UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Energy Consumption of Functional Programs in the Context of Lazy Evaluation

Gilberto Amaral Cordeiro Melfe

Tese para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Professor Doutor Simão Melo de Sousa
Co-orientadores: Professores Doutores João Paulo Fernandes, Fernando Castor

Covilhã, Outubro de 2016

Dedication

To all that contributed...

Acknowledgments

I would like to thank to all that have, directly or indirectly, helped with the development of this work. In particular I would like to thank my adviser, for the extreme availability, and patience, to listen to all my doubts/problems related to this work, and unrelated to the work. Without him I would never have finished this work. A word of appreciation goes also to the co-advisers, brazillian colleagues co-authors of a conference paper (Luís, Francisco, Paulo) and all colleagues at the Release Group who kindly welcomed me into their laboratory.

Resumo

O planeta Terra dispõe de recursos naturais limitados disponíveis para suportar o nosso quotidiano, sejam eles matérias primas para manufactura ou energia para gerar trabalho. O ritmo a que consumimos esse recursos está a aproximar-se dos limites dentro dos quais a natureza pode restabelecê-los, e a que nós podemos extraí-los.
É com esses recursos que desenvolvemos a mais variada tecnologia, da qual o nosso modo de vida moderno é cada vez mais dependente, para providenciar todos os tipos de serviços imagináveis. Em particular, as Tecnologias de Informação e Comunicação (TIC) são uma parte essencial da vida de hoje. Com cada vez mais dispositivos, suportando diferentes serviços, em utilização, o seu consumo de energia cresce diariamente.
Cientes deste factos, os desenvolvedores de hardware/software procuram modos de optimizar o consumo de energia dos artefactos computationais (hardware/software).
O nosso trabalho, focado no software, foi motivado pela necessidade de apurar se, e até que ponto, podemos poupar energia adaptando programas existentes.
Nessa medida, implementámos um benchmark que foi utilizado para analisar o consumo energético de várias implementações de abstracções de estruturas de dados comuns, implementadas na biblioteca `Edison`, para a linguagem de programação `Haskell`.
As nossas descobertas levam-nos a concluir que podemos poupar energia, extensivamente, dependendo do padrão de utilização, por parte dos programas, das operações nativas disponíveis na `Edison`.

Palavras-chave

Eficiência energética, Estruturas de dados puramente funcionais, Edison, Haskell

Resumo alargado

A sociedade moderna tem evoluído a um ritmo admirável. O nosso estilo de vida moderno, faz cada vez mais uso de tecnologias de toda a espécie para nos facilitar a vida. Como exemplos temos, a conectividade global, providenciada pela Internet, a facilidade de viajar por todo o mundo, etc.. Esse mesmo, desejado, estilo de vida leva-nos a querer consumir mais produtos e serviços.

No entanto, temos recursos naturais limitados disponíveis para suportar o nosso quotidiano, sejam eles matérias primas para manufactura ou energia para gerar trabalho. Por outro lado, o ritmo a que consumimos esse recursos está a aproximar-se dos limites dentro dos quais a natureza pode restaurá-los, e a que nós podemos extraí-los.

Em particular, as Tecnologias de Informação e Comunicação (TIC) são uma parte essencial da vida de hoje. De modo crescente, as nossas actividades diárias fazem uso de todo o tipo de dispositivos como smartphones ou outros "computadores". Com cada vez mais dispositivos, suportando diferentes serviços, em utilização, o seu consumo de energia cresce diariamente. Esta realidade verifica-se por exemplo no crescente consumo energético do largo número de data centers que suportam serviços online.

No contexto das TIC a preocupação em/necessidade de, utilizar os recursos criteriosamente é reconhecida à bastante tempo, tendo tido o foco primeiramente no hardware, e mais recentemente no software. Os desenvolvedores de hardware/software procuram assim, modos de optimizar o consumo de recursos, principalmente de energia, dos artefactos computationais (hardware/software). De facto existem estudos que prevêem poupanças de 30% a 90% de energia consumida por dispositivos de hardware a executar software optimizado.

O nosso trabalho foi motivado pela necessidade de apurar se, e até que ponto, podemos poupar energia adaptando programas existentes.

Nessa medida, implementámos um benchmark que foi utilizado para analisar o consumo energético de várias implementações de abstracções de estruturas de dados comuns, implementadas na biblioteca `Edison`, para a linguagem de programação `Haskell`.

As nossas descobertas levam-nos a concluir que podemos poupar energia, extensivamente, dependendo do padrão de utilização, por parte dos programas, das operações nativas disponíveis na `Edison`.

Abstract

We have limited natural resources available to support our daily living, be they raw materials for manufacturing or energy to generate work. The pace at which we consume those resources is approaching the limits at which nature can replenish them, and at which we can extract them. It is with those resources that we develop the most varied technology, on which our modern way of life is increasingly more dependent, to provide every kind of service conceivable.
In particular, the Information and Communication Technologies are an essential part of today's living. With ever more devices, supporting different services, in utilization, their energy demand grows daily.
Aware of this facts, hardware/software developers seek ways to optimize the energy consumption by the computing hardware/software artifacts.
Our work, focused on software, was driven by the need to know if, and to what extent, can we save energy by refactoring existing programs.
To that extent, we implemented a benchmark that was used to analyze the energy consumption of various implementations of common data structure abstractions, implemented in the `Edison` library, for the `Haskell` programming language.
Our findings lead us to conclude that, we can save energy, to a great extent, depending on the usage pattern, by software programs, of the native operations available in `Edison`.

Keywords

Energy efficiency, Purely functional data structures, Edison, Haskell

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API	Application Programming Interface
DIMM	Dual In-line Memory Module
DRAM	Dynamic Random Access Memory
FFI	Foreign Function Interface
GHC	Glasgow Haskell Compiler
GPU	Graphics Processing Unit
HNF	Head Normal Form
ICT	Information and Communication Technology
IT	Information Technology
JDK	Java Development Kit
MSR	Model-Specific Register
OLS	Ordinary Least-Squares
OS	Operating System
PKG	Package (power domain)
PP0	Power Plane 0 (power domain)
PP1	Power Plane 1 (power domain)
RAPL	Running Average Power Limit
UBI	Universidade da Beira Interior
WHNF	Weak Head Normal Form

Chapter 1

Introduction

Modern society is evolving at a pace that has seen no parallel in the past. The modern lifestyle includes global connectivity, high consumerism and frequent and easy travelling. This lifestyle, however, implies an immoderate demand for natural resources, which is unsustainable in the long term due to two essential reasons:

- we have but a finite, planet's worth of "supplies", both in terms of physical raw materials and energy supply [Gui14];
- the pace at which resource consumption has been growing is gaining on the pace at which resources can be made available [J.95, SFKW+].

The current modern lifestyle is also highly dependent on Information and Communication Technologies (ICT). Indeed, our everyday activities are more and more dependent on more and more IT devices such as smartphones, tablets or laptops. Indeed, it is estimated that there are currently 2.08 billion smartphone users [Sta14] and that, that number will probably more than double by 2020 [Eri15].
While the use of such devices seeks to increase both comfort and productivity it also implies a significant energy consumption impact [FFMB11, YWJ10, FZ08].
Also, through the use of (essentially) mobile devices, people increasingly access a lot of services, like social networks, entertainment services (games and on-demand video services) and online collaboration platforms (such as Google Docs, for example). These services, in turn, are backed by a growing number of large data centers which also consume a lot of energy [FZ08].
As more and more services become reliant on ICT (like public administration services) we find ourselves more and more dependent on technology to run our daily lives.
In the context of IT, the need to judiciously utilize resources has long been realized. Indeed, it has been estimated that 50% of the overall costs, incurred on by organizations, can be attributed to their IT departments [HA09]. More broadly, according to [VVHC+10] the IT's share of the global energy consumption was about 7% in 2008, and it is predicted that it will double by 2020.
This realization, however, has historically been addressed mainly on the hardware part of IT systems. Indeed, for quite some time hardware manufacturers have been developing their technologies, trying to deliver the same (throughput) performance at lower energy consumptions [CSB92, TMW94, YN03].
More recently, we have started witnessing a trend that tries to analyze and optimize energy consumption with a focus on software. This can be seen e.g., in mobile devices studies [KL10, BBV09, TMOM12, KLGT09] and in general purpose programming languages studies [STM+14, VBB+14, SPC14, PCL14b, LPL15].
This trend is also in line with the software developers interest, which is confirmed by recent studies [PCL14a]. In fact it has been observed that, optimized software could save between 30% to 90% of the energy consumed by devices [Sof15].
In this thesis, we seek to contribute to the improvement of the energy footprint associated to software written in a particular programming language and using concrete programming constructions.

The programming language of focus is Haskell, a declarative, statically and strongly typed, lazy, purely functional language. The constructions we consider are the different realizations of common abstractions such as Sets or Sequences.
The different implementations of the abstractions considered are provided by the Haskell implementation of the Edison library [Oka01].
Based on Edison, we implemented a benchmark, based on [Lew11], to exercise the different implementations provided in the library.
From the analysis of the experimental results we obtained, we can see that there are real energy savings to be realized, by substituting one implementation by another, depending on the usage pattern of the Edison Application Programming Interface (API) operations. Furthermore that substitution is quite straightforward, as most implementations adhere to a common API.
While the Edison library already incorporates an extensive unit test suite to guarantee functional correctness, it can benefit from the type of performance analysis we consider in this work [Doca].
To the best of our knowledge our study is one of only two in existence, to approach the energy consumption/efficiency focusing on the Haskell programming language. While we investigated a data structures library, [Lim16] has explored concurrent Haskell programs.

1.1 Research Questions

Our work is an attempt to answer the following general research question:

RQ.: To what extent can we save energy by refactoring existing Haskell programs to use different data structure implementations?

More specifically our study is motivated by the following more concrete research questions:

RQ1.: How do different *implementations* of the same abstractions compare in terms of runtime and energy efficiency?

