, …);
- a *constructor pattern* of the form $(\mathtt{c}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_r)$, where `c` is a constructor of arity r and the $\mathtt{p}_i$s are themselves patterns;
  - a pattern of the form $(\mathtt{s}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_r)$ where `s` is a sum constructor is specifically called a *sum pattern*;
  - conversely, a pattern of the form $(\mathtt{t}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_r)$ where `t` is a product constructor is called a *product pattern*. ■

Let's return to the *TAIL* function: in Haskell we can write it as follows.

```
myTail Nil = Nil
myTail (Cons x xs) = xs
```

This is indeed how pattern matching works in this language; instead of having an IF to check if a list is Nil, we use *multiple equations*. When calling `myTail l`, where `l` is a list, the language first tries to match `l` with `Nil`; if that fails, it tries to match `l` with a generic `Cons` binding its arguments to `x` and `xs` respectively. Since a list is either a `Nil` or a `Cons`, the matching is *exhaustive*: we explored all possible cases. The general form for pattern matching arguments of a function `f` in Haskell is

$$\begin{aligned}
\mathtt{f}\ \mathtt{p}_1 &= \mathtt{E}_1\\
\mathtt{f}\ \mathtt{p}_2 &= \mathtt{E}_2\\
&\ldots\\
\mathtt{f}\ \mathtt{p}_n &= \mathtt{E}_n.
\end{aligned}$$

When `f` is applied, its actual parameters get tested against the first pattern $\mathtt{p}_1$: if there's a match, the returned value is the evaluation of $\mathtt{E}_1$, else the matching *fails*, the parameters get tested against $\mathtt{p}_2$ and so on. Obviously, the matching can be *non-exhaustive*: if the $\mathtt{p}_i$s don't cover all the possible values that the arguments of the function can take, the entire matching has to fail with an *error*.

The way Haskell manages pattern matching suggests how to handle it in our enriched λ-calculus:

- a matching can *fail*, so we need a constant *FAIL* which is returned in this case;
- if a matching has failed, we need a way to select the next equation; we introduce the infix function $[\!]$ (pronounced 'fatbar'), with the following semantics:

$$\begin{aligned} a \,[\!]\, b &= a \qquad \text{if } a \neq \bot \text{ and } a \neq \mathit{FAIL} \\ \mathit{FAIL} \,[\!]\, b &= b \\ \bot \,[\!]\, b &= \bot; \end{aligned}$$

- if *all* the matchings have failed, we need another constant *ERROR* to express failure for the entire operation.

With these additional tools, we can finally show the λ-term equivalent to the Haskell code seen previously:

$$\begin{aligned} f := \lambda x \,.\, (\, & ((\lambda p'_1 \,.\, E'_1)\, x) \\ & [\!]\, ((\lambda p'_2 \,.\, E'_2)\, x) \\ & \quad \ldots \\ & [\!]\, ((\lambda p'_n \,.\, E'_n)\, x) \\ & [\!]\, \mathit{ERROR}\,). \end{aligned}$$

Here, p'_i and E'_i are the result of translating respectively the pattern $\mathtt{p_i}$ and the expression $\mathtt{E_i}$, and x is a new variable which doesn't occur free in any of the E'_is. We will discuss in a short while on how the actual translation from Haskell to the λ-calculus is done.

That leaves only functions with multiple arguments, of the form

$$\mathtt{f\ p_1\ \ldots\ p_n = E}$$

where the $\mathtt{p_i}$s are patterns. Like before, it makes sense to translate the patterns and use the translations with new variables:

$$f := \lambda v_1 \,.\, \ldots\, \lambda v_n \,.\, (((\lambda p'_1 \,.\, \ldots\, \lambda p'_n \,.\, E')\, v_1\, \ldots\, v_n)\, [\!]\, \mathit{ERROR}).$$

Let's examine what happens when a matching – say, the i-th – fails. At the end of the $(i-1)$-th substitution, we have

$$(\lambda p'_i \,.\, \ldots\, \lambda p'_n \,.\, E')\, a_i\, \ldots\, a_n$$

where the a_ks are the actual parameters that are being matched. We proceed, and the i-th matching fails, so we have this strange result:

$$FAIL\ a_{i+1}\ \ldots\ a_n$$

It turns out that an additional reduction rule is needed; if T is a λ-term, then

$$FAIL\ T \to FAIL.$$

With this rule we can end the evaluation of $((\lambda p'_1\,.\ \ldots\ \lambda p'_n\,.\,E')\ v_1\ \ldots\ v_n)$ with a single *FAIL*, so $[\![$ selects *ERROR* as the final result.

3.2.3 Denotational semantics of pattern-matching λ-abstractions

In the previous chapter we gave a denotational semantics for the applied λ-calculus; since we introduced a new type of λ-term – $(\lambda p\,.\,E)$, where p is a pattern – we ought to extend the `eval` function to give values to the new expressions that can be built using it. We proceed by cases, guided by the definition of patterns given in section 3.2.2.

Variable Patterns

If the pattern p is made only by a variable v, the term $(\lambda p\,.\,E)$ is a plain abstraction term, whose semantics have been discussed in section 2.6.1:

$$\texttt{eval}_\rho[\![\,(\lambda p\,.\,E)\,a\,]\!] = \texttt{eval}_\rho[\![\,(\lambda v\,.\,E)\,a\,]\!].$$

Constant Patterns

A constant pattern of the form $(\lambda k\,.\,E)$, where k is a constant. Its semantics is easy: either the argument of the application, say a, is equal to k, or it isn't. If the evaluation of a doesn't terminate, neither should the evaluation of the abstraction. The `eval` function is thus extended as follows:

$$\texttt{eval}_\rho[\![\,(\lambda k\,.\,E)\,a\,]\!] = \begin{cases} \texttt{eval}_\rho[\![\,E\,]\!] & \text{if } \texttt{eval}_\rho[\![\,a\,]\!] = \texttt{eval}_\rho[\![\,k\,]\!] \\ FAIL & \text{if } \texttt{eval}_\rho[\![\,a\,]\!] \neq \texttt{eval}_\rho[\![\,k\,]\!] \\ \bot & \text{if } \texttt{eval}_\rho[\![\,a\,]\!] = \bot. \end{cases}$$

Sum Patterns

Sum patterns require a certain amount of attention. In order to evaluate a term of the form $((\lambda\,(s\,p_1\ \ldots\ p_n)\,.\,E)\,A)$, we need to evaluate A first to see "what kind" of object it is; at the same time, though, we want our evaluation

to be lazy (i.e. non-strict), so we evaluate A only to its constructor form (we don't evaluate its arguments). The rule is:

- if $\texttt{eval}_\rho[\![A]\!] = (s\ [\![a_1]\!]\ \ldots\ [\![a_n]\!])$, where the a_is are syntactic objects representing the unevaluated arguments of the constructor s,

$$\texttt{eval}_\rho[\![(\lambda\ (s\ p_1\ \ldots\ p_n)\ .\ E)\ A]\!] = \texttt{eval}_\rho[\![(\lambda p_1\ .\ \ldots\ \lambda p_n\ .\ E)\ a_1\ \ldots\ a_n]\!];$$

- if $\texttt{eval}_\rho[\![A]\!] = (s'\ \ldots)$, where $s \neq s'$,

$$\texttt{eval}_\rho[\![(\lambda\ (s\ p_1\ \ldots\ p_n)\ .\ E)\ A]\!] = \mathit{FAIL};$$

- if $\texttt{eval}_\rho[\![A]\!] = \bot$,

$$\texttt{eval}_\rho[\![(\lambda\ (s\ p_1\ \ldots\ p_n)\ .\ E)\ A]\!] = \bot.$$

Product Patterns

When talking about product patterns, there's no need to evaluate the top-level of the argument A since there is only one constructor for the type we're considering[3] – so, we don't need to discriminate between multiple constructors. This leads to two different ways to define the `eval` function for product patterns: *strict* product matching, where A's top-level is evaluated anyway, or *lazy* (non-strict) product matching, where we defer the evaluation to a later time. The equation for lazy product matching is

$$\begin{aligned}\texttt{eval}_\rho[\![(\lambda\ (t\ p_1\ \ldots\ p_n)\ .\ E)\ A]\!] = \texttt{eval}_\rho[\![(\lambda p_1\ .\ \ldots\ \lambda p_n\ .\ E)]\!]\ &(\textsc{Eval}\text{-}t\text{-}1\ A)\\ &(\textsc{Eval}\text{-}t\text{-}2\ A)\\ &\ldots\\ &(\textsc{Eval}\text{-}t\text{-}\mathrm{n}\ A)\end{aligned}$$

The Eval-*t*-i (where t is the type we're considering) are auxiliary functions defined as follows:

$$\textsc{Eval}\text{-}t\text{-}\mathrm{i}\ (t\ x_1\ \ldots\ x_i\ \ldots\ x_r) = \begin{cases} a_i & \text{if } \texttt{eval}_\rho[\![x_i]\!] = a_i \\ \bot & \text{if } \texttt{eval}_\rho[\![x_i]\!] = \bot. \end{cases}$$

With lazy product-matching, the evaluation of the arguments of the abstraction can be postponed until necessary– in fact, in some cases, they will never be evaluated.

[3] Again, we won't talk about type-checking; we assume that the argument is always of the "right" type, or $\bot$ as the worst case.

3.3 *let* and *letrec* expressions

One of the fundamental constructs in Haskell (and, indeed, in any other functional language) is the *definition*, a way to bind variable names to values. To simplify the translation from Haskell into λ-calculus, we add two types of expression into the enriched λ-calculus: *let* and *letrec*.

3.3.1 Simple *let*-expressions

The first type of definition construct that we introduce is the simple *let* (called "simple" by contrast with *pattern-matching let*).

Definition 3.2 (Simple *let*-expression)**.** A *simple let-expression* has syntax

$$let\, v = B\, in\, E$$

where v is the *bound variable*, B is the *definition body* and $v = B$ is the *definition* of the *let*. B and E are expressions in the enriched λ-calculus. The *let* construct binds B to v, but only in the scope of E. ■

For practical purposes, we allow multiple definitions in the same *let*; writing

$$\begin{aligned} let\, v &= B \\ \quad w &= C \\ in\, E & \end{aligned}$$

is the same as writing $let\, v = B\, in\, (let\, w = C\, in\, E)$.