RQ2.: For concrete operations, what is the relationship between their *performance* and their energy consumption?

In the next section, we briefly introduce the main Haskell programming language concepts that the reader will need to grasp, in order to be able to follow the discussion of the work in the rest of the document.

1.2 A background on Haskell

In this section we provide some background on Haskell, with the intent of helping the reader to understand terms and code samples presented later.
Haskell is a declarative, statically and strongly typed, lazy, purely functional language.
Being declarative means that a haskell program is a high level description of what needs to be done, not exactly of how, that is to be done.
By statically typed we mean that an Haskell expression/program has a type at compile time.
Strongly typed means each Haskell expression has one type, even if it is a polymorphic one, that is, if it is a *String* then is it not a *Bool*, or some other type.
Being lazy means an Haskell expression is only evaluated if, and when, it is first needed.

Being purely functional means that functions in `Haskell` are functions in the mathematical sense, pure, for the same inputs they will always return the same outputs.
A key aspect of `Haskell` programming are types. `Haskell` has a few primitive types e.g Float, Char, Integer, Bool, which are floating-point numbers, characters, arbitrary precision integers, and True or False, boolean values, respectively. We can also define our own types. The **type** *Name* = *String* expression defines a type synonym, *Name*, to mean *String*, a list of charaters ([*Char*], a predefined type). We can define a new data type[1] with the code pictured in Listing 1.1.

Listing 1.1 Data type definition example.

```
data Maybe a = Nothing | Just a
```

We have defined an "optional" data type *Maybe a* which can hold nothing (with the *Nothing* data constructor) or something (with the *Just* data constructor). Indeed, something, anything, because we used what is called a type variable, in this case *a*, to define a polymorphic type. Through appropriate instantiation the *Maybe a* type could assume the *Maybe Integer* or *Maybe Bool* types, although in different places in a program.
Let us now introduce a function that produces values, of type *Maybe Integer*.

Listing 1.2 Function definition example.

```
factorial :: Integer -> Maybe Integer
factorial n
    | n < 0 = Nothing
    | otherwise = Just ( fact n )
    where
        fact :: Integer -> Integer
        fact 0 = 1
        fact n = n * fact ( n - 1 )
```

In Listing 1.2 we see a top-level function definition for a *factorial* function.
To define a function in `Haskell` we optionally declare it's type, with a function signature and then write a series of equations. An expression's type is declared with the :: sign.
In our example, the function signature *factorial* :: *Integer* → *Maybe Integer* tells us the name of the function, and defines it's type, preceding and following the :: respectively. This function's specific type is: a function taking one input, of the *Integer* type, and returning one output, of *Maybe Integer* type. The "information" that it is a function is extracted from the presence of the → sign. With this information in hand the reader can hopefully glean the definition of another function *fact*, defined inside the first. This is a local function definition (introduced by a **where** clause), only "viewable" in the context of the first equation of the *factorial* function. The listed *factorial* function has only one equation, i.e. *factorial n*..., whereas the *fact* function has two equations, e.g., *fact* 0 = 1.
The *factorial* function makes use of guards, for example $|n<0 = Nothing$, meaning if the boolean expression $n < 0$ is true then the result will be *Nothing*. The *otherwise* part is a synonym for the *True* boolean value, meaning that if that guard is ever considered as the possible result then it will always succeed, and the result will be *Just* (*fact n*).

[1] Note: The *Maybe* type is predefined in Haskell.

Note, that the order of both the equations (e.g., for *fact*) and the guards (e.g., for *factorial*) is significant, they are "checked" from top to bottom.
The *fact* definition is an example of a recursive definition, the function calls on itself to perform some part of the total work required, although with a "simpler" set of parameters.
Regarding lazyness, unless instructed otherwise `Haskell` will only evaluate an expression (e.g. a piece of data) if it is really needed. It will only evaluate enough of it, meaning for example, that, by default, it will only evaluate it enough to discover it's first constructor. This is called Weak Head Normal Form (WHNF). This default can be overridden for specific components of a data type by using the ! construct in the type definition. The ! will appear in a few of the data types described in Chapter 2. If an expression is instead fully evaluated then we say it has been evaluated to Head Normal Form (HNF) (usually just called Normal Form (NF)). This can be achieved not only by the use of ! but also, as we will see in Chapter 4, by the use of the *deepseq* family of functions from the *Control.DeepSeq* module.
Let us now present another example, in Listing 1.3, with which we will introduce some more `Haskell` concepts.

Listing 1.3 Introduction of a few more Haskell concepts.

```
data BinTree a = Empty | Node a ( BinTree a ) ( BinTree a )

minTree :: Ord a => BinTree a -> Maybe a
minTree Empty = Nothing
minTree ( Node x leftSubTree rightSubTree ) =
    let
        minLSTree = minTree leftSubTree
        minRSTree = minTree rightSubTree
        minSubTrees = minMaybe minLSTree minRSTree
    in
        minMaybe ( Just x ) $ minSubTrees

    where
        minMaybe :: Ord a => Maybe a -> Maybe a -> Maybe a
        minMaybe Nothing     Nothing     = Nothing
        minMaybe ( Just x )  ( Just y )  = Just ( min x y )
        minMaybe ( Just x )  ( Nothing ) = Just x
        minMaybe ( Nothing ) ( Just y )  = Just y
```

In that listing, we can see a data definition for binary trees (*BinTree*) and a, top-level, *minTree* function which discovers the minimum element of a *BinTree*, if the tree is not empty.
There is also a definition of a, local, *minMaybe* function, which takes two *Maybe* values and returns another *Maybe* value. This function is defined in such a way that tries to match each of it's arguments to a predefined pattern. For example, the third equation in that definition, tries to match the first argument with a *Just x* pattern and it's second argument with a *Nothing* pattern. If both matches succeed then the right-hand side of that equation will be the result of the function. This mechanism is called pattern matching[2].
In the *minTree* function a, **let** ... **in**..., construct is used to make local value (it could also be function) definitions. Three local definitions are put in place (following "let"), the last of which depends on the first two. The order is not significant. Those definitions can then be used, in the following "in" part.

[2]The _ pattern matches anything.

Also in the *minTree* function, we used the `$` function to exemplify the fact that in `Haskell` functions are first order entities. They can be passed as parameters, returned as results and partially applied. The `$` function takes a function as a first parameter and applies it to it's second parameter. It's type is thus $(a \to b) \to a \to b$. In the example, the *minMaybe* function is partially applied to the *Just x* value, returning another function which takes just one *Maybe* parameter and returns a *Maybe* result. The `$` function receives that function as a parameter and applies it to the *minSubTrees* value, thus generating the final result of the *minTree* function[3]. The `$` function is, in this case, used as an infix function also called an operator.

1.3 Outline

The remainder of this work is structured as follows:

Chapter 2 in which we describe, Edison, a library of implementations for a few common data structure abstractions;

Chapter 3 in which we describe a benchmark and tools used in our work;

Chapter 4 in which we describe the methodology followed in implementing our work;

Chapter 5 in which we present the results obtained through our experimentation;

Chapter 6 in which we present the conclusions drawn.

[3] In the *minMaybe* definition a *min* function is used that can calculate the minimum of two values. It is part of the `Haskell`'s Prelude, the `Haskell`'s standard library of functions.

Chapter 2

A library of purely functional data structures

In this chapter we describe the `Edison` library [Oka01, Oka99], that we have relied on to compare different implementations of purely functional data structures.
`Edison` is a mature and well documented library that provides several functional data structures that implement three types of abstractions: `Sequences`, `Collections` and `Associative Collections`. While implementations of Edison are available in other programming languages, e.g., in `ML` [Oka99], here we focus on its `Haskell` version. In `Haskell` two packages make up the library, `EdisonAPI` [Docb] and `EdisonCore` [Docc]. The first of these defines interfaces, that the modules included in the second, must then implement.
In Table 2.1 the different implementations available for the mentioned abstractions are presented.

Table 2.1: Abstractions and Implementations available in Edison.

Sequences	Collections	Associative Collections
ListSeq		
BraunSeq		
FingerSeq	LazyPairingHeap	
SizedSeq	LeftistHeap	
RevSeq	MinHeap	StandardMap
JoinList	SplayHeap	AssocList
RandList	SkewHeap	PatriciaLoMap
BinaryRandList	StandardSet	TernaryTrie
SimpleQueue	EnumSet	
BankersQueue	UnbalancedSet	
MyersStack		

In the remainder of this chapter, we describe in detail the different abstractions and implementations provided by the library. In section 2.1 we describe `Sequences`; in section 2.2 we present `Collections`; finally, in section 2.3 we describe `Associative Collections`.

2.1 The Sequence abstraction

The `Sequence` abstraction models a conceptual data-structure type, in which different extremities are distinguished and a specific insertion/removal order, is favored. In `Edison`, the `Sequence` abstraction includes, e.g., lists, queues and stacks. Furthermore, all implementations of this abstraction define a reusable, coherent and uniform set of functions.
Examples of functions (and their types) defined over `Sequences` are: $lcons :: a \rightarrow Seq\ a \rightarrow Seq\ a$[1] and $rcons :: a \rightarrow Seq\ a \rightarrow Seq\ a$, which given an element of type a and a sequence of elements of type a, $Seq\ a$, produce a new sequence of the same type, obtained by inserting that element at the left, or right, of the original sequence, respectively; $concat :: Seq\ (Seq\ a) \rightarrow Seq\ a$

[1] In `Haskell`, the notation $e :: t$ is used to declare that expression e is of type t. Also, the notation $f :: a \rightarrow b$ is used to declare the type of a function f as "Taking (as input) something of type a to (and producing as output) something of type b".

which given a sequence, *Seq* (*Seq a*), containing a number of sequences of elements of type *a*, gathers all the elements in those sequences in one sequence of elements of type *a*, *Seq a*; and *map* :: $(a \rightarrow b) \rightarrow$ *Seq a* $\rightarrow$ *Seq b* which, given a function f, taking a value of type *a* and producing a value of type *b*, i.e., $f :: a \rightarrow b$, will apply *f* to all elements of a sequence of *a* typed elements, *Seq a*, transforming all elements to *b* typed elements, thereby producing as a result a sequence of elements of type *b*, *Seq b*.
The functions defined by the `Sequence` abstraction have associated, theoretical, asymptotic running time complexities, against which the corresponding complexities for all concrete implementations, compare. These default running times are presented in Table 2.2[2].