3.3.2 Simple *letrec*-expressions

The term *letrec* is short for "let recursively"; simple *letrec*-expressions are similar to simple *let*-expressions, save for the fact that all variables bound by a *letrec* are in the scope of all its definition bodies, to allow recursion. A simple *letrec*-expression has syntax

$$\begin{aligned} letrec\, v_1 &= E_1 \\ v_2 &= E_2 \\ &\cdots \\ v_n &= E_n \\ in\, E. & \end{aligned}$$

3.3.3 Patterns in *let*(*rec*)-expressions

In simple *let*(*rec*)-expressions, the left-hand side of a definition is always a single variable; however, functional programming languages allow *patterns* to be in the left-hand side of a definition.

```
let (Cons x xs) = myTail (Cons 2 (Cons 3 Nil))
 in x
```

This Haskell code is perfectly valid and the result of the evaluation is `3` (the head of the list returned by `myTail`). We want a similar behaviour for the enriched λ-calculus, so we introduce *pattern-matching let*(*rec*)-expressions; those expressions can have *refutable* or *irrefutable* patterns as the left-hand side of definitions.

Definition 3.3 (Irrefutable patterns)**.** A pattern p is *irrefutable* if it is

- either a variable v
- or a product pattern of the form $(t\, p_1\, p_2\, \ldots\, p_r)$ where the p_is are irrefutable patterns.

Otherwise, p is said to be *refutable*. ■

let(*rec*)-expressions with irrefutable patterns are easier to manage: once the argument has passed type-checking, the matching cannot fail. For this reason, we treat irrefutable *let*(*rec*)s separately.

3.3.4 Irrefutable *let*-expressions

Irrefutable *let*s are of the form

$$let\ p = B\ in\ E$$

where p is an irrefutable pattern and B and E are expressions. Since many implementations provide simple *let*(*rec*) as a built-in construct for efficiency and type-checking reasons [S. L. Peyton Jones, 1987], we give a translation rule from irrefutable *let*s to simple *let*s.

- If p is a single variable v, then the irrefutable *let*-expression is in fact a simple *let*-expression:

$$let\ p = B\ in\ E \quad \equiv \quad let\ v = B\ in\ E \qquad \text{if } p = v.$$

- If p is a product pattern $(t\, p_1\, \ldots\, p_r)$, then we can use the auxiliary functions SEL-t-i to translate the irrefutable *let* into a simple *let*:

$$\begin{aligned} let\,(t\,p_1\,\ldots\,p_r) = B\,in\,E \quad \equiv \quad & let\,v = B \\ & in\,(let\,p_1 = \text{SEL-}t\text{-1}\,v \\ & \qquad p_2 = \text{SEL-}t\text{-2}\,v \\ & \qquad \ldots \\ & \qquad p_r = \text{SEL-}t\text{-r}\,v \\ & \qquad in\,E) \end{aligned}$$

  The SEL-t-i functions are similar to the EVAL-t-i ones, with the important difference that the latter causes the evaluation of the i-th argument while the former just returns it.

$$\text{SEL-}t\text{-i}\,a = \begin{cases} x_i & \text{if } a = (t\,x_1\,\ldots\,x_i\,\ldots\,x_r) \\ \bot & \text{if } a = \bot \end{cases}$$

  The translation shown above is consistant the lazy product-matching evaluation style.

3.3.5 Irrefutable *letrec*-expressions

Irrefutable *letrec*s are similar to their non-recursive cousins, but have to account for the names to be in the scope of all the definition bodies. An irrefutable *letrec*-expression with a single definition it's easily translated into an equivalent irrefutable *let*-expression:

$$letrec\,p = B\,in\,E \quad \equiv \quad let\,p = Y\,(\lambda p\,.\,B)\,in\,E.$$

For multiple definitions, we can come up with a shortcut: instead of dealing with every definition separately, we can "pack" all the patterns in a single product pattern (with a new product constructor t_{new}), and since all patterns are irrefutable then the new product pattern will be irrefutable too.

$$\begin{aligned} letrec\,p_1 = B_1 \quad \equiv \quad & letrec\,(t_{new}\,p_1\,\ldots\,p_n) = (t_{new}\,B_1\,\ldots\,B_n)\,in\,E \\ & p_2 = B_2 \\ & \ldots \\ & p_n = B_n \\ in\,E & \end{aligned}$$

After this transformation, we can treat the resulting *letrec*-expression as a normal single-definition *letrec*.

3.3.6 General *let*(*rec*)-expressions

We've seen that irrefutable *let*(*rec*)-expressions are easy to treat; unfortunately, even a single constant or sum sub-pattern in the left-hand side of a definition makes the entire pattern (and thus the entire expression) refutable. The problem with refutable patterns is that they can fail matching; we should always check if the definition body *conforms* to the pattern at the left-hand side of the definition. We can do this *conformality check* in two moments:

- when the evaluation for the entire expression begins;
- on the first occasion where the pattern components are used.

The second approach is clearly the lazy one, so that's the way we should do it. We follow a similar approach to the one of section 3.3.5: we "pack" the refutable pattern's variables into a new, *irrefutable* product pattern that will go on the left-hand side of the definition, moving the refutable portion into the right-hand one. First of all, we should formally define the set of variables occurring in a pattern.

Definition 3.4 (Set of variables of a pattern)**.** For a given pattern p, its *set of variables* $Var(p)$ is defined as follows:

- if p is a variable v, then $Var(p) = \{v\}$;
- if p is a constant k, then $Var(p) = \varnothing$;
- if p is a structured pattern $(c\, p_1 \ldots p_r)$, then

$$Var(p) = Var(p_1) \cup \ldots \cup Var(p_r).$$ ■

Definition 3.5 (Conformality transformation)**.** Given a definition of the form $p = B$ (it doesn't matter if in a *let* or in a *letrec*), where p is a refutable pattern, we can apply a *conformality transformation* on it:

$$p = B \quad \equiv \quad (t_{new}\, v_1 \ldots v_n) = ((\lambda p\,.\,(t_{new}\, v_1 \ldots v_n))\, B) \,[]\, ERROR$$

where $\{v_1, \ldots, v_n\} = Var(p)$ and t_{new} is a new product constructor of arity n. ■

Thanks to the conformality transformation, we can translate general *let*(*rec*)s into irrefutable *let*(*rec*)s.

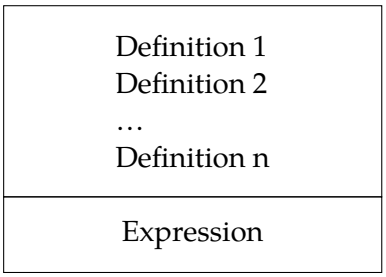
3.3.7 Grammar for the enriched λ-calculus

With the introduction of *let*(*rec*) constructs, our enriched λ-calculus is now complete. Its syntax can be defined by the following grammar.

$\langle Expr \rangle \rightarrow \langle Constant \rangle$	Constants
$\mid \langle Variable \rangle$	Variables
$\mid \langle Expr \rangle \ \langle Expr \rangle$	Applications
$\mid \boldsymbol{\lambda} \ \langle Pattern \rangle \ \mathbf{.} \ \langle Expr \rangle$	λ-abstractions
$\mid \langle Expr \rangle \ [] \langle Expr \rangle$	fatbar
$\mid \mathbf{let} \ \langle DecList \rangle \ \mathbf{in} \ \langle Expr \rangle$	*let*-expressions
$\mid \mathbf{letrec} \ \langle DecList \rangle \ \mathbf{in} \ \langle Expr \rangle$	*letrec*-expressions
$\langle Dec \rangle \rightarrow \langle Pattern \rangle = \langle Expr \rangle$	Definitions
$\langle DecList \rangle \rightarrow \langle Dec \rangle \mid \langle Dec \rangle, \langle DecList \rangle$	Multiple definitions
$\langle Pattern \rangle \rightarrow \langle Constant \rangle$	Constant patterns
$\mid \langle Variable \rangle$	Variable patterns
$\mid \langle Constructor \rangle \ \langle PatList \rangle$	Constructor patterns
$\langle PatList \rangle \rightarrow \langle Pattern \rangle \mid \langle Pattern \rangle \ \langle PatList \rangle$	Constructor arguments

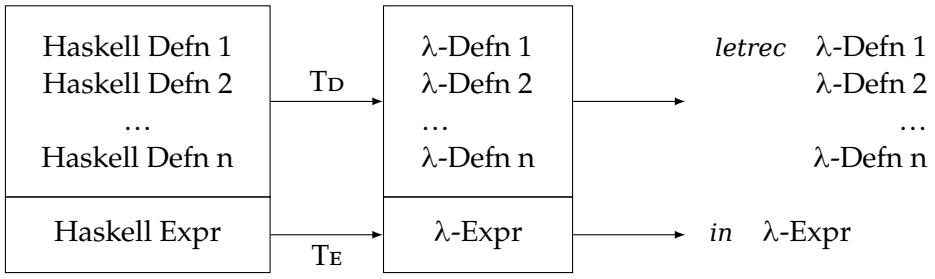
3.4 From Haskell to the enriched λ-calculus

Having all the ingredients set in place, we can finally begin to describe how to translate a code written in (a subset of) Haskell into the enriched λ-calculus we've introduced in this chapter. First of all, we define two translation schemes: one for *definitions* and one for *expressions*. After all, a Haskell program can be viewed as a set of definitions needed to evaluate an expression:



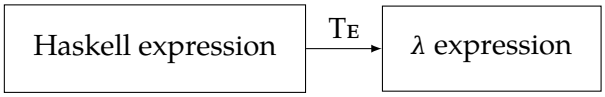
The expressions will be translated by a function TE (short for 'Translate Expressions') which takes a Haskell expression as its input and gives a λ-term as its output, while definitions will likewise be translated by a function TD ('Translate Definitions') which outputs definitions suitable for *let*(*rec*)s of

the enriched λ-calculus. The outputs of these functions are used to build the final *letrec*-expression[4]:



3.4.1 The TE translation scheme

We already said that the translation scheme for expression will be implemented as a function TE:



We'll give a precise definition of TE analyzing every possible form of an Haskell expression.

Constants

If the enriched λ-calculus contains the same set of constants as Haskell (something that we take for granted), the translation of a constant is straightforward:

$$\mathrm{TE}[\![\, \mathsf{k} \,]\!] \quad \equiv \quad k.$$

Note that built-in functions ($+$, $-$, etc) are treated in the same way as constants.