Table 2.2: Default asymptotic time complexities for Sequences.

Function	Time
map :: $(a \rightarrow b) \rightarrow$ *Seq a* $\rightarrow$ *Seq b*	O($t * n$)
singleton :: $a \rightarrow$ *Seq a*	O(1)
concatMap :: $(a \rightarrow$ *Seq b*$) \rightarrow$ *Seq a* $\rightarrow$ *Seq b*	O($t * n + m$)
empty :: *Seq a*	O(1)
append :: *Seq a* $\rightarrow$ *Seq a* $\rightarrow$ *Seq a*	O(n_1)
lcons :: $a \rightarrow$ *Seq a* $\rightarrow$ *Seq a*	O(1)
rcons :: $a \rightarrow$ *Seq a* $\rightarrow$ *Seq a*	O(n)
fromList :: $[a] \rightarrow$ *Seq a*	O(n)
copy :: *Int* $\rightarrow a \rightarrow$ *Seq a*	O(n)
lhead :: *Seq a* $\rightarrow a$	O(1)
ltail :: *Seq a* $\rightarrow$ *Seq a*	O(1)
rhead :: *Seq a* $\rightarrow a$	O(n)
rtail :: *Seq a* $\rightarrow$ *Seq a*	O(n)
null :: *Seq a* $\rightarrow$ *Bool*	O(1)
size :: *Seq a* $\rightarrow$ *Int*	O(n)
toList :: *Seq a* $\rightarrow [a]$	O(n)
concat :: *Seq* (*Seq a*) $\rightarrow$ *Seq a*	O($n + m$)
reverse :: *Seq a* $\rightarrow$ *Seq a*	O(n)
reverseOnto :: *Seq a* $\rightarrow$ *Seq a* $\rightarrow$ *Seq a*	O(n_1)
fold :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow$ *Seq a* $\rightarrow b$	O($t * n$)
fold1 :: $(a \rightarrow a \rightarrow a) \rightarrow$ *Seq a* $\rightarrow a$	O($t * n$)
foldr :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow$ *Seq a* $\rightarrow b$	O($t * n$)
foldl :: $(b \rightarrow a \rightarrow b) \rightarrow b \rightarrow$ *Seq a* $\rightarrow b$	O($t * n$)
foldr1 :: $(a \rightarrow a \rightarrow a) \rightarrow$ *Seq a* $\rightarrow a$	O($t * n$)
foldl1 :: $(a \rightarrow a \rightarrow a) \rightarrow$ *Seq a* $\rightarrow a$	O($t * n$)
reducer :: $(a \rightarrow a \rightarrow a) \rightarrow a \rightarrow$ *Seq a* $\rightarrow a$	O($t * n$)
reducel :: $(a \rightarrow a \rightarrow a) \rightarrow a \rightarrow$ *Seq a* $\rightarrow a$	O($t * n$)
reduce1 :: $(a \rightarrow a \rightarrow a) \rightarrow$ *Seq a* $\rightarrow a$	O($t * n$)
take :: *Int* $\rightarrow$ *Seq a* $\rightarrow$ *Seq a*	O(i)
drop :: *Int* $\rightarrow$ *Seq a* $\rightarrow$ *Seq a*	O(i)
splitAt :: *Int* $\rightarrow$ *Seq a* $\rightarrow$ (*Seq a*, *Seq a*)	O(i)
subseq :: *Int* $\rightarrow$ *Int* $\rightarrow$ *Seq a* $\rightarrow$ *Seq a*	O($i + len$)
filter :: $(a \rightarrow$ *Bool*$) \rightarrow$ *Seq a* $\rightarrow$ *Seq a*	O($t * n$)

[2]Function "families" like fold* and reduce* include strict versions which are not presented.

Table2.2 – continued from previous page

Function	Time
partition :: $(a \to Bool) \to Seq\ a \to (Seq\ a, Seq\ a)$	O($t * n$)
takeWhile :: $(a \to Bool) \to Seq\ a \to Seq\ a$	O($t * n$)
dropWhile :: $(a \to Bool) \to Seq\ a \to Seq\ a$	O($t * n$)
splitWhile :: $(a \to Bool) \to Seq\ a \to (Seq\ a, Seq\ a)$	O($t * n$)
inBounds :: $Int \to Seq\ a \to Bool$	O(i)
lookup :: $Int \to Seq\ a \to a$	O(i)
lookupWithDefault :: $a \to Int \to Seq\ a \to a$	O(i)
update :: $Int \to a \to Seq\ a \to Seq\ a$	O(i)
adjust :: $(a \to a) \to Int \to Seq\ a \to Seq\ a$	O($i + t$)
mapWithIndex :: $(Int \to a \to b) \to Seq\ a \to Seq\ b$	O($t * n$)
foldrWithIndex :: $(Int \to a \to b \to b) \to b \to Seq\ a \to b$	O($t * n$)
foldlWithIndex :: $(b \to Int \to a \to b) \to b \to Seq\ a \to b$	O($t * n$)
zip :: $Seq\ a \to Seq\ b \to Seq\ (a, b)$	O($min(n_1, n_2)$)
zipWith :: $(a \to b \to c) \to Seq\ a \to Seq\ b \to Seq\ c$	O($t * min(n_1, n_2)$)
unzip :: $Seq\ (a, b) \to (Seq\ a, Seq\ b)$	O(n)
unzipWith :: $(a \to b) \to (a \to c) \to Seq\ a \to (Seq\ b, Seq\ c)$	O($t * n$)

In Table 2.2, and in the implementation specific tables, presented later, the timings are given, generally, in terms of, n, the size of a single parameter sequence; t, the running time of a parameter function; n_1 and n_2, the sizes of two parameter sequences; m, the size of an output sequence; i, an index of an element of a sequence; and len, a length of a portion of a sequence. In the remainder of this section we describe in more detail each of the `Sequence` implementations available in `Edison`.

2.1.1 The ListSeq implementation

The underlying data type for the *ListSeq* implementation is the standard list type defined in the *Prelude*[3]:

type *Seq a* = $[a]$

The asymptotic time complexities of this implementation are the baseline for the library (as published in the module *Data.Edison.Seq*). Only the functions *toList* and *fromList* differ. The differences are presented in Table 2.3.

Table 2.3: Asymptotic time complexities, for the ListSeq implementation, that differ from the baseline.

Function	Time
toList, *fromList*	O(1)

2.1.2 The BraunSeq implementation

The *BraunSeq* implementation relies on a balanced binary tree [DD09] as an underlying data-structure. It is encoded as the following `Haskell` data type:

[3]The *Prelude* is `Haskell`'s standard library of functions.

data *Seq a* $=$ *E* | *B a* (*Seq a*) (*Seq a*)

A tree might be empty, or a tree with an element at every branch and empty leaves.
In this implementation an invariant is maintained: the left subtree is either exactly the same size as the right subtree, or at most one element larger.
The asymptotic time complexities differ from the defaults for the functions in Table 2.4.

Table 2.4: Asymptotic time complexities, for the BraunSeq implementation, that differ from the baseline.

Function	Time
lview, lcons, ltail*	O($\log n$)
rcons, rview, rhead*, rtail*, size	O($\log^2 n$)
copy, inBounds, lookup*, update, adjust	O($\log i$)
append	O($n_1 \log n_2$)
concat	O($n + m \log m$)
drop, splitAt	O($i \log n$)
subseq	O($i \log n + len$)
reverseOnto	O($n_1 \log n_2$)
concatMap	O($n * t + m \log m$)

2.1.3 The FingerSeq implementation

The *FingerSeq* implementation realizes the *Sequence* abstraction, making use of a general-purpose data structure, a *FingerTree* [HP06]. The underlying data type is[4]:

data *Digit a*
$=$ *One a*
| *Two a a*
| *Three a a a*
| *Four a a a a*

data *Node v a* $=$ *Node*$_2$ $!v$ *a a* | *Node*$_3$ $!v$ *a a a*

data *FingerTree v a*
$=$ *Empty*
| *Single a*
| *Deep* $!v$ $!(Digit\ a)$ (*FingerTree v* (*Node v a*)) $!(Digit\ a)$

newtype *Seq a* $=$ *Seq* (*FingerTree SizeM* (*Elem a*))

No asymptotic time complexities are given in the documentation for the *FingerSeq* implementation. But, looking at the source code for *FingerSeq* we conclude that quite a number of the functions defined there, are implemented with a simple: unwrap from the Seq constructor, compute with FingerTree provided function and rewrap in the Seq constructor, style, for example, for *rcons*: *rcons x* $=$ *Seq* $\circ$ *FT.rcons* (*Elem x*) $\circ$ *unSeq*. Therefore, the few, time complexities given for *FingerTree* are "the same" for *FingerSeq*. From these, those that differ from the default complexities listed earlier, are presented in Table 2.5.

2.1.4 The SizedSeq implementation

The *SizedSeq* implementation is not really a sequence implementation. It is an adaptor over a parameter, existing, implementation. It keeps track of the sequence size explicitly.

[4]The $!v$ is used to make the evaluation of *v*, strict/eager (`Haskell`'s default is lazy).

Table 2.5: Asymptotic time complexities, for the FingerSeq implementation, that differ from the baseline.

Function	Time
rcons	O(1)
rview	O(1)
append	O($\log(min(n_1, n_2))$)

The underlying data type is:

data *Sized s a* $= N\ !Int\ (s\ a)$

The *N* data constructor wraps:

- the *s a*; a sequence implementation *s* in which the elements are of type *a*;
- an *Int*, which is the size of the sequence.

All operations time complexities are those of the underlying implementation, except that of the *size* operation, given in Table 2.6

Table 2.6: Asymptotic time complexities, for the SizedSeq implementation, that differ from the baseline.

Function	Time
size	O(1)

2.1.5 The RevSeq implementation

The *RevSeq* implementation is also an adaptor for previously existing implementations.
It reverses the order of the elements in the wrapped sequence implementation. This adaptor is useful if an implementation has, for example, fast access times in its right-hand side, but we want to revert this to the left-hand side. This adaptor also keeps track of the sequence size.
The underlying data type is:

data *Rev s a* $= N\ !Int\ (s\ a)$

This datatype is the same datatype as for the *SizedSeq* adaptor.
The asymptotic time complexities for the application of this adaptor over a underlying existing implementation are determined by that implementation, except that the access times for both sides of the sequence are exchanged. Also the *size* operation time complexity differs as stated in Table 2.7.

Table 2.7: Asymptotic time complexities, for the RevSeq implementation, that differ from the baseline.

Function	Time
size	O(1)

2.1.6 The JoinList implementation

The JoinList sequence implementation is based on a tree data-structure [KoC15], which might be empty (*E*), or will contain elements only in it's leaves (*L a*)[5].

[5]This tree data-structure is called a leaftree.

The underlying data type is:

data *Seq a* = *E* | *L a* | *A* (*Seq a*) (*Seq a*)

An invariant: *E* never a child of *A*, must be maintained.
The asymptotic time complexities differ from the defaults for the functions in Table 2.8.

Table 2.8: Asymptotic time complexities, for the JoinList implementation, that differ from the baseline.

Function	Time
rcons, append	O(1)
ltail*, lview	O(1) when used single-threaded, O(n) otherwise
lhead*	O(n)
inBounds, lookup	O(n)
copy	O($\log i$)
concat	O(n_1)
concatMap	O($n * t$)

2.1.7 The RandList implementation

The *RandList* implementation aims to provide a data-structure that supports both efficient access to random elements contained in it, and primitive list operations (*head*, *cons*, *tail*) that run as fast as their native list counterparts [Oka95a].
That data-structure is a list of complete binary trees [She09] with elements of a type *a*.
The underlying data type is:

data *Tree a* = *L a* | *T a* (*Tree a*) (*Tree a*)
data *Seq a* = *E* | *C* $!Int$ (*Tree a*) (*Seq a*)

Two invariants must be maintained:

- the list of complete binary trees is maintained in non-decreasing order of size;
- the first argument to the data-construtor *C* is the number of nodes in the encapsulated tree.

The asymptotic time complexities differ from the defaults for the funtcions in Table 2.9.

Table 2.9: Asymptotic time complexities, for the RandList implementation, that differ from the baseline.

Function	Time
rhead*, size	O($\log n$)
copy, inBounds	O($\log i$)
lookup*, update, adjust, drop	O($min(i, \log n)$)
subseq	O($min(i, \log n) + len$)

2.1.8 The BinaryRandList implementation

The *BinaryRandList* implementation represents a linear data structure, which may be empty (with the *E* data constructor) or have two distinct recursive cases that model the fact that the list has an even (with the *Even* data constructor), or odd (with the *Odd* data constructor), number of elements [Oka99].
The underlying data type is:

data *Seq a* $=$ *E* | *Even* $(Seq\ (a, a))$ | *Odd a* $(Seq\ (a, a))$

The asymptotic time complexities differ from the defaults for the functions in Table 2.10.

Table 2.10: Asymptotic time complexities, for the BinaryRandList implementation, that differ from the baseline.

Function	Time
lcons, lhead, ltail*, lview*, rhead*, size, lookup*, update, adjust, drop	O($\log n$)
copy, inBounds	O(i)
append, reverseOnto	O($n_1 + \log n_2$)
take, splitAt	O($i + \log n$)
subseq	O($\log n + len$)
zip	O($min(n_1, n_2) + \log max(n_1, n_2)$)

2.1.9 The SimpleQueue implementation

The SimpleQueue implementation of the Sequence abtraction is based on two lists. One representing the front of the queue, and the second, the rear of the queue.
The underlying data type is:

data *Seq a* $=$ *Q* $[a]$ $[a]$

That is the data constructor *Q* encapsulates the two standard `Haskell` lists mentioned before.
The rear is maintained in reverse order, the first element of the rear list is actually the last element of the sequence.
An invariant must be obeyed/maintained: the front will be empty only if the rear is also empty.
This guarantees that the first element of the queue can always be accessed in O(1) time [Oka99].
The asymptotic time complexities differ from the defaults for the functions in Table 2.11.

Table 2.11: Asymptotic time complexities, for the SimpleQueue implementation, that differ from the baseline.

Function	Time
rcons, fromList	O(1)
lview, ltail*	O(1) if single threaded, O(n) otherwise
inBounds, lookup, update, drop, splitAt	O(n)

2.1.10 The BankersQueue implementation

The *BankersQueue* implementation of the `Sequence` abtraction is similar to the *SimpleQueue* implementation. The differences are that the size of the sequence is tracked explicitly and a different invariant is abided by.
The underlying data type is:

data *Seq a* $=$ *Q* $!Int$ $[a]$ $[a]$ $!Int$

That is, the *Q* data constructor encapsulates two *Int*s and two lists. The first list represents the front of the queue; the second list represents the rear of the queue. The first *Int* is the length (or size) of the front, and the second the length of the rear. The rear list is maintained in reverse order (the first element is the last element of the sequence).
An invariant must be obeyed/maintained: the front will be at least as long as the rear [Oka95b].
The asymptotic time complexities differ from the defaults for the functions in Table 2.12.

Table 2.12: Asymptotic time complexities, for the BankersQueue implementation, that differ from the baseline.

Function	Time
rcons, size, inBounds	O(1)

2.1.11 The MyersStack implementation

The MyersStack sequence implementation is a realization of the stack abstraction which also permits accesses to the kth element [Mye83].
The underlying data type is:

data *Seq a* = *E* | *C* !Int *a* (*Seq a*) (*Seq a*)

This represents a binary tree (as already stated for other implementations).
The asymptotic time complexities differ from the defaults for the functions in Table 2.13.

Table 2.13: Asymptotic time complexities, for the MyersStack implementation, that differ from the baseline.

Function	Time
lookup, inBounds, drop	O($min(i, \log n)$)
rhead*, size	O($\log n$)
subseq	O($min(i, \log n) + len$)

2.2 The Collection abstraction

The `Collections` abstraction includes `Sets` and `Heaps` (priority queues where the priority is the element). In this thesis `Heaps` and `Sets` are described in detail in sections 2.2.1 and 2.2.2 respectively.
A `Collection` in `Edison` is characterized by whether or not it satisfies three properties:

1. observability: whether the elements in a collections can, or not, be recovered from the collection.
2. ordering: whether the type of elements in a collection satisfies a total ordering requirement;
3. uniqueness: whether the elements in a collection are distinct;

Currently all `Collections` in `Edison` abide by the observability and ordering properties, with `Sets` also guarantying the uniqueness of the elements stored in them.
No default asymptotic time complexities are provided for this abstraction. Regarding specific implementations there is only one source of such complexities, the *Data.Set* library, which is the underlying implementation for the *StandardSet* realization of the abstraction.

2.2.1 Heaps

A `Heap` is a `Collection`, generally based on a tree shaped data structure, and usually maintaining the minimum (it could also be the maximum) element readily available for "inspection". That is, determining the minimum element should be an computationally inexpensive operation.

2.2.1.1 The LazyPairingHeap implementation

The *LazyPairingHeap* implementation is a heap-ordered tree which can branch one-way, or two-way, depending on the number of (odd or even) children[Oka99].
The underlying data type is:

data *Heap a* $=$ *E* $\mid$ H_1 *a* (*Heap a*) $\mid$ H_2 *a* $!(Heap\ a)$ (*Heap a*)

A well-formed *LazyPairingHeap* abides by the invariant: the left child of a H_2 node must not be empty.

2.2.1.2 The LeftistHeap implementation

A *LeftistHeap* is a heap implementation based on a heap ordered binary tree [Oka99]. Being heap ordered means that whatever the node in the tree, the element contained in it is no larger than the elements in the nodes of it's subtrees. Also the tree must conform to the so-called leftist property. This property states that for any node, the rank of it's left subtree is no lesser than the rank of it's right subtree. The rank of a node is defined to be, the length of the rightmost path from that node to an empty node (the length of the node's right spine).
The underlying data type is:

data *Heap a* $=$ *E* $\mid$ *L* $!Int$ $!a$ $!(Heap\ a)$ $!(Heap\ a)$

2.2.1.3 The MinHeap implementation

The *MinHeap* "implementation" is really just an adaptor for other heap implementations, that keeps the minimum element separately.
The underlying data type is:

data *Min h a* $=$ *E* $\mid$ *M a h*

2.2.1.4 The SplayHeap implementation

The *SplayHeap* collection implementation is based on a splay tree [ST85]. A splay tree is similar to a balanced binary search tree. In a splay tree the balancing is carried out as operations over the data struture are performed, by way of transformations that tend to increased the balance, but no explicit information regarding that purpose is kept inside the data structure [Oka99]. Data structures behaving in this way are usually called `self-adjusting`.
The underlying data type is:

data *Heap a* $=$ *E* $\mid$ *T* (*Heap a*) *a* (*Heap a*)

The elements in the heap are maintained in binary search tree order [SW14a] (duplicates allowed).