Variables

The rule for translating variable names (including user-defined constructors and functions) is simple enough:

$$\mathrm{TE}[\![\, \mathsf{v} \,]\!] \quad \equiv \quad v.$$

[4]Since we don't know if there's recursion in the definitions, it's safer to always stick with a *letrec*-expression. This introduces a bigger overhead, so an optimization named *dependency analysis* is used to discriminate wherever a *letrec* is necessary or it can be replaced with a *let*. Dependency analysis is actually required for type-checking, and is described in [S. L. Peyton Jones, 1987].

Function application

In most cases, function application in Haskell is identical to the one of the λ-calculus: a simple juxtaposition of expressions. In this case we have

$$\mathrm{TE}[\![\, \mathtt{E_1\,E_2} \,]\!] \quad \equiv \quad \mathrm{TE}[\![\, \mathtt{E_1} \,]\!]\, \mathrm{TE}[\![\, \mathtt{E_2} \,]\!].$$

Some common operators (like the arithmetic ones) can be used infix. The translation of expressions having (built-in) infix operators is

$$\mathrm{TE}[\![\, \mathtt{E_1\,infix\,E_2} \,]\!] \quad \equiv \quad \mathrm{TE}[\![\, \mathtt{infix} \,]\!]\, \mathrm{TE}[\![\, \mathtt{E_1} \,]\!]\, \mathrm{TE}[\![\, \mathtt{E_2} \,]\!].$$

Furthermore, any user-defined function that takes two arguments can be made infix in Haskell by surrounding its name with backticks (`` `fun` ``). This case is easily taken care of:

$$\mathrm{TE}[\![\, \mathtt{E_1\,`fun`\,E_2} \,]\!] \quad \equiv \quad \mathrm{TE}[\![\, \mathtt{fun} \,]\!]\, \mathrm{TE}[\![\, \mathtt{E_1} \,]\!]\, \mathrm{TE}[\![\, \mathtt{E_2} \,]\!].$$

Lists

Lists in Haskell have a special syntax: the "cons" constructor is an infix operator, represented by the symbol `:`, while empty lists are denoted by empty square brackets `[]`. There's another syntax for non-empty lists: `[a, b, c, ...]`, which is syntactic sugar for `a:b:c:...:[]`. The rules for lists are:

$$\begin{aligned}
\mathrm{TE}[\![\, \mathtt{:} \,]\!] &\equiv \mathit{CONS} \\
\mathrm{TE}[\![\, \mathtt{[]} \,]\!] &\equiv \mathit{NIL} \\
\mathrm{TE}[\![\, \mathtt{[E]} \,]\!] &\equiv \mathit{CONS}\, \mathrm{TE}[\![\, \mathtt{E} \,]\!]\, \mathit{NIL} \\
\mathrm{TE}[\![\, \mathtt{[E_1,E_2,\ldots,E_n]} \,]\!] &\equiv \mathit{CONS}\, \mathrm{TE}[\![\, \mathtt{E_1} \,]\!]\, \mathrm{TE}[\![\, \mathtt{[E_2,\ldots,E_n]} \,]\!].
\end{aligned}$$

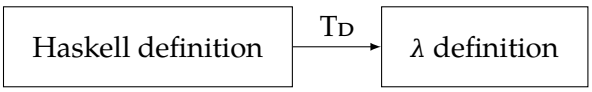
Tuples

Like lists, tuples have a special syntax in Haskell: their elements are written inside round brackets `()`, separated by commas. So, for example, `(a,b,c,d)` is the 4-tuple composed by the elements a b c d. We need a translation rule for every possible *n*-tuple:

$$\begin{aligned}
\mathrm{TE}[\![\, \mathtt{(E_1,E_2)} \,]\!] &\equiv \mathit{PAIR}\, \mathrm{TE}[\![\, \mathtt{E_1} \,]\!]\, \mathrm{TE}[\![\, \mathtt{E_2} \,]\!] \\
\mathrm{TE}[\![\, \mathtt{(E_1,E_2,E_3)} \,]\!] &\equiv \mathit{TRIPLE}\, \mathrm{TE}[\![\, \mathtt{E_1} \,]\!]\, \mathrm{TE}[\![\, \mathtt{E_2} \,]\!]\, \mathrm{TE}[\![\, \mathtt{E_3} \,]\!] \\
&\ldots \\
\mathrm{TE}[\![\, \mathtt{(E_1,\ldots,E_n)} \,]\!] &\equiv \mathit{n\text{-}TUPLE}\, \mathrm{TE}[\![\, \mathtt{E_1} \,]\!]\, \ldots\, \mathrm{TE}[\![\, \mathtt{E_n} \,]\!]
\end{aligned}$$

3.4.2 The $\mathrm{T_D}$ translation scheme

Translating definitions is somewhat easier, since there are few cases and a lot of the work has already been done when we introduced pattern-matching (section 3.2).



Variable definitions

An Haskell definition like

```
v = 1 + 2
```

where `v` is a variable, can be translated easily: the variable retains its name, while the right-hand expression is handled by the $\mathrm{T_E}$ translation scheme.

$$\mathrm{T_D}[\![\ \mathtt{v\ =\ E}\ ]\!] \quad \equiv \quad v = \mathrm{T_E}[\![\ \mathtt{E}\ ]\!]$$

Pattern definitions

Definitions in which the left-hand side is a pattern are similar to variable definitions, with the important caveat that the pattern has to be handled by the $\mathrm{T_E}$ translation scheme:

$$\mathrm{T_D}[\![\ \mathtt{p\ =\ E}\ ]\!] \quad \equiv \quad \mathrm{T_E}[\![\ \mathtt{p}\ ]\!] = \mathrm{T_E}[\![\ \mathtt{E}\ ]\!]$$

Simple function definitions

Functions in Haskell are translated with λ-abstractions, using $\mathrm{T_E}$ to translate the function body. Overall, it's similar to the translation of a variable definition.

$$\mathrm{T_D}[\![\ \mathtt{f\ v_1\ \ldots\ v_n\ =\ E}\ ]\!] \quad \equiv \quad f = \lambda v_1 . \ \ldots\ \lambda v_n . \ \mathrm{T_E}[\![\ \mathtt{E}\ ]\!]$$

Pattern-matching function definitions

If the arguments of the function are patterns, we should handle the case of a failed matching; the translation of the patterns is, as usual, handled by $\mathrm{T_E}$. The general scheme is identical to the one we showed in section 3.2.

$$\begin{aligned}\mathrm{T_D}[\![\ \mathtt{f\ p_1\ \ldots\ p_n\ =\ E}\ ]\!] \quad \equiv& \\ f = \lambda v_1 . \ \ldots\ \lambda v_n . \ &(((\lambda \mathrm{T_E}[\![\ \mathtt{p_1}\ ]\!] . \ \ldots\ \lambda \mathrm{T_E}[\![\ \mathtt{p_n}\ ]\!] . \mathrm{T_E}[\![\ \mathtt{E}\ ]\!])\ v_1\ \ldots\ v_n) \\ &[\!]\ \mathit{ERROR})\end{aligned}$$

Multiple function definitions

Extending the previous definition to include multiple definitions is easy:

$$\mathrm{T_D}\left[\!\!\left[\begin{array}{l} \mathtt{f\ p_{1,1}\ \ldots\ p_{1,n}\ =\ E_1} \\ \mathtt{f\ p_{2,1}\ \ldots\ p_{2,n}\ =\ E_2} \\ \ldots \\ \mathtt{f\ p_{n,1}\ \ldots\ p_{n,n}\ =\ E_n} \end{array}\right]\!\!\right] \equiv$$

$$\begin{aligned} f = \lambda v_1 . \ \ldots\ \lambda v_n . \ & (((\lambda \mathrm{T_E}[\![\, \mathtt{p_{1,1}} \,]\!]. \ \ldots\ \lambda \mathrm{T_E}[\![\, \mathtt{p_{1,n}} \,]\!].\mathrm{T_E}[\![\, \mathtt{E_1} \,]\!])\ v_1\ \ldots\ v_n) \\ & [\!]\ ((\lambda \mathrm{T_E}[\![\, \mathtt{p_{2,1}} \,]\!]. \ \ldots\ \lambda \mathrm{T_E}[\![\, \mathtt{p_{2,n}} \,]\!].\mathrm{T_E}[\![\, \mathtt{E_2} \,]\!])\ v_1\ \ldots\ v_n) \\ & \ldots \\ & [\!]\ ((\lambda \mathrm{T_E}[\![\, \mathtt{p_{n,1}} \,]\!]. \ \ldots\ \lambda \mathrm{T_E}[\![\, \mathtt{p_{n,n}} \,]\!].\mathrm{T_E}[\![\, \mathtt{E_n} \,]\!])\ v_1\ \ldots\ v_n) \\ & [\!]\ \mathit{ERROR}) \end{aligned}$$

where the v_is are variables not free in any of the expressions E_j.

3.4.3 An example

Let's try to translate the following Haskell program: a function that takes a list of numbers and a number n, and outputs the sum of all elements of the list multiplied by n.

```
sumN [] n = 0
sumN (x:xs) n = (x * n) + (sumN xs n)
```

```
sumN [1,2,3] 2
```

Translating the definitions

$$
\begin{aligned}
&\mathrm{T_D}\left[\!\!\left[\begin{array}{l}\texttt{sumN [] n = 0}\\ \texttt{sumN (x : xs) n = (x * n) + (sumN xs n)}\end{array}\right]\!\!\right]\\
\equiv\\
&\quad sumN = \lambda v_1 . \lambda v_2 . (((\lambda \mathrm{T_E}[\![\,\texttt{[]}\,]\!] . \lambda \mathrm{T_E}[\![\,\texttt{n}\,]\!] . \mathrm{T_E}[\![\,\texttt{0}\,]\!])\, v_1\, v_2)\\
&\qquad\qquad [\!] ((\lambda \mathrm{T_E}[\![\,\texttt{(x : xs)}\,]\!] . \lambda \mathrm{T_E}[\![\,\texttt{n}\,]\!] .\\
&\qquad\qquad\quad \mathrm{T_E}[\![\,\texttt{(x * n) + (sumN xs n)}\,]\!])\, v_1\, v_2)\\
&\qquad\qquad [\!] \mathit{ERROR})\\
\equiv\\
&\quad sumN = \lambda v_1 . \lambda v_2 . (((\lambda\, NIL . \lambda n . 0)\, v_1\, v_2)\\
&\qquad\qquad [\!] ((\lambda\, (CONS\, x\, xs) . \lambda n .\\
&\qquad\qquad\quad + \mathrm{T_E}[\![\,\texttt{(x * n)}\,]\!]\, \mathrm{T_E}[\![\,\texttt{(sumN xs n)}\,]\!])\, v_1\, v_2)\\
&\qquad\qquad [\!] \mathit{ERROR})\\
\equiv\\
&\quad sumN = \lambda v_1 . \lambda v_2 . (((\lambda\, NIL . \lambda n . 0)\, v_1\, v_2)\\
&\qquad\qquad [\!] ((\lambda\, (CONS\, x\, xs) . \lambda n .\\
&\qquad\qquad\quad + (* \mathrm{T_E}[\![\,\texttt{x}\,]\!]\, \mathrm{T_E}[\![\,\texttt{n}\,]\!])\, (\mathrm{T_E}[\![\,\texttt{sumN}\,]\!]\, \mathrm{T_E}[\![\,\texttt{xs}\,]\!]\, \mathrm{T_E}[\![\,\texttt{n}\,]\!]))\, v_1\, v_2)\\
&\qquad\qquad [\!] \mathit{ERROR})\\
\equiv\\
&\quad sumN = \lambda v_1 . \lambda v_2 . (((\lambda\, NIL . \lambda n . 0)\, v_1\, v_2)\\
&\qquad\qquad [\!] ((\lambda\, (CONS\, x\, xs) . \lambda n . (* x\, n)\, (sumN\, xs\, n))\, v_1\, v_2)\\
&\qquad\qquad [\!] \mathit{ERROR})
\end{aligned}
$$

Translating the expression

$$
\begin{aligned}
&\mathrm{T_E}[\![\,\texttt{sumN [1,2,3] 2}\,]\!]\\
\equiv\\
&\mathrm{T_E}[\![\,\texttt{sumN}\,]\!]\, \mathrm{T_E}[\![\,\texttt{[1,2,3]}\,]\!]\, \mathrm{T_E}[\![\,\texttt{2}\,]\!]\\
\equiv\\
&sumN\, (CONS\, 1\, \mathrm{T_E}[\![\,\texttt{[2,3]}\,]\!])\, 2\\
\equiv\\
&sumN\, (CONS\, 1\, (CONS\, 2\, \mathrm{T_E}[\![\,\texttt{[3]}\,]\!]))\, 2\\
\equiv\\
&sumN\, (CONS\, 1\, (CONS\, 2\, (CONS\, 3\, NIL)))\, 2
\end{aligned}
$$

The final λ-expression

$$
\begin{array}{l}
\mathit{letrec} \\
\quad \mathit{sumN} = \lambda v_1 \,.\, \lambda v_2 \,.\, (((\lambda\, \mathit{NIL} \,.\, \lambda n \,.\, 0)\, v_1\, v_2) \\
\qquad\qquad\qquad [\!] \; ((\lambda\, (\mathit{CONS}\, x\, xs) \,.\, \lambda n \,.\, (* \, x\, n)\, (\mathit{sumN}\, xs\, n))\, v_1\, v_2) \\
\qquad\qquad\qquad [\!] \; \mathit{ERROR}) \\
\mathit{in} \\
\quad \mathit{sumN}\, (\mathit{CONS}\, 1\, (\mathit{CONS}\, 2\, (\mathit{CONS}\, 3\, \mathit{NIL})))\, 2
\end{array}
$$

3.5 From enriched to applied λ-calculus

The enriched λ-calculus that we introduced in this chapter is quite similar to the applied λ-calculus of chapter 2: since we can consider the constants *FAIL* and *ERROR* as part of the applied λ-calculus, the only new constructs are the $[\!]$ function, pattern-matching λ-abstractions and *let*(*rec*)-expressions. In this section we show how to translate those constructs into equivalent expressions of the applied λ-calculus.

3.5.1 The $[\!]$ function

The $[\!]$ function we introduced in section 3.2.2 is infix; since functions in applied λ-calculus are non-infix, the only step we have to take to translate $[\!]$ is creating a new, non-infix function *FATBAR*:

$$
\begin{array}{ll}
\mathit{FATBAR}\, a\, b = a & \text{if } a \neq \bot \text{ and } a \neq \mathit{FAIL} \\
\mathit{FATBAR}\, \mathit{FAIL}\, b = b & \\
\mathit{FATBAR}\, \bot\, b = \bot. &
\end{array}
$$

With this "new" function, the translation is straightforward:

$$
a \; [\!] \; b \quad \equiv \quad \mathit{FATBAR}\, a\, b.
$$

3.5.2 Pattern-matching λ-abstractions

If the pattern p in the abstraction $(\lambda p \,.\, E)$ is a variable, there's nothing to translate: the expression already belongs to the applied λ-calculus. The other possible cases are when the pattern is a constant, a sum-constructor or a product-constructor: we'll treat those cases separately, guided by the semantics of the `eval` function we defined previously.

Constant patterns

In section 3.2.3 we defined the `eval` function for constant patterns in λ-abstractions:

$$\texttt{eval}_\rho[\![\,(\lambda k\,.\,E)\,a\,]\!] = \begin{cases} \texttt{eval}_\rho[\![\,E\,]\!] & \text{if } \texttt{eval}_\rho[\![\,a\,]\!] = \texttt{eval}_\rho[\![\,k\,]\!] \\ \mathit{FAIL} & \text{if } \texttt{eval}_\rho[\![\,a\,]\!] \neq \texttt{eval}_\rho[\![\,k\,]\!] \\ \bot & \text{if } \texttt{eval}_\rho[\![\,a\,]\!] = \bot. \end{cases}$$

We don't care about $\bot$ values, since we're only working on the syntax of the expression; so, we only have to check if the argument is equal to k, and if not we return *FAIL*. This can be done simply with the built-in IF function:

$$((\lambda k\,.\,E)\,A) \quad \equiv \quad ((\lambda v\,.\,\textsc{if}\ (=\ k\,v)\,E\,\mathit{FAIL})\,A)$$

where v is a new variable which doesn't occur free in E.

Product-constructor patterns

The `eval` function for product patterns is:

$$\begin{aligned}\texttt{eval}_\rho[\![\,(\lambda\,(t\,p_1\,\ldots\,p_n)\,.\,E)\,A\,]\!] = \texttt{eval}_\rho[\![\,(\lambda p_1\,.\,\ldots\,\lambda p_n\,.\,E)\,]\!]\,&(\textsc{Eval}\text{-}t\text{-}1\,A)\\ &(\textsc{Eval}\text{-}t\text{-}2\,A)\\ &\ldots\\ &(\textsc{Eval}\text{-}t\text{-}\mathrm{n}\,A).\end{aligned}$$

Since we're manipulating syntactic objects, we don't want the Eval-*t*-i functions to evaluate our arguments; when using this semantics as a guide for translating product-matching λ-abstractions, we use the similar Sel-*t*-i functions introduced in section 3.3.4:

$$\begin{aligned}((\lambda\,(t\,p_1\,\ldots\,p_n)\,.\,E)\,A) \quad \equiv \quad (\lambda p_1\,.\,\ldots\,\lambda p_n\,.\,E)&(\textsc{Sel}\text{-}t\text{-}1\,A)\\ &\ldots\\ &(\textsc{Sel}\text{-}t\text{-}n\,A).\end{aligned}$$

Notice that the right-hand side of the equation still has pattern-matching λ-abstractions, but it has *smaller* patterns than the original expression; repeated applications of the rules of this section will eliminate them.

Sum-constructor patterns

The `eval` function for sum pattern was defined by cases: if the argument A evaluates to a sum-constructor s with unevaluated arguments $a_1, \ldots, a_r$

then

$$\texttt{eval}_\rho[\![\,(\lambda\,(s\,p_1\,\ldots\,p_n)\,.\,E)\,A\,]\!] = \texttt{eval}_\rho[\![\,(\lambda p_1\,.\,\ldots\,\lambda p_n\,.\,E)\,a_1\,\ldots\,a_n\,]\!];$$

else

$$\texttt{eval}_\rho[\![\,(\lambda\,(s\,p_1\,\ldots\,p_n)\,.\,E)\,A\,]\!] = \mathit{FAIL}.$$

The translation is simple:

$$\begin{aligned}((\lambda\,(s\,p_1\,\ldots\,p_n)\,.\,E)\,A) \quad &\equiv \\ \equiv \quad &\begin{cases}(\lambda p_1\,.\,\ldots\,\lambda p_n\,.\,E)\,(a_1\,\ldots\,a_n) & \text{if } A = (s\,a_1\,\ldots\,a_n) \\ \mathit{FAIL} & \text{if } A = (s'\,a_1\,\ldots\,a_n),\ s \neq s'\end{cases}\end{aligned}$$

3.5.3 *let*(*rec*)-expressions

Since we've shown how to transpose every non-simple *let* and every *letrec* into simple *let*s, a single rule for the latter satisfies our needs:

$$\mathit{let}\,v = B\,\mathit{in}\,E \quad \equiv \quad (\lambda v\,.\,E)\,B.$$

As said before, many implementations provide at least the simple *let* as a built-in construct in the target language, and other implementations actually delay the transformation of any *let*(*rec*)-expression for optimized typechecking [S. L. Peyton Jones, 1987].

4 | Graph Reduction

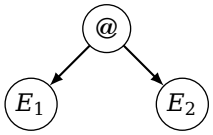
With the introduction of the enriched λ-calculus in the previous chapter, we showed all the necessary steps to translate a Haskell program into a λ-expression. In this chapter we introduce a way to represent these expressions in the computer's memory and how to work on this representation to obtain the final result – an expression in normal form.

4.1 Graph representation

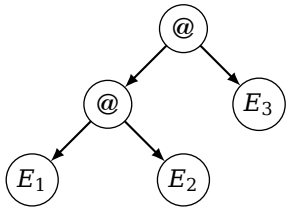
A λ-expression is a complex object, hard to store in memory while keeping informations on its synctactic structure; it's advisable to choose a representation that's easier to manage. In this work we use a *graph*, manipulating it to obtain the final expression in normal form through *graph reduction* [Wadsworth, 1971; Keller and Hoevel, 1985].

As a starting point, we transform the λ-expression into its *abstract syntax tree* (AST). All constants, variables and built-in functions are *leaves* of the AST; the nodes are roots of subtrees which represent λ-terms, and are labeled with special characters that help discriminate between application and abstraction terms.