2.2.1.5 The SkewHeap implementation

The *SkewHeap* data structure is a self-adjusting implementation akin to the *LeftistHeap* implementation [ST86].
The underlying data type is:

data *Heap a* $=$ *E* $\mid$ *T a* (*Heap a*) (*Heap a*)

2.2.2 Sets

A `Set` is a `Collection` in wich no duplicate elements are allowed.
Characteristic functions defined over `Sets` are, for example: *intersection* :: *Set a* $\rightarrow$ *Set a* $\rightarrow$ *Set a*, which computes the common elements of two sets, and *difference* :: *Set a* $\rightarrow$ *Set a* $\rightarrow$ *Set a*, which calculates the member elements of a set not members of another set, both taking two sets of elements of type *Set a* and producing another set of the same type, and *subset* :: *Set a* $\rightarrow$ *Set a* $\rightarrow$ *Bool* which returns a True or False (Bool[6]) value indicating if the first parameter set is a subset of the second parameter set.

2.2.2.1 The StandardSet implementation

The *StandardSet* implementation is just a wrapper around the standard `Haskell` library *Data.Set*.
The *Data.Set* implementation is based on size balanced binary trees [NR72].
The underlying data type is:

type *Set* = *Data.Set.Set*

2.2.2.2 The EnumSet implementation

In this implementation of the `Set` abstraction, its instances (sets) are realized recurring to "bit strings" and bitwise operations over those "strings".
The underlying data type is:

newtype *Set a* = *Set Word*

This set implementation can only be used to model sets for which the maximum number of elements that may appear in the set is less than or equal to the number of bits in the *Word*[7] type.

2.2.2.3 The UnbalancedSet implementation

In this implementation a set is modeled as an unbalanced binary search tree. In such a tree no equilibrium in the distribution of nodes/elements is enforced, and as such, the performance of operations over the tree may degenerate into that of a simple list.
The underlying data type is:

data *Set a* = *E* | *T* (*Set a*) *a* (*Set a*)

On instances of this datatype an invariant is maintained, the binary search tree order [SW14a]. That is, for any node with an element y, all elements in the left subtree are lesser than y, and, all elements in the right subtree are greater than y.

2.3 Associative Collections

The `Associative Collections` abstraction includes e.g., finite maps, finite relations and priority queues (with distinct priority and element[8]). They generically map keys of a type *k* to values of

[6] The `Haskell` Boolean type.
[7] The `Haskell` *Word* type is an unsigned integral type, of size equal to that of the type *Int*.
[8] Whereas in a queue the priority is drawn from the insertion order, here the priority is a stand-alone value.

a type *a*. Exceptions are the *PatriciaLoMap* and *TernaryTrie* implementations which use more restricted types of keys (*Int* and $[k]$ respectively). The other implementations respect the same `API`.

`Associative Collections`, like `Collections`, are charaterized by the three proterties already mentioned for the later, observability, ordering and uniqueness.

No default asymptotic time complexities are provided for this abstraction. Regarding specific implementations there is only one source of such complexities, the *Data.Map* library, which is the underlying implementation for the *StandardMap* realization of the abstraction.

In the remainder of this section we describe the available implementations in `Edison`.

2.3.1 The StandardMap implementation

The *StandardMap* implementation is just a wrapper around the standard `Haskell` library *Data.Map*.

The *Data.Map* implementation is based on size balanced binary trees [NR72].

The underlying data type is therefore:

type *FM* $=$ *Data.Map.Map*

2.3.2 The AssocList implementation

The *AssocList* implementation realizes the `Associative Collections` abstraction via an association list [Wik16]. An association list is basically a collection of pairs in which one of the components is called the key, and the other is called the value. Each such pair is interpreted as one mapping from the key to the value, in the list.

The underlying data type is:

data *FM k a* $=$ *E* | *I k a* $(FM\ k\ a)$

In the *AssocList* implementation duplicate associations are removed conceptually, but not physically. If duplicate associations are found then the first occurrence of a key is the one considered to be in the map.

2.3.3 The PatriciaLoMap implementation

The *PatriciaLoMap* implementation realizes finite maps based on little-endian patricia trees [OG98].

The underlying data type is:

data *FM a* $=$ *E* | *L Int a* | *B Int Int* $!(FM\ a)\ !(FM\ a)$

The *PatriciaLoMap* implementation abides by a number of invariants, e.g. “no *B* node has an *E* child”.

2.3.4 The TernaryTrie implementation

The *TernaryTrie* implementation models finite maps as ternary search tries[9] [SW14b].

A ternary search trie is a tree shaped structure, in which, each node branches into three subtrees. Also each node contains a part of a key which will ultimately lead to a value associated

[9] Trie is pronounced as try!

with the whole of that key. The three subtrees are associated with a part of a key that is considered to be lesser, equal or greater than the part stored at the node from which those subtrees stem from.
The tree structure is kept balanced.
The underlying data type is:

> **data** *FM k a* = *FM* $!(Maybe\ a)\ !(FMB\ k\ a)$
> **data** *FMB k v* = *E* | *I* $!Int\ !k\ !(Maybe\ v)\ !(FMB\ k\ v)\ !(FMB'\ k\ v)\ !(FMB\ k\ v)$
> **newtype** *FMB′ k v* = *FMB′* $(FMB\ k\ v)$

2.4 Final Remarks

In this chapter, we have described a software library that offers a number of purely functional implementations, for three different data structure abstractions. This library will be our object of study. In the next chapter we will present the "environment" in which our study was conducted, and the tools used to perform this study.

Chapter 3

Experimental Setting

In Chapter 2, we have described a library of several different implementations for common data structure abstractions, Edison.
In order to evaluate the performance of those data structure implementations according to some criteria, we need to create programs ("functions") that make use of those implementations, and execute these functions (run the programs) while measuring certain characteristics of interest.
The set of functions/programs to run is called a benchmark. The benchmark (operations/components) used in this work is described in Section 3.1.
To actually execute the "programs" of the benchmark and record the measures of interest, we make use of a benchmarking tool named Criterion, described in Section 3.2.
In Section 3.3, we allude to the underlying technology that allows us to gather the energy consumption measures, that are the driving purpose of this study, the RAPL interface[Cou14].
Finally, in Section 3.4, the underlying hardware/software setup used in the execution of our measuring is presented.

3.1 The Benchmark

Following the approach considered in different studies [PCS+16, MPC14, Ca14, PLCL16], our benchmark is inspired by the microbenchmark to evaluate the run time performance of Java's JDK (Java Development Kit) Collection API (Application Programming Interface) implementations, presented in [Lew11]. The operations that are defined in such benchmark are listed in Table 3.1.

Table 3.1: Benchmark Operations.

$iters$	$operation$	$base$	$elems$
1	add	100000	100000
1000	addAll	100000	1000
1	clear	100000	n.a.
1000	contains	100000	1
5000	containsAll	100000	1000
1	iterator	100000	n.a.
10000	remove	100000	1
10	removeAll	100000	1000
10	retainAll	100000	1000
5000	toArray	100000	n.a.

All the operations can be abstracted by the format:

$$iters * operation(base, elems)$$

This format reads as: iterate $operation$ a given number of times ($iters$) over a data structure with a $base$ number of elements. If $operation$ requires an additional data structure, the number of elements in it is given by $elems$. All the operations are suggested to be executed over a base

structure with 100000 elements. So, the second entry in the table suggests adding 1000 times all the elements of a structure with 1000 elements to the base structure (of size 100000).

3.2 A library for implementing/conducting microbenchmarks

`Criterion` [O'S09] is a microbenchmarking library that is used to measure the performance of `Haskell` code. It provides a framework for both the execution of the benchmarks as well as the analysis of their results, being able to measure events with duration in the order of picoseconds. `Criterion` is robust enough to filter out noise coming, e.g., from the clock resolution, the operating system's scheduling or garbage collection. `Criterion`'s strategy to mitigate noise is to measure many runs of a benchmark in sequence and then use a linear regression model to estimate the time needed for a single run. That way, the outliers become visible.
Having been proposed in the context of a functional language with lazy evaluation, `Criterion` natively offers mechanisms to evaluate the results of a benchmark in different *depths*, such as Weak Head Normal Form or Normal Form.
`Criterion` is able to measure CPU time, CPU cycles, memory allocation and garbage collection. In our work, we have utilized a modified version which has had its domain extended so that it is also able to measure the amount of energy consumed during the execution of a benchmark[Lim16]. The adaptation of `Criterion` has been conducted based on two essential considerations. First, the energy consumed in the sampling time intervals used by `Criterion` is obtained via external C function invocations to RAPL (Section 3.3). This is similar to the time measurements natively provided by `Criterion`, which are also realized via Foreign Function Interface[1] (FFI) calls. Second, we need to handle possible overflows occurring on RAPL registers [DGH+10]. For two consecutive reads `x` and `y` of values in such registers, this was achieved by discarding the energy consumed in the corresponding (extremely small) time interval if `y`, which is read later, is smaller than `x`.
In the extended version of `Criterion`, energy consumption is measured in the same execution of the benchmarks which is used to measure runtime performance. In this version, all the aforementioned aspects of `Criterion`'s original methodology have straightforwardly been adapted for energy consumption analysis. The source code for the modified Criterion is available on GitHub[2].
In the remainder of this section we will exemplify the usage of the library. Consider the source code in Listing 3.1.
This code defines the straightforward recursive version of the factorial function, taking one argument *n*, which calculates the factorial of a non-negative *Integer* number. It also defines the *main*[3] function which makes use of the benchmarking machinery provided by Criterion. In this *main* function we define:

- a group of benchmarks, with the *bgroup* function, named "factorialBGroup" (multiple groups of benchmarks are allowed);
- within this group, four benchmarks, each with it's own label ("a" to "d"), with the *bench* function.

[1] The Foreign Function Interface is `Haskell`'s interfacing mechanism to software components written in other programming languages.
[2] https://github.com/green-haskell/criterion
[3] The *main* function in a *Main* module is the entry point to a `Haskell` program.