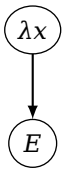
The *application term* $(E_1\,E_2)$ is represented by the @ character:



Functions with multiple arguments explicitly use currying: $(E_1\,E_2\,E_3)$ becomes



The *abstraction term* $(\lambda x\,.\,E)$ is represented by a node that specifies the name of the formal parameter, with the abstraction body as its child:



The syntax tree built according to these rules will be manipulated by an *evaluator* and transformed into a *cyclic graph* through successive reductions. Before going into the detail of those reductions, we shall talk about a fundamental detail of evaluation: what is the next redex to be evaluated, and how to select it. Different strategies will lead to different results, and we want to follow the principles of lazy evaluation that we introduced in chapter 1:

- functions' arguments should be evaluated when *needed*, not when the function is applied;
- arguments should be evaluated once; further uses of the argument within the same function should use the value computed the first time.

This style is also called *call-by-need* evaluation, in contrast with the *call-by-value* employed by eager (non-lazy) languages.

4.2 Selection of the next redex

We already introduced, in chapter 2.5.1, a particular reduction order which guarantees termination if the expression has a normal form: *normal order*. It turns out that this reduction order is quite effective at guaranteeing one of the principles of lazy evaluation[1], namely the fact that arguments should be evaluated only when needed, but – like every other reduction order – it *fully evaluates* the expression until it is in normal form. This behaviour is entirely correct, but sometimes we want to stop evaluation while the expression is only partially evaluated: think of sum-pattern matching, when we want to check that an expression is built using a specific constructor without having to evaluate all its arguments. In other words, we shall fully evaluate *only the top-level redex* and stop at that, even if there may be unevaluated inner redexes. An expression in which the top level has been fully evaluated is said to be in *weak head normal form*[2].

[1] It's not the only reduction order that guarantees lazy evaluation: another one, described in [H. P. Barendregt et al., 1987] and named *innermost spine reduction*, is equally effective, but it's harder to implement; for this reason, we'll focus on normal order.

[2] There also exists a *head normal form*, which is largely equivalent to WHNF except that an expression in HNF cannot have a λ-abstraction as the first argument of the top-level.

Definition 4.1 (Weak head normal form)**.** A λ-expression is said to be in *weak head normal form* (WHNF) if and only if it is of the form

$$F\,E_1\,E_2\,\ldots\,E_n$$

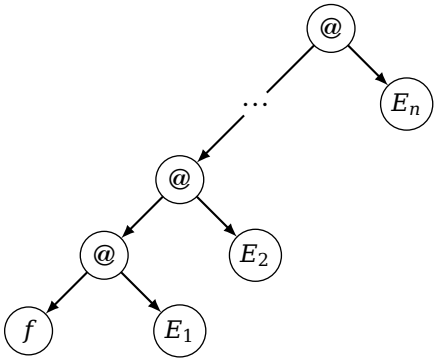
where $n \geq 0$ and F is either:

- a variable
- a data object (a number, a *CONS*, ...)
- a λ-abstraction (or built-in function), and $F\,E_1\,E_2\,\ldots\,E_m$ is not a redex $\forall m \leq n$ (i.e. it's a function applied to too few arguments). ■

We can think of reduction as a two-step procedure: first we reduce all the top-level redexes, bringing the expression in WHNF, then we reduce all the inner redexes to obtain the normal form.

4.2.1 Finding the next top-level redex

Our λ-expressions all have the same general form $f\,E_1\,E_2\,\ldots\,E_n$, i.e. a head f followed by n expressions E_i, represented by the following graph:



There are various possibilities:

- f is a variable name; since f is a top-level, this variable occurs free in the entire expression. Since with our translation it's impossible to generate a non-bound variable, this is an error and we should report it;
- f is a data object (a number, a boolean, a *CONS*...). The expression is in WHNF and the evaluation ends, provided that the number of arguments n is zero; else, the data object is improperly applied to some arguments, and this is a type error[3];

[3] In fact, if the program is properly type-checked, this never happens.

- f is a built-in function taking k arguments. If $n < k$, the expression is in WHNF and the evaluation ends, else the next redex selected by normal order is $f\,E_1\,E_2\,\ldots\,E_k$;
- f is a λ-abstraction. If there are no arguments ($n = 0$), the expression is in WHNF and the evaluation ends, else the next redex selected by normal order is $f\,E_1$ (λ-abstractions are always applied one argument at a time).

It's easy to see that the simplest way to find the next redex is to descend the left branch of the graph, stopping when a function is encountered. Then, we try to go back up m nodes, where m is dependant on the nature of f ($m = 1$ if f is a λ-abstraction, $m = k$ if it's a built-in function taking k arguments); if we encounter the root of the entire graph before the end of the climb, the expression is in weak head normal form and we stop.

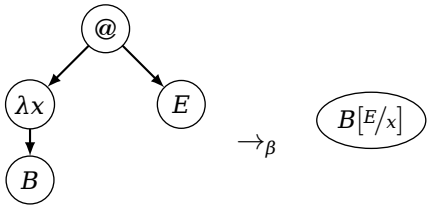
4.3 Graph reduction of λ-expressions

Having found the next redex, we can finally begin the proper reduction: a *local transformation* of the graph (at this time, an abstract syntax tree) we built in section 4.1, which successively modifies the graph itself until it reaches the final result of the computation. We've seen in the previous section that the expression, not being in WHNF, has either a λ-abstraction or a built-in function as its top-level: we will handle the two cases separately.

4.3.1 Reducing the application of λ-abstractions

If the top-level is a λ-abstraction and there's at least an application node in the graph, we can perform a β-reduction:

$(\lambda x.B)\,E \rightarrow_\beta B[E/x]$



Operationally, we must *instantiate* (i.e. create a new copy of) the body of the abstraction B, substituting the occurrences of the formal parameter x with the argument E. This operation may be tricky, so we'll follow three principles:

- since the argument E can be big and/or include further redexes, instead of copying it into the nodes previously occupied by the formal

parameter x we will substitute *pointers* to it. In this way we reduce the complexity and assure that any redex in E is evaluated only once;

- the redex node may be shared, so we must put the result in that same node (physically overwriting the previous content);
- the λ-abstraction may be shared with other nodes, so we should create a new copy of the body and then use this new copy as the recipient for the substitution. Note that this doesn't contradict the previous point: if the redex is the λ-abstraction, we "detach" the abstraction node and "attach" a copy of its body, then we substitute the argument to the parameter in the copy overwriting the node.

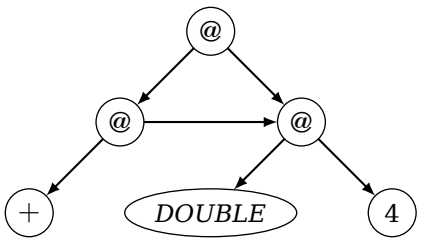
An implementation that follows these principles is called *lazy graph reduction*.

Substituting pointers

When substituting the argument for the formal parameter, we can simply *copy* the argument wherever the formal parameter occurs in the body. This is the simplest way, and a reduction done in this way is called *tree reduction*; however, there are important disadvantages in this approach:

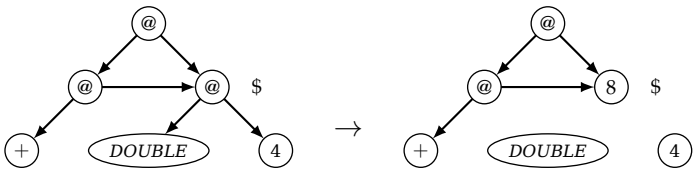
- if the argument is a very large expression and/or there are many occurrences of the formal parameter, we waste a large amount of space making multiple copies of the same object;
- if the argument contains redexes, we do unneeded work duplicating them and having to reduce them separately; this is also against the principle of lazy evaluation.

Both of these problems are solved if we instead substitute any formal parameter occurrence with a *pointer* to the argument; this effectively transforms the abstract syntax tree into a proper graph, since many pointers can point to the same node (*sharing*). For example, the term
$+\,(DOUBLE\,4)\,(DOUBLE\,4)$ is represented in this way:



Overwriting the root of the redex

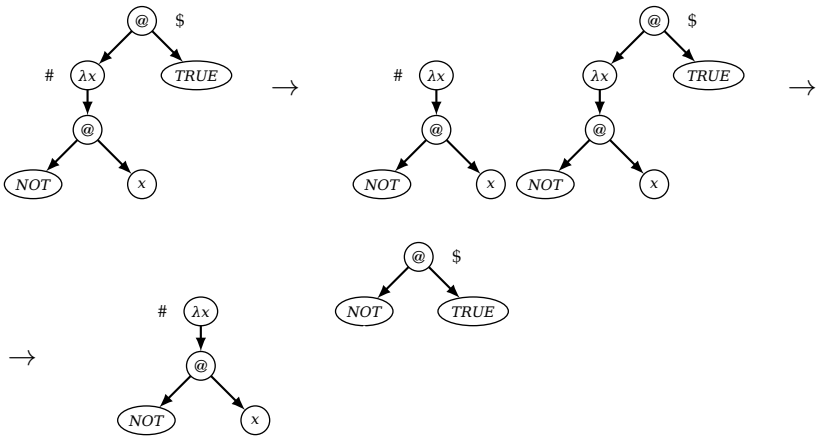
To properly utilize sharing we must ensure that, when an expression is reduced, the graph is modified to reflect the result; in this way, every expression is evaluated at most once. To achieve this, we simply overwrite the root of the redex with (the root of) the result: in this way we assure that every pointer that pointed to the expression now points to the result. For example, in the previous figure the next redex to reduce is *DOUBLE* 4 (since $+$ requires its arguments to be evaluated), whose root is marked by a \$, and after the evaluation is completed the result is written into the same node.



Note that other parts of the redex (in this case, the nodes *DOUBLE* and 4) are not affected by the overwriting; they are simply detached from this part of the graph, but we cannot reuse them since we don't know if they are shared. We assume that there's a garbage collector that removes unreferenced nodes, so we won't draw them anymore.

Instantiating the body

As discussed before, when dealing with a λ-abstraction we should make a new copy of the abstraction body and apply the argument's substitution to the copy; this is done in case the abstraction is shared between more than one expression. This makes the λ-abstraction node a sort of "template" from which a new instance is constructed every time it is applied. An expression like $(\lambda x\,.\,NOT\,x)\,TRUE$ is reduced as follows:



Here, the original λ-abstraction node is tagged with # to emphasize that what gets reduced is *its copy*.

To better describe the instantiation process, we'll introduce a recursive function $\mathit{Instantiate}(\mathit{Body}, \mathit{Variable}, \mathit{Value})$, which *copies Body* substituting *Value* for the free occurrences of *Variable*:

$$\mathit{Instantiate}(\mathit{Body}, \mathit{Variable}, \mathit{Value}) = \mathit{Body}[{}^{\mathit{Value}}/_{\mathit{Variable}}]$$

The *Instantiate* function is defined by case analysis:

1. if *Body* is a variable x and $\mathit{Variable} = x$, then return *Value* (we're substituting a single occurrence of *Variable*);
2. if *Body* is a variable x and $\mathit{Variable} \neq x$, then return *Body*;
3. if *Body* is a constant (including built-in function names), then return *Body*;
4. if *Body* is an application term $E_1\,E_2$, return the application

   $$(\mathit{Instantiate}(E_1, \mathit{Variable}, \mathit{Value})\ \ \mathit{Instantiate}(E_2, \mathit{Variable}, \mathit{Value}));$$

5. if *Body* is a λ-abstraction $(\lambda x\,.\,E)$ and $\mathit{Variable} = x$, return *Body* (since the new λ-abstraction binds the variable x anew, so no substitution will occur inside it);
6. if *Body* is a λ-abstraction $(\lambda x\,.\,E)$ and $\mathit{Variable} \neq x$, return

   $$\lambda x\,.\,\mathit{Instantiate}(E, \mathit{Variable}, \mathit{Value});$$

   we must instantiate the λ-abstraction in case there are occurrences of *Variable* inside it.

This *Instantiate* function is pretty efficient, but there's an hidden risk: if there isn't any free occurrence of *Variable* in *Body*, we may end up doing unnecessary work (see cases 4 and 6) and copying expressions when we could share them. Checking for a new clause at the beginning of the *Instantiate* process could suffice:

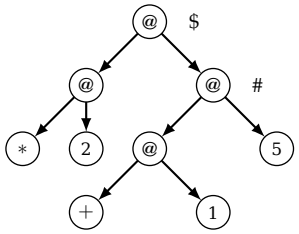
0. if *Body* does not contain any free occurrence of *Variable*, return *Body*.

It turns out that this check is quite expensive to make; an implementation that does it (or something to the same effect) is called *fully lazy*. In the following chapter we'll see a way to obtain the same result in a different way.

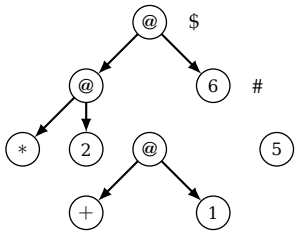
4.3.2 Reducing the application of built-in functions

If the redex is a built-in function and there are enough arguments, we should first evaluate the necessary arguments (which ones are *necessary* depends from the function itself), calling the evaluator recursively; then we can apply the built-in function, and overwrite the root with the result. The following example shows the reduction of a built-in function application.

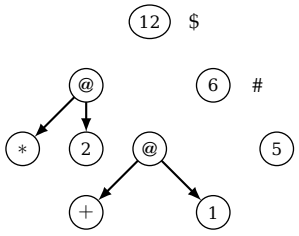
Example 4.1. Consider the expression $* \; 2 \; (+ \; 1 \; 5)$, represented by the graph



The evaluator first goes down the left side of the graph, encountering $*$ as the next function to apply; then it goes back up and selects $ as the root of the redex. The built-in function $*$ requires that its arguments are evaluated; the evaluator recursively calls itself on the left application node, only to discover that the first argument is already in WHNF, then calls itself on the right application node (marked with #). This is a redex, so again it calls itself on the two arguments of $+$ (which are in WHNF) and then applies it:



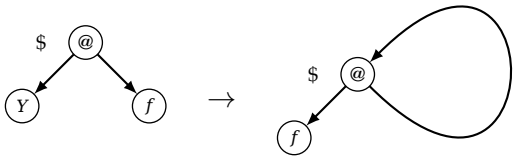
Now both arguments of the $*$ function are fully evaluated, so we can apply the function and obtain the final result:



▲

4.3.3 Implementing the Y function

In section 2.4.1 we said that implementing the Y function as a λ-abstraction is inefficient and that we're better implementing it directly. Here we show a very simple way to do this operation:



This reduction directly corresponds to the reduction rule for Y:
$Yf \rightarrow f\,(Yf)$. This is the only source of cycles for our graphs, and this implementation of Y is thus called *cyclic* Y or *knot-tying* Y. Thanks to it, we can manage recursive functions and infinite data structures without having to allocate a new cell each time the (Yf) redex is evaluated.

5 | Supercombinators and λ-lifting

In the last chapter we introduced an useful representation of a λ-expression in memory, in the form of a graph, and described how to reduce this graph in order to obtain the final result of the evaluation. We saw that the trickiest part, the one that can cause a massive overhead in the process, was the application of a λ-abstraction to an argument: we had to *instantiate* the abstraction body, making a new copy of it through a recursive function called *Instantiate* that analyzed every node and selected its behaviour accordingly.

5.1 Compilation

We come up with a better alternative: what if we could *compile* the abstraction body to a *fixed set of instructions* of the target language? The instantiation process will then amount to simply obeying the instructions associated with the body. All the tests previously made by *Instantiate* recursively would now be done just once, when compiling the λ-body, and we could further optimize the compilation through known techniques in this field.

As it turns out, this wonderful idea has one problem: not every λ-abstraction body can be compiled. For example, consider the expression

$$\lambda x\,.\,(\lambda y\,.\,+\,x\,y),$$

whose body is $(\lambda y\,.\,+\,x\,y)$. When applying the whole abstraction to a single argument, say 1, the body gets instantiated as $(\lambda y\,.\,+\,1\,y)$; change the argument, and the instantiated body changes too – it's clearly impossible to compile a *single* fixed sequence for this λ-abstraction. The problem is that x occurs free in the body of the λy abstraction, so we have to make a *new* instance of the latter wherever x gets bound to a different value by the λx abstraction; if we had no free variables in the body we could compile it without hassle.

There are two approaches to solving this problem. We could give an *environment* to the compiled code, allowing it to access the values of free variables in the body (*parameterizing* the compiled code on them); this is the

route followed by all the block-structured languages (like C, Java, ...). Alternatively, we could *transform* the program into an equivalent one in which all λ-abstractions are compilable; this approach does not require an environment and it's called *supercombinator graph reduction*, while the transformation algorithm is called *λ-lifting*.

5.2 Supercombinators

In section 2.2 we introduced *combinators* as λ-terms without free variables. We now introduce a subset of combinators, called *supercombinators*.

Definition 5.1 (Supercombinators)**.** A λ-expression of the form

$$\lambda x_1 . \lambda x_2 . \ \ldots \ \lambda x_n . E,$$

where E is not a λ-abstraction, is called a *supercombinator* $\$S$ of arity n if and only if

- $\$S$ has no free variables;
- $n \geq 0$ (there need be no λs at all);
- any λ-abstraction is E is a supercombinator. ■

Definition 5.2 (Supercombinator redexes and reduction)**.** A *supercombination redex* is the application of a supercombinator of arity n to exactly n arguments. *Supercombinator reduction* is the act of replacing a supercombinator redex by an instance of the supercombinator body, with the arguments substituted for free occurrences of the corresponding formal parameter. ■

Supercombinators of arity 0 are called *constant applicative forms* or CAFs. Since they don't have any λ-abstractions, they are never instantiated and can be safely shared in the graph; they don't need to be compiled. Supercombinators of arity $n > 0$ will be our unit of compilation: they have no free variables, so we can compile a fixed code sequence for them, and any abstraction in their body will be a supercombinator, thus having no free variables too and not requiring a copy when instantiating the supercombinator body.

5.2.1 A smarter β-reduction

Why are supercombinators amenable to compilation? The answer is that they can be the subject of a "multi-argument" reduction when they are applied to the right amount of parameters: take the term of the previous example $\lambda x . (\lambda y . + x y)$, applying it to two arguments (say 1 and 2). The normal

reduction introduced in section 2.3.2 would proceed like this:

$$\begin{aligned} & (\lambda x\,.\,(\lambda y\,.\,+\,x\,y))\,1\,2 \\ \rightarrow_\beta\ & (\lambda y\,.\,+\,1\,y)\,2 \\ \rightarrow_\beta\ & +\,1\,2. \end{aligned}$$

However, there's no reason we couldn't apply both arguments at once, performing the λx and λy reduction simultaneously: the result of performing the λx abstraction alone is a λy abstraction, and no further work can be done on it until it is given the second argument, hence we could wait until both arguments are present and then perform both reductions at once:

$$\begin{aligned} & (\lambda x\,.\,(\lambda y\,.\,+\,x\,y))\,1\,2 \\ \rightarrow\ & +\,1\,2. \end{aligned}$$

This multi-argument reduction is possible only when applied to supercombinators; in fact, this is simply what we defined as supercombinator reduction.

In the rest of the chapter, we will name supercombinators prepending an \$ to their name: moreover, we'll use a special notation to emphasize their special status. The supercombinator $\lambda x\,.\,(\lambda y\,.\,+\,x\,y)$ then becomes

$$\$SUP\,x\,y = +\,x\,y$$

where $\$SUP$ is an arbitrary name. Our strategy will be to transform the λ-expression we wish to compile into a *set* of supercombinator definitions plus a (simpler) expression to be evaluated:

$\$SUP\,x\,y = +\,x\,y$
$\$SUP\,1\,2$

is how we translate the term $(\lambda x\,.\,(\lambda y\,.\,+\,x\,y))\,1\,2$. Since supercombinator reduction occurs only when all arguments are available, we can regard the supercombinator definitions as a set of *rewrite rules*: a reduction consists of rewriting an expression which matches the left-hand side of a rule with an instance of the corresponding right-hand side. These kinds of systems are called *term rewriting systems* [TERESE, 2001].

5.2.2 λ-lifting

We can now describe an algorithm that transforms λ-abstractions into supercombinators.