Listing 3.1 Criterion benchmark implementation example.

```
import Criterion.Main

-- The function to benchmark.
factorial :: Integer -> Maybe Integer
factorial n
    | n < 0 = Nothing
    | otherwise = Just ( fact n )
    where
        fact :: Integer -> Integer
        fact 0 = 1
        fact n = n * fact ( n - 1 )

-- Our benchmark harness.
main = defaultMain [
    bgroup "factorialBGroup" [
          bench "a" $ whnf factorial ( -1 )
        , bench "b"  $ whnf factorial 0
        , bench "c"  $ whnf factorial 2
        , bench "d" $ whnf factorial 16
        ]
    ]
```

We also use the *whnf* function to instruct Criterion, to evaluate the factorial function's result to Weak Head Normal Form. It is also possible to request from Criterion the evaluation to Normal Form, using the *nf* function instead.

Function *whnf* receives as arguments the function to evaluate, saturated with all but the last of it's parameters, and the evaluated function's last parameter. These are the *factorial* function and each of the "-1", "0", "2", "16" numbers, respectively.

An example usage/output of such a benchmark is demonstrated in Figure 3.1. Executing the benchmark with the "--regress energy:iters" option (the first line in Figure 3.1) we get the (abbreviated) outputs following in that figure. The output is "broken" by benchmarks groups and benchmarks. In this example we have only one group of benchmarks, factorialBGroup, and four benchmarks, "a" to "d", of which only two, "a" and "d" are pictured.

The output lines we are most interested in are the `time` line, and the `iters` line.

The `time` line displays the estimated time taken by a single execution of the function being benchmarked. In the example output, benchmark "a" i.e., each execution of (*factorial* (-1)), runs in approximately 7.230 nanoseconds (ns). This estimation is obtained by a "Ordinary Least-Squares" (OLS) linear regression, obtained from the raw measurements (times to run a number of iterations/executions).

The `iters` line, results from the same kind of procedure as the `time` line (OLS linear regression on raw measurements) but presents us with an estimate of the energy consumed by one execution of the function being benchmarked. For example, running benchmark "d", tells us that an execution of the *factorial* 16 consumes around 1.5e-7 Joules (J).

The first value in each line is the main estimate for the measure in that line, the values in parentheses are lower and upper bounds on the estimate.

The R^2 lines (the first unlabeled line and `energy` labeled line) show a value that "measures" the accuracy of the linear regression. It's called R^2 goodness-of-fit, and it's value should lie between 0.99 and 1.

Figure 3.1: Modified Criterion, usage/output example.

```
test-fact --regress energy:iters

benchmarking factorialBGroup/a
time                 7.230 ns   (7.191 ns .. 7.279 ns)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 7.231 ns   (7.196 ns .. 7.272 ns)
std dev              125.7 ps   (110.1 ps .. 138.0 ps)
energy:              0.995 R²   (0.992 R² .. 0.997 R²)
  iters              1.326e-7   (1.303e-7 .. 1.345e-7)
  y                  -2.268e-2  (-2.747e-2 .. -1.766e-2)
variance introduced by outliers: 25% (moderately inflated)

...

benchmarking factorialBGroup/d
time                 7.769 ns   (7.670 ns .. 7.948 ns)
                     0.997 R²   (0.993 R² .. 1.000 R²)
mean                 7.728 ns   (7.674 ns .. 7.905 ns)
std dev              300.2 ps   (18.50 ps .. 625.4 ps)
energy:              0.991 R²   (0.987 R² .. 0.994 R²)
  iters              1.500e-7   (1.446e-7 .. 1.547e-7)
  y                  -2.505e-2  (-3.149e-2 .. -1.917e-2)
variance introduced by outliers: 64% (severely inflated)
```

A more detailed example/explanation, without the energy measuring extension, can be found on the tool author's webpage at `http://www.serpentine.com/criterion/tutorial.html`.

3.3 An interface for measuring energy consumption

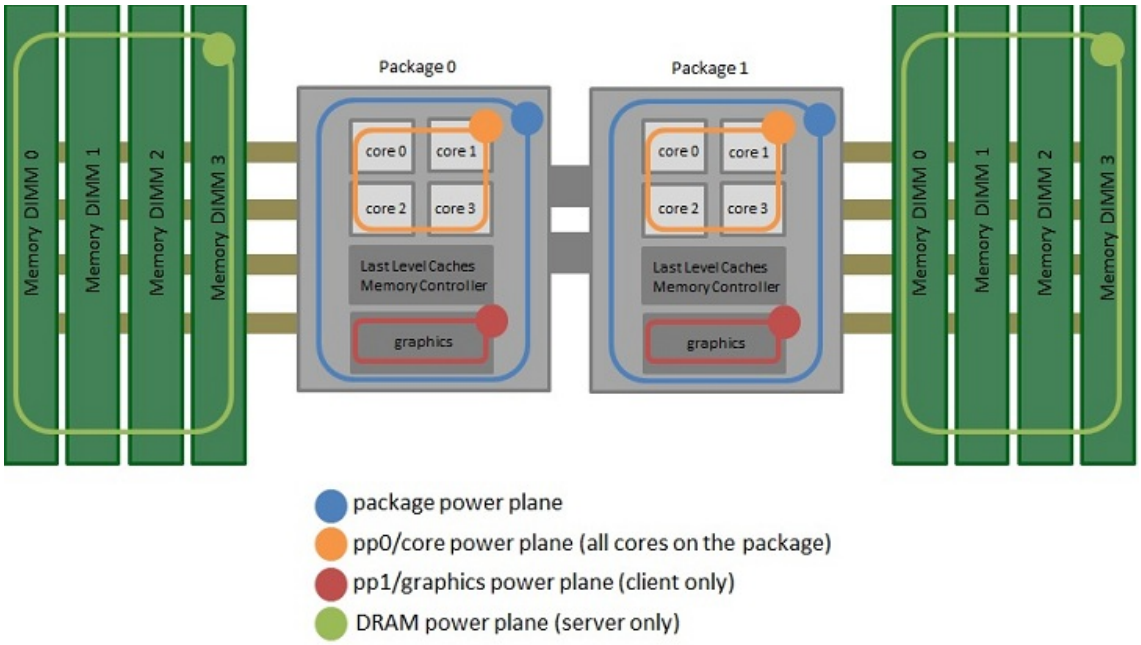
Running Average Power Limit (RAPL) [DGH+10] is an interface provided by modern Intel processors, using the Sandy Bridge and successor microarchitectures (roughly, Core second generation microprocessors and successors), to allow setting custom power limits to the processor packages. Using this interface one can access energy and power readings via a model-specific register (MSR). RAPL uses a software power model to estimate the energy consumption based on various hardware performance counters, temperature, leakage models and I/O models [WJK+12]. Its precision and reliability has been extensively studied [RNA+12, HDVH12].
RAPL interfaces operate at the granularity of a processor socket (package). There are MSRs to access 4 domains:

- Package (PKG): total energy consumed by an entire socket
- Power Plane 0 (PP0): energy consumed by all cores and caches
- Power Plane 1 (PP1): energy consumed by the on-chip Graphics Processing Unit (GPU)
- Dynamic Random Access Memory (DRAM): energy consumed by all Dual In-line Memory Modules (DIMMs)

The client (consumer desktop) platforms have access to {PKG, PP0, PP1} while the server platforms have access to {PKG, PP0, DRAM}. These domains are illustrated in Figure 3.2[4].

[4]Source: `https://software.intel.com/en-us/articles/intel-power-governor`

Figure 3.2: RAPL domains.



For this work, we collected the energy consumption data from the PKG domain using the `msr` module of the Linux kernel to access the MSR readings.

3.4 The test-bed

For this study, all experiments were conducted on a machine with 2x10-core Intel Xeon E5-2660 v2 processors (Ivy Bridge microarchitecture) and 256GB of DDR3 1600MHz memory. This machine runs the Ubuntu Server 14.04.3 LTS (Linux kernel 3.19.0-25) Operating System (OS). The compiler was Glasgow Haskell Compiler (GHC) 7.10.2, using Edison 1.3 (Chapter 2), and the modified Criterion (Section 3.2) library. Also, all experiments were performed with no other load on the OS.

3.5 Final Remarks

In this chapter, we have described the benchmark on which our work is based, the benchmarking tool, Criterion, used to measure both execution time and energy consumption of the benchmark and the RAPL interface which allows us gather the energy consumption measures from the processor MSRs.
Together with the Edison library, described in Chapter 2, this comprises all the technology needed to perform our proposed study.
In the next chapter, we shall describe the methodology followed, while undertaking that study.

Chapter 4

Experimental Methodology

Our analysis proceeded by applying the benchmark defined in the Section 3.1 to the different implementations provided by `Edison`.
For different reasons, we ended up excluding some implementations from our experimental setting. This was the case of `RevSeq` and `SizedSeq`, for `Sequences`, and `MinHeap` for `Heaps`, since they are adaptors of other implementations for the corresponding abstractions. `EnumSet`, for `Sets`, was not considered because it can only hold a limited number of elements, which makes it incompatible with the considered benchmark. As said before, `PatriciaLoMap` and `TernaryTrie` are not totally compatible with the `Associative Collections API`, so they could not be used in our uniform benchmark. Finally, `MyersStack`, for `Sequences` was discarded since its underlying data structure has redundant information in such a way that fully evaluating its instances has exponential behaviour. We have also split the comparison of `Collections` in independent comparisons of `Heaps` and `Sets`. This is due to the fact that these abstractions do not strictly adhere to the same `API`.
The methodology we followed to implement the benchmark operations described in Section 3.1, was to use the functions from Edison, the library described in Chapter 2, that in each case, more closely interpret these operations. In Table 4.1 we present the complete list of `Edison` functions that were used in the implementation of the benchmark operations. In the first column the benchmark operations are listed. In the remaining columns, the first row identifies each data structure type, and the rest, the functions with which we implemented the corresponding operation on the first column.

Table 4.1: Edison functions used to implement the benchmark operations.

	Sequences	**Collections**		**Associative Collections**
		Sets	**Heaps**	
add	lcons, rcons	insert	insert	insert
addAll	append	union	union	union
clear	null, ltail	difference	minView, delete	difference
contains	null, filter	member	member	member
containsAll	foldr, map, null, filter	subset	null, member, minView	submap
iterator	map	foldr	fold	map
remove	null, ltail	deleteMin	deleteMin	null, deleteMin
removeAll	filter, null	difference	minView, delete	difference
retainAll	filter, null	intersection	filter, member	intersectionWith
toArray	toList	foldr	fold	foldrWithKey

In the context of a language with lazy evaluation such as `Haskell`, the operations that the benchmark suggests to iterate a given number of times need to be implemented carefully, in a way that ensures that the result of each iteration is fully evaluated (evaluated to normal form). Indeed, while the full evaluation of the final result can be ensured by the use of `Criterion` (with the *nf* function), if the intermediate ones are not demanded, the lazy evaluation machinery

avoids building them. This led us to use primitives such as *deepseq* [dee] in many definitions. We present an example in Listing 4.1, where we employed *deepseq* to iterate a number of times the *removeAll* operation for `Sequences`[1][2].

Listing 4.1 removeAll, benchmark operation implementation, for the Sequence abstraction.

```
{- Remove all elements, contained in t, from s. -}

removeAll :: S.Seq Int -> S.Seq Int -> S.Seq Int
removeAll s t = S.filter ( not . ( t `contains` ) ) s

{- Remove all elements, contained in t, from s. Repeat n times. -}

removeAllNTimes :: S.Seq Int -> S.Seq Int -> Int -> S.Seq Int
removeAllNTimes s _ 0 = s
removeAllNTimes s t n = deepseq ( removeAll s t ) ( removeAllNTimes s t ( n - 1 ) )
```

The *removeAll* function simply filters a `Sequence` *s*, maintaining in it, those elements that are not contained in a `Sequence` *t*. As for the "repeating" function, we make, in it's definition, use of the *deepseq* function. This function completely evaluates it's first parameter, and then, returns it's second parameter. It is used to guarantee that intermediate "results" are fully evaluated.
We have tried to follow as much as possible the data structure sizes and number of iterations suggested by the benchmark described in the previous chapter. In a few cases, however, we needed to simplify concrete operations for specific abstractions. This simplification was performed whenever a concrete operation failed to terminate within a 3 hours bound for a given implementation. In such cases, we repeatedly halved the size of the base data structure, starting at 100000, 50000 and so on. When the data structure size of 3125 was reached without the bound being met, we started reducing the number of iterations in half. With this principle in mind, no change was necessary for `Heaps` and `Sets`. For `Associative Collections` and `Sequences`, however, this was not the case. Table 4.2 lists the operations whose inputs or number of iterations were simplified. The underlined elements of this table are the ones that differ from those in the original benchmark.

Table 4.2: Modified Benchmark Operations.

abstraction	*iters*	*operation*	*base*	*elems*
Associative Collections	1	clear	50000	n.a.
	2500	remove	3125	1
	10	retainAll	25000	1000
	2500	toArray	3125	n.a.
Sequences	1	add	3125	25000
	625	containsAll	3125	1000

For the *containsAll* operation, relating to `Sequences`, even when reducing the amount of effort suggested by the benchmark, for one concrete implementation, *FingerSeq*, we were not able to obtain results in reasonable time. For this operation that implementation was discarded.

[1] The prefix, until the last ., before datatype and function names, means those entities are imported from an external `Haskell` module. In this case a module has been imported and qualified/renamed to *S*, e.g. **import** *qualified Data.Edison.Seq.BankersQueue as S*.

[2] Note, the use of the function composition function, the single ∘ (dot); $f \circ g$ means to apply f to the result of g applied to some argument.

In the following sections we describe some of the concrete implementations we devised for the benchmark operations, for each of the data structure abstractions.

4.1 Operations over Sequences

Most operations in the underlying benchmark have straightforward correspondences in the implementation functions provided by `Edison`. This is the case, for example, of the operation *add*, which can naturally be interpreted by functions *insert*, for `Heaps`, `Sets` and `Associative Collections`, as we have shown in Table 4.1. For `Sequences`, the underlying ordering notion allows two possible interpretations for adding an element to a sequence: in its beginning or in its end. In this case, we defined *add* as pictured in Listing 4.2, to alternatively use both interpretations.

Listing 4.2 Add, benchmark operation implementation, for the Sequence abstraction.

```
{- Add n, distinct, consecutive, elements, from m, to the Sequence seq. -}

add :: S.Seq Int -> Int -> Int -> S.Seq Int
add seq 0 _ = seq
add seq n m =
    let
        elemToAdd = m + n - 1
        nextNumber = n - 1
        cons = if even n then S.rcons else S.lcons
    in
        add ( elemToAdd `cons` seq ) nextNumber m
```

With that definition, *add s n m* inserts the *n* elements `{m+n-1, m+n-2, ..., m}` into *s*.
In Listing 4.3 the implementation of the *containsAll* benchmark operation, for `Sequences` is presented.

Listing 4.3 ContainsAll, benchmark operation implementation, for the Sequence abstraction.

```
{- Checks if a sequence s contains all elements in a sequence t. -}

containsAll :: S.Seq Int -> S.Seq Int -> Bool
containsAll s t = S.foldr (&&) True . S.map ( s `contains` ) $ t

{- Repeat n times, the containsAll check. -}

containsAllNTimes :: S.Seq Int -> S.Seq Int -> Int -> Bool
containsAllNTimes _ _ 0 = False
containsAllNTimes s t n =
    ( (||) ( containsAllNTimes s t ( n - 1 ) ) ) $!! ( s `containsAll` t )
```

As defined in Listing 4.3, *containsAll* transforms a `Sequence` *t* of *Int*s into a `Sequence` of *Bool*s where each value tells us if *s* contains a specific *Int* in *t* and then we fold that `Sequence` into a single *Bool* as the function result. This will be *True* if *s* contains all elements of *t* and *False* otherwise.
Also presented in that listing is the definition of the function that is executed by `Criterion` (Section 3.2), repeating the *containsAll* operation a number of times.

Note the use of the $!! operator which is but a different form of *deepseq*. It fully evaluates it's second parameter before applying to it, the function provided as the first parameter. Because $!! is being used as an infix function (an operator) the first parameter is the portion of the expression to it's left side, and the second parameter, the portion to it's right.
In Listing 4.4 we present the *retainAll* benchmark operation implementation for `Sequences`.

Listing 4.4 RetainAll, benchmark operation implementation, for the Sequence abstraction.

```
{- Retain all elements contained in sequence t, in sequence s. -}

retainAll :: S.Seq Int -> S.Seq Int -> S.Seq Int
retainAll s t = S.filter ( t `contains`) s

-- The retainAllNTimes function is "identical" to the removeAllNTimes
```

The implementation of the *retainAll* operation for that data structure abstraction is very similar to the implementation of the *removeAll* operation for the same abstraction. Actualy it is the "inverse" of it, the `Sequence` *s* is filtered, maintaining in it the elements contained in a `Sequence` *t*.
The *retainAllNTimes* function which runs the *retainAll* function a number of times is identical to the *removeAllNTimes* function already presented (in Listing 4.1).

4.2 Operations over Collections

In this section, we describe some of our implementations for the benchmark operations over `Collections`. These implementations are divided across two sections, one for `Heaps` (section 4.2.1) and one for `Sets` (section 4.2.2).

4.2.1 Operations over Heaps

In Listing 4.5 the *containsAll* benchmark operation implementation, for `Heaps` is presented.

Listing 4.5 containsAll, benchmark operation implementation, for the Collection abstraction, for Heaps.

```
{- Checks if a Heap h contains all elements in a Heap i. -}

containsAll :: H.Heap Int -> H.Heap Int -> Bool
containsAll h i
    | H.null h = H.null i
    | H.null i = True
    | otherwise =
        case H.minView i of
            Just ( m , roi ) -> if ( h `contains` m ) then
                                    ( containsAll h roi ) else False
            Nothing -> True

-- The containsAllNTimes function is "identical" to the same function for
-- Sequences.
```

The *containsAll* function checks for some boundary cases, and tries to separate from a heap *i* it's minimum value. If all the values it can extract (recursively) are contained in a heap *h* then the result must be *True*, otherwise, that is if any value from *i* is not contained in *h*, then the result will be *False*.
The only difference in the "repeating" function to the corresponding function for `Sequences` is it's type, *Heap* instead of *Seq*. Therefore this function is not presented.
In Listing 4.6 the *add* benchmark operation implementation, for `Heaps` is presented.

Listing 4.6 add, benchmark operation implementation, for the Collection abstraction, for Heaps.

```
{- Add n, distinct, consecutive elements, from m, to the Heap h. -}

add :: H.Heap Int -> Int -> Int -> H.Heap Int
add h 0 _ = h
add h n m = add ( H.insert ( m + n - 1 ) h ) ( n - 1 ) m
```

The `Collections` data structure abstraction already provides a function to insert one element into a `Heap`, *insert*. We define the *add* function/operation recursively to, using *insert*, add the benchmark prescribed number of elements to a `Heap`.
In Listing 4.7 the *addAll* benchmark operation implementation, for `Heaps` is presented[3].

Listing 4.7 addAll, benchmark operation implementation, for the Collection abstraction, for Heaps.

```
{- Add all elements contained in Heap i, to the Heap h. -}

addAll :: H.Heap Int -> H.Heap Int -> H.Heap Int
addAll = H.union

-- The addAllNTimes function is "identical" to the same function for Sequences.
```

The *addAll* operation can readily be defined using the *union* function, provided by Edison, again showing that Edison provides straightforward interpretations for the operations of the benchmark we are using.