Algorithm 1 λ-lifting

1: **while** there are untranslated λ-abstractions **do**
2: choose a λ-abstraction a which has no inner λ-abstractions in its body
3: perform β-abstraction on a to take out all its free variables as extra parameters
4: give an arbitrary name to the a, like $\$NAME$
5: replace the occurrence of a with the name applied to the free variables
6: compile the abstraction body and associate the name with the compiled code
7: **end while**

Example 5.1. Let's try to translate $(\lambda x\,.\,(\lambda y\,.\,+\,y\,x)\,x)\,4$ with the λ-lifting algorithm. The starting point is

$(\lambda x\,.\,(\lambda y\,.\,+\,y\,x)\,x)\,4$

The first step is selecting a λ-abstraction which doesn't have further λ-abstractions in its body. The only one here that meets the requirement is $(\lambda y\,.\,+\,y\,x)$, which has x as a free variable. The second step is to "bring out" the x as an extra parameter:

$$(\lambda y\,.\,+\,y\,x) \quad \leftarrow_{\beta} \quad (\lambda x\,.\,\lambda y\,.\,+\,y\,x)\,x \quad \underset{\alpha}{\longleftrightarrow} \quad (\lambda w\,.\,\lambda y\,.\,+\,y\,w)\,x.$$

(We used α-conversion to give a different name to the parameter, to avoid confusion.) The expression $(\lambda w\,.\,\lambda y\,.\,+\,y\,w)$ is now a supercombinator, and we can substitute it into the original term:

$(\lambda x\,.\,(\lambda w\,.\,\lambda y\,.\,+\,y\,w)\,x\,x)\,4$

The third and fourth steps require us to give a name to this new supercombinator and substitute it to the occurrence in the expression:

$\$Y\,w\,y = +\,y\,w$
$(\lambda x\,.\,\$Y\,x\,x)\,4$

The fifth step is to compile the body of $\$Y$. This will be discussed later, but for now let's imagine having a very simple C-like procedure:

```
compiled-Y(w,y) {
  return (y + w);
}
```

Wherever $\$Y$ is applied to two arguments, the evaluation will simply be the execution of the procedure `compiled-Y`. One cycle of the λ-lifting argument is now done. Next, we select $\lambda x\,.\,\$Y\,x\,x$ as the next λ-abstraction: it doesn't have any free variable, so it's already a supercombinator. We give a name to this supercombinator and substitute it in the expression:

$\$Y\,w\,y = +\,y\,w$ $\$X\,x = \$Y\,x\,x$
$\$X\,4$

The compiled code for $\$X$ is:

```
compiled-X(x) {
  return compiled-Y(x,x);
}
```

Finally, we notice that $\$X\,4$ is a supercombinator too (of arity zero). We perform a final λ-lifting cycle:

$\$Y\,w\,y = +\,y\,w$ $\$X\,x = \$Y\,x\,x$ $\$Program = \$X\,4$
$\$Program$

where the compiled code for the supercombinator $\$Program$ is simply

```
compiled-Program() {
  return compiled-X(4);
}
```

When the evaluator has to evaluate the λ-term $\$Program$, it will simply call the `compiled-Program()` procedure, which will return 8. ▲

5.2.3 Eliminating redundant parameters

While executing the λ-lifting algorithm, redundant definitions or definitions with redundant parameters may show up. Redundant definitions are of the form $\$A = \$B$; we can eliminate them by using directly $\$B$ wherever $\$A$ appears. Redundant parameters show up in definitions of the form $\$C\,b = \$D\,a\,b$, and are easily taken care of through η-reduction: $\$C = \$D\,a$. Note that this last optimization is actually undesirable when using some sophisticated implementations.

5.2.4 Parameter ordering

When we take out several free variables from a single λ-abstraction, the order in which we put them can be actually important. For example, the expression $(\lambda x\,.\,\lambda z\,.\,+\,y\,(*\,x\,z))$ can be translated in two seemingly equivalent ways:

$\$S\,x\,y\,z = +\,y\,(*\,x\,z)$	$\$S\,y\,x\,z = +\,y\,(*\,x\,z)$
$(\lambda x\,.\,\$S\,x\,y)$	$(\lambda x\,.\,\$S\,y\,x)$

Since the $\$S$ supercombinator "takes care" of putting the parameters in the right order in the combined code, we may think that the two forms are indeed equivalent. If we continue with the algorithm, lifting the λx abstraction, we see an important difference:

$\$S\,x\,y\,z = +\,y\,(*\,x\,z)$ $\$T\,y\,x = \$S\,x\,y$	$\$S\,y\,x\,z = +\,y\,(*\,x\,z)$ $\$T\,y\,x = \$S\,y\,x$
$(\$T\,y)$	$(\$T\,y)$

Both forms of the $\$S$ supercombinator lead to the same result, the introduction of a new supercombinator $\$T$, but the right one can be further reduced: through two η-conversions we obtain $\$T = \$S$, so $\$T$ is actually redundant and we can write the bottom expression of the right-hand box simply as $(\$S\,y)$. We can't do the same with the left-hand box because $\$T\,y\,x$ and $\$S\,x\,y$ are not η-convertible, so it seems that there's a 'preferred' order: we should put the variables bound at inner levels last. We could *number* every abstraction, starting from 0 at the top-level and increasing with every layer of abstraction, and then order the variables according to the number of the abstraction which generated them.

Definition 5.3 (de Bruijn numbers)**.** The *de Bruijn number* (or *lexical level-number*) of a λ-abstraction is the number of textually enclosing λs plus 1. ■

For example, in $\lambda x\,.\,\lambda y\,.\,+\,x\,y$ the λx abstraction has de Bruijn number 1 while the λy abstraction has de Bruijn number 2 (is enclosed by the λx). The lexical level of a variable is then equal to the de Bruijn number of the λ-abstraction which binds it, with constants, built-in functions and previously-defined supercombinators regarded as being bound at the top-level and having thus lexical level 0. Every time we generate an extra parameter (second step of the λ-lifting algorithm) we simply have to put it into a position compatible with its level; in this way we maximize the chances of being able to apply η-reduction to supercombinator definitions.

5.2.5 The case for recursive supercombinators

In section 3.3.5 we proposed a simple way to treat recursive definitions: we pack mutually recursive definitions into a single tuple, then use the Y function to eliminate recursion and obtain a simple *let*. This is quite inefficient: every time we have to build a tuple just to have it taken apart slightly later. Moreover, with supercombinators we introduced a way to actually *name* λ-abstractions, so they can refer to themselves.

To obtain better performances we avoid using the Y function completely, having a set of mutually recursive supercombinator definitions in its place. Obviously, when translating the Haskell code we shouldn't actually fully reduce the enriched λ-calculus down to the applied λ-calculus; we want to stop when all the definitions are expressed with simple *let*(*rec*) constructs (in other words, we don't want to translate simple *letrec*s into simple *let*s, nor we want to translate simple *let*s into λ-abstractions). As a final note, we observe that the supercombinator notation

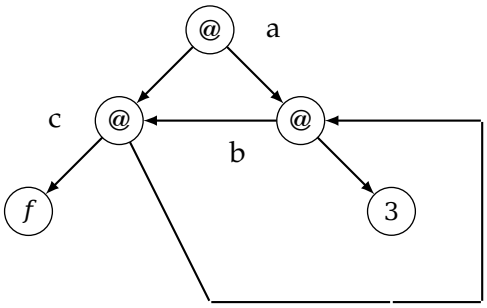
$$\begin{array}{|c|}\hline \$S1\ x\ y = B1 \\ \$S2\ f = B2 \\ \ldots \\ \hline E \\ \hline \end{array}$$

is equivalent to the *letrec* notation:

$$\begin{array}{l} letrec \\ \quad \$S1 = \lambda x\,.\,\lambda y\,.\,B1 \\ \quad \$S2 = \lambda f\,.\,B2 \\ \quad \ldots \\ in \\ \quad E \end{array}$$

5.2.6 *let*(*rec*)s in supercombinator bodies

In the previous chapter we introduced graph reduction: starting from the tree representation of a λ-expression, we used a set of rules to transform that tree into a cyclic, directed graph with node sharing like this one:



What's the textual representation for this graph? The trick is to name nodes, as we did here, and express the graph thus:

$$a = c\,b$$
$$b = c\,3$$
$$c = f\,b$$

with a as the root node. This is, in fact, just a *letrec*:

$$\begin{aligned} &\mathit{letrec} \\ &\quad a = c\,b \\ &\quad b = c\,3 \\ &\quad c = f\,b \\ &\mathit{in} \\ &\quad a. \end{aligned}$$

We just gave another definition for the *letrec* construct: it's a textual description of a graph. If a supercombinator has a *letrec* inside its body, then its body is a graph; these kinds of supercombinators are said to have *graphical bodies*. In a similar way, the bodies of supercombinators can contain *let*-expression: in this case, the represented graph is acyclic.

A supercombinator with a graphical body has an important property: its instantiation is simply a matter of constructing the graph associated with the body, which can then be reduced using the rules we've seen in the previous chapter. This is precisely the kind of *compilation* we missed in previous sections; other types of supercombinator compilation can be found in [Keller and Hoevel, 1985] and [S. Peyton Jones, 1992], the latter being the strategy used by the Haskell language.

5.2.7 λ-lifting and *letrec*s

The good news is that the λ-lifting doesn't need any modification to work with recursive supercombinators: *letrec*s are treated just like any other expression, and the actual lifting still applies only to λ-abstractions. We

need only to take care of which lexical level-number to assign to variables bound in a *letrec*: since they will be instantiated when the *immediately enclosing λ-abstraction* is applied to an argument (which is also the time when the *letrec* is instantiated), it makes sense to assign the same lexical level-number as the immediately enclosing abstraction.

If there's no enclosing λ-abstractions, the definition bodies of the *letrec* cannot have any free variable other than the ones defined in the same *letrec*: in other words, they're combinators, and we can λ-lift them to remove inner λs and turn them into supercombinators. For example, the program that computes the infinite list composed only by 1s is a *letrec*

$letrec\ x = CONS\ 1\ x\ in\ x$

that can be lifted to

$\$x = CONS\ 1\ \$x$
$\$x$

5.3 Fully lazy λ-lifting

There's a further, important optimization that we can do to our implementation of λ-lifting to make it even *lazier* than the one we described earlier. We said in section 4.3.1 that, when instantiating an abstraction, we risk copying the same constant expression over and over rather than sharing a single copy of it: this is needlessly time- and space-consuming, but it's hard to spot these cases even with compilation given that those sharable expressions can be generated during reduction.