4.2.2 Operations over Sets

In Listing 4.8 the *containsAll* benchmark operation implementation, for `Sets` is presented.
As is observable, for `Sets`, the *containsAll* benchmark operation directly corresponds to a single "native" function from the Edison library, which checks is set *t* is a subset of a set *s*.
The only difference in the "repeating" function to the corresponding function for `Sequences` or `Heaps` is it's type, *Set* instead of *Seq* or *Heap*. Therefore this function is not presented.
In Listing 4.9 the *contains* benchmark operation implementation, for `Sets` is presented.
Checking if a `Set` *contains* an element amounts to checking for `Set` membership with the *member* function. But since the order of parameters for *contains* and *member* is reversed we have to

[3]The *addAll* function in this example is defined in a pointfree manner/notation in which the same parameters on the left and right-hand sides are omitted; that definition is equivalent to *addAll h i = H.union h i*

Listing 4.8 containsAll, benchmark operation implementation, for the Collection abstraction, for Sets.

```
{- Checks if a Set s contains all elements in a Set t. -}

containsAll :: S.Set Int -> S.Set Int -> Bool
containsAll s t = S.subset t s

-- The containsAllNTimes function is "identical" to the same function for
-- Sequences and Heaps.
```

Listing 4.9 contains, benchmark operation implementation, for the Collection abstraction, for Sets.

```
{- Check if a Set s contains an element e. -}

contains :: S.Set Int -> Int -> Bool
contains = flip S.member

-- The containsNTimes function is "identical" to the containsAllNTimes
-- function for Sequences or Heaps.
```

use the flip function[4].
In Listing 4.10 the *toArray* benchmark operation implementation, for `Sets` is presented.

Listing 4.10 toArray, benchmark operation implementation, for the Collection abstraction, for Sets.

```
{- Convert a Set into a Array. -}

toArray :: S.Set Int -> [ Int ]
toArray = S.foldr (:) []

-- The toArrayNTimes function follows the same pattern as the
-- removeAllNTimes for Sequences.
```

The *toArray* operation implementation folds over a `Set` adding elements to a list/array. If the `Set` is empty, so will be the list, if not, each of the `Set`'s elements will be added to the list.

4.3 Operations over Associative Collections

In Listing 4.11 the *containsAll* benchmark operation implementation, for `Associative Collections` is presented.
Once again the *containsAll* operation can be directly mapped to a single Edison API function (using flip as a "translation" function).
As before, the *containsAllNTimes* function is omitted because it is identical to the same function for the other data structure abstractions.
In Listing 4.12 the *add* benchmark operation implementation, for `Associative Collections` is presented.

[4] If f has type $a \to b$ then *flip* f has type $b \to a$.

Listing 4.11 containsAll, benchmark operation implementation, for the Associative Collection abstraction.

```
{- Check if a associative collection a contains all elements in a
   associative collection b. -}

containsAll :: A.FM Key Datum -> A.FM Key Datum -> Bool
containsAll = flip A.submap

-- The containsAllNTimes function is "identical" to the same function for
-- Sequences and Collections (Heaps and Sets).
```

Listing 4.12 add, benchmark operation implementation, for the Associative Collection abstraction.

```
{- Add n, distinct, consecutive, elements, from m, to an associative collection. -}

add :: A.FM Key Datum -> Int -> Datum -> A.FM Key Datum
add a 0 _ = a
add a n m = add ( A.insert ( m + n - 1 ) ( m + n - 1 ) a ) ( n - 1 ) m
```

The `Associative Collections` data structure abstraction, like `Collections`, already provides a function to insert one element into a `Associative Collection`, *insert*. We define the *add* function/operation in a similar fashion as for `Heaps`.
In Listing 4.13 the *iterator* benchmark operation implementation, for `Associative Collections` is presented.

Listing 4.13 iterator, benchmark operation implementation, for the Associative Collection abstraction.

```
{- Iterate through a associative collection. -}

iterator :: A.FM Key Datum -> A.FM Key Datum
iterator = A.map id
```

Our *iterator* benchmark operation implementation, traverses an `Associative Collection` maintaining all the elements in place, with the identity function *id*[5].

4.4 Final Remarks

In this chapter we described the methodology employed in performing our study.
We put forward "our interpretation" of the benchmark described in Section 3.1 and presented some of the implementations for the operations prescribed by that benchmark. For the sake of simplicity, most implementations, however, are not shown in this document. The interested reader may find all the source code implementing the benchmark for our study, available through: `https://github.com/green-haskell/edison-benchmark`.
In the next chapter we present the results obtained from our study.

[5]The identity function is such that $id\ a = a$

Chapter 5

Results

In this thesis, we have been describing various aspects of the study we set out to accomplish. In this chapter, we will now present the results we obtained, following the methodology described in the previous chapter. In Section 5.1 we present the results for the `Sequence` abstraction. The `Collections` abstraction results are presented in Section 5.2. That section is subdivided into two sections, one for `Heaps` (Section 5.2.1) and one for `Sets` (Section 5.2.2). Finally in Section 5.3 the results for `Associative Collections` are presented.

As was the case for the benchmark operations implementations, in the previous chapter, not all of the observed results for all operations on all abstractions are included here. They are available at the companion website, at green-haskell.github.io.

5.1 Sequences

In Figure 5.1 we present the results obtained for the *add* benchmark operation for `Sequences`.

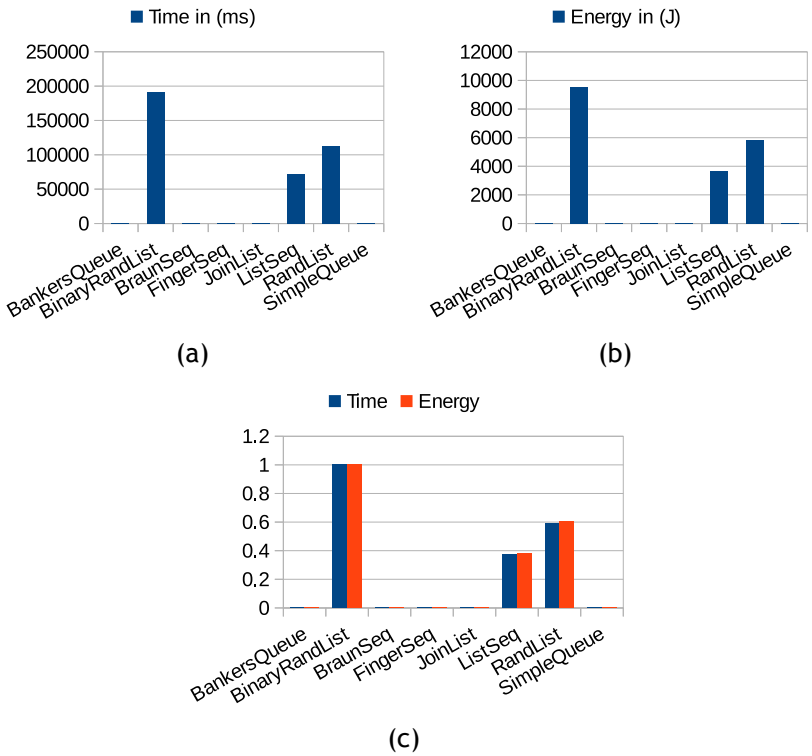


Figure 5.1: Results for the add operation for Sequences.

In that figure we present three graphics, (a) and (b) for the absolute values for time and energy consumption (as measured by Criterion) respectively, and (c) showing the proportions of the maximum time (the blue bars) and energy consumption (the orange bars), respectively. This

is also true for all of the other graphics shown in this chapter, for the other data structure abstractions.
So in Figure 5.1 we can see that, for the add operation, the worst performing implementation was the *BinaryRandList* implementation followed by the *RandList* and *ListSeq* implementations. More specifically, the *BinaryRandList* implementation add operation, took almost 200000 milliseconds (ms) to execute, and consumed almost 10000 Joules (J) of energy. This corresponds to 1 (100%) in the proportions graphic, (c). As for *ListSeq*, it ran in approximately 75000 ms an consumed almost 4000 J of energy. This corresponds, in graphic (c) (the proportions graphic), to approximately 40% (0.4) of the maximum.
One can also easily gather from the graphics that there is a significant difference in efficiency, not only among the three most inefficient implementations (a maximum difference just above 60%), but mostly between those and the five other, better performing, implementations.
In graphic (c) portrayed in Figure 5.1, the differences between the least consuming implementations are not easily discerned. Figure 5.2 shows the same "dataset", omitting the three most consuming implementations, to make the relations between the more efficient implementations clearer.

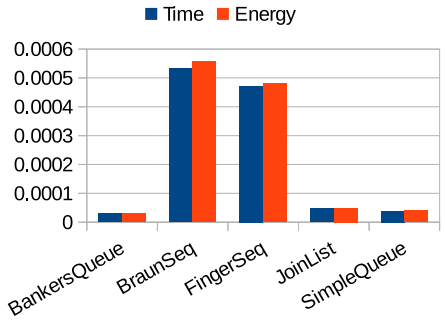


Figure 5.2: Results for the add operation for Sequences, omitting the three most consuming implementations.

We can see that the most efficient implementations, for the *add* operation, for `Sequences`, take a very low percentage of the time/energy, taken by the most consuming implementation, to realize the same work. Also, even among the most efficient there are clear differences.
As a validation of our empirical study, in Table 5.1 we show the `Sequences` implementations ordered from most efficient to least efficient, for the *add* operation, together with the asymptotic complexities (as given in the `Edison` documentation) for the two `Edison` functions used in defining the *add* operation.

Table 5.1: Asymptotic time complexities, for the lcons and rcons functions, used in the add operation definition, for Sequences.

	lcons	rcons
BankersQueue	O(1)	O(1)
SimpleQueue	O(1)	O(1)
JoinList	O(1)	O(1)
FingerSeq	O(1)	O(1)
BraunSeq	O($\log n$)	O($\log^2 n$)
ListSeq	O(1)	O(n)
RandList	O(1)	O(n)
BinaryRandList	O($\log n$)	O($n \log n$)

We can see in that table, that the asymptotic complexities match up with the order obtained

by our experimentation.

In Figure 5.3 we present the results obtained for the *containsAll* benchmark operation for `Sequences`.