Example 5.2. Let's try to evaluate the expression

$$\begin{aligned} &letrec\ f = g\ 4 \\ &\qquad g = \lambda x\,.\,\lambda y\,.\ +\ y\,(sqrt\ x) \\ &\quad in\ +\ (f1)\,(f2). \end{aligned}$$

In the following reduction we use the character # to represent a pointer to a shared expression.

$$
\begin{array}{ll}
\quad + \ (f\,1)\ (f\,2) & \\
\rightarrow + \ (\#\,1)\ (\#\,2) & \#:(\lambda x\,.\,\lambda y\,.\,+\ y\,(sqrt\,x))\,4 \\
\rightarrow + \ (\#\,1)\ (\#\,2) & \#:(\lambda y\,.\,+\ y\,(sqrt\,4)) \\
\rightarrow + \ (\#\,1)\ (+\,2\ \underline{(sqrt\,4)}) & \#:(\lambda y\,.\,+\ y\,(sqrt\,4)) \\
\rightarrow + \ (\#\,1)\,4 & \#:(\lambda y\,.\,+\ y\,(sqrt\,4)) \\
\rightarrow + \ (+\,1\ \underline{(sqrt\,4)})\,4 & \\
\rightarrow + \ 3\,4 & \\
\rightarrow 7. &
\end{array}
$$

We can clearly see that $sqrt\,4$ (here underlined) is evaluated twice, since a new instance of it is made each time the λy is applied; $sqrt\,4$ is a *dynamically created* constant subexpression of the abstraction body. Translating the *letrec* into a supercombinator won't suffice:

$$
\begin{array}{|c|}
\hline
\$g\,x\,y = +\,y\,(sqrt\,x) \\
\$f = g\,4 \\
\$Prog = +\,(\$f\,1)\,(\$f\,2) \\
\hline
\$Prog \\
\hline
\end{array}
$$

$$
\begin{array}{ll}
\quad \$Prog & \\
\rightarrow + \ (\#\,1)\ (\#\,2) & \#:\$g\,4 \\
\rightarrow + \ (\#\,1)\ (+\,2\ \underline{(sqrt\,4)}) & \#:\$g\,4 \\
\rightarrow + \ (\#\,1)\,4 & \#:\$g\,4 \\
\rightarrow + \ (+\,1\ \underline{(sqrt\,4)})\,4 & \\
\rightarrow + \ 3\,4 & \\
\rightarrow 7 &
\end{array}
$$

▲

To be as lazy as possible we want to share even this dynamically created constant expressions: we want to evaluate every expression *at most once*, to obtain what is called *full laziness*.

5.3.1 Maximal Free Expressions

We discovered that instantiating "too much" of a λ-expression can make us lose full laziness; we shouldn't instantiate those subexpressions which

contains no occurrences of the parameter, because if the parameter doesn't occur then the value of the subexpression is the same throughout all instances.

Definition 5.4 (Proper subexpressions)**.** An expression E is a *proper subexpression* of a λ-abstraction F if and only if E is a subexpression of F and $E \neq F$. ■

Definition 5.5 (Maximal free expressions)**.** A subexpression E of a λ-abstraction L is *free* in L if all variables in E are free in L. A *maximal free expression* (MFE) of L is a free expression which is not a proper subexpression of any other free expression of L. ■

An MFE is exactly what can be shared throughout multiple instances of a λ-abstraction in order to retain full laziness. When performing β-reduction we must not instantiate an MFE: we just substitute pointers to the single shared instance in the body of the abstraction.

Example 5.3. Without reducing MFEs, the evaluation of the expression of the previous example goes like this. The first steps are the same:

$$
\begin{array}{ll}
\quad + \ (f\,1)\,(f\,2) & \\
\rightarrow + \ (\#\,1)\,(\#\,2) & \#\,:(\lambda x\,.\,\lambda y\,.\,+\,y\,(sqrt\,x))\,4 \\
\rightarrow + \ (\#\,1)\,(\#\,2) & \#\,:(\lambda y\,.\,+\,y\,(sqrt\,4)).
\end{array}
$$

Now the evaluator recognizes $(sqrt\,4)$ as a MFE and substitutes a pointer to the abstraction body:

$$
\begin{array}{lll}
\rightarrow + \ (\#\,1)\,(+\,2\,\#_1) & \#\,:(\lambda y\,.\,+\,y\,\#_1), & \#_1\,:\,(sqrt\,4) \\
\rightarrow + \ (\#\,1)\,(+\,2\,\#_1) & \#\,:(\lambda y\,.\,+\,y\,\#_1), & \#_1\,:\,2 \\
\rightarrow + \ (\#\,1)\,4 & \#\,:(\lambda y\,.\,+\,y\,\#_1), & \#_1\,:\,2 \\
\rightarrow + \ (+\,1\,\#_1)\,4 & \#_1\,:2 & \\
\rightarrow + \ 3\,4 & & \\
\rightarrow 7 & &
\end{array}
$$

hereby evaluating $(sqrt\,4)$ just once. ▲

5.3.2 MFEs and λ-lifting

Identifying maximal free expressions dynamically is rather difficult to do efficiently. [Hughes, 1983] proposed a slight modification to the λ-lifting algorithm that suffices to preserve full laziness:

- instead of abstracting out free variables of a λ-abstraction (step 2 of the algorithm, section 5.2.2), abstract out *entire maximal free expressions* as extra parameters.

If this step is performed, we call the resulting algorithm *fully lazy λ-lifting*. In the previous example, instead of abstracting $(\lambda y\,.\,+\,y\,(sqrt\,x))$ in the normal way (which would have led us to $g\,x\,y = +\,y\,(sqrt\,x)$) we bring out the *entire* MFE $(sqrt\,x)$ as an extra parameter:

$$\$g1\;e\;y = +\,y\,e$$
$$\$g\;x = \$g1\;(sqrt\,x)$$

Fully lazy λ-lifting slightly increases the complexity of the final expression (there are more supercombinator definitions than before), but the trade-off is a fully lazy evaluation. A slight optimization that can be made is recognizing if an expression has *no* free variables (a CAF): in that case, instead of abstracting it out as an extra parameter, it can be given a name and made into a supercombinator.

When dealing with recursive expressions, the same problem arises with MFEs defined inside *letrec*s: they will not be lifted out as free expressions. The solution lies in reworking the *let*(*rec*) definitions by "floating" out them as far as possible: a variable x bound in a *let*(*rec*) will (generally) depend on the value of certain free variables (x's *free variable set*), so we can float the definition of x outwards until the next enclosing λ-abstraction binds one of the variables in its free variable set. Any definition with an empty free variable set will be lifted to the top-level, where it will be turned into a supercombinator directly ([S. L. Peyton Jones and Lester, 1991]).

6 | Conclusions

This short work has been useful for me to understand the inner workings of a computation paradigm that's quite different from those I've studied throughout the three-year undergraduate course. While I came to realize that, due to the overhead that it introduces, lazy evaluation is seldom necessary, I think that there are many application fields that would surely benefit from its use.

6.1 Future works

A lot has been left out in this work for the sake of simplicity and to focus on proper lazy evaluation: we said next to nothing on types and type checking, gave no attention to list comprehensions (a special syntax for lists in Haskell), and completely ignored advanced Haskell features like monads and type classes; since most production-level software written in this language (like Facebook's Sigma) make use of these features, it would be helpful to show how they are handled.

For almost all topics there's some sort of further optimization to be made; for example, our translation for product-types make use of different Eval-*t*-i functions for every type *t*, which is prohibitively complex when programmers can define their own types; we could do better by adding a *structure tag*, identical for all types, which can be used by the Eval-i functions to select the *i*-th argument regardless of the actual type. Other improvements can be made using dependency analysis to remove unneeded *letrec*s (see sec. 3.4), or eliminating redundant full laziness after an application of the lazy λ-lifting algorithm.

We didn't discuss possible alternative approaches to supercombinator graph reduction; [S. L. Peyton Jones, 1987] has an interesting digression on the use of the SK supercombinators. Moreover, the actual implementation of supercombinator graph reduction has not been discussed at all; it can be made using a high-level language like C and Java to be fully portable, or one can choose a lower-level implementation to obtain better performance.

These points can all be addressed in a future work that aims for completeness.

Index

Bibliography

Abramsky, Samson and Achim Jung (1994). "Domain Theory". In: *Handbook of Logic in Computer Science (Vol. 3)*. Ed. by Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum. Oxford University Press.

Barendregt, H. (1984). *The Lambda Calculus: Its Syntax and Semantics, 2nd Edition*. Studies in Logic and the Foundations of Mathematics. Elsevier Science.

Barendregt, H. P. et al. (1987). "Needed Reduction and Spine Strategies for the Lambda Calculus". In: *Information and Computation* 75.3.

Church, A. (1941). *The Calculi of Lambda-conversion*. Annals of mathematics studies. Princeton University Press.

Curry, H.B. and R. Feys (1958). *Combinatory Logic*. Combinatory Logic vv. 1 and 2. North-Holland Publishing Company.

Hughes, John (1983). *The Design and Implementation of Programming Languages*. Tech. rep. PRG40. OUCL.

Keller, R M and L W Hoevel (1985). "Distributed Computation by Graph Reduction". In: *Systems Research* 2.4.

Peyton Jones, Simon et al. (1992). "Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.2". In: *SIGPLAN Notices* 27.5.

— (2007). "A History of Haskell: Being Lazy with Class". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM.

Peyton Jones, Simon L. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Inc.

Peyton Jones, Simon L. and David Lester (1991). "A Modular Fully-lazy Lambda Lifter in Haskell". In: *Software Practice and Experience* 21.5.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press.

Schönfinkel, M. (1924). "Ü ber die Bausteine der mathematischen Logik". German. In: *Mathematische Annalen* 92.3-4. ISSN: 0025-5831.

Scott, Dana (1982). "Lectures on a Mathematical Theory of Computation". In: *Theoretical Foundations of Programming Methodology*. Ed. by Manfred

Broy and Gunther Schmidt. Vol. 91. NATO Advanced Study Institutes Series. Springer Netherlands.

Stoy, Joseph E. (1981). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press.

Tennent, R. D. (1994). "Denotational Semantics". In: *Handbook of Logic in Computer Science (Vol. 3)*. Ed. by Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum. Oxford University Press.

TERESE (2001). *Term Rewriting Systems*. Ed. by J. W. Klop, Marc Bezem, and R. C. De Vrijer. Cambridge University Press.

Turing, Alan Mathison (1936). "On computable numbers, with an application to the Entscheidungsproblem". In: *Journal of Math* 58.345-363.

Wadsworth, C.P. (1971). *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford.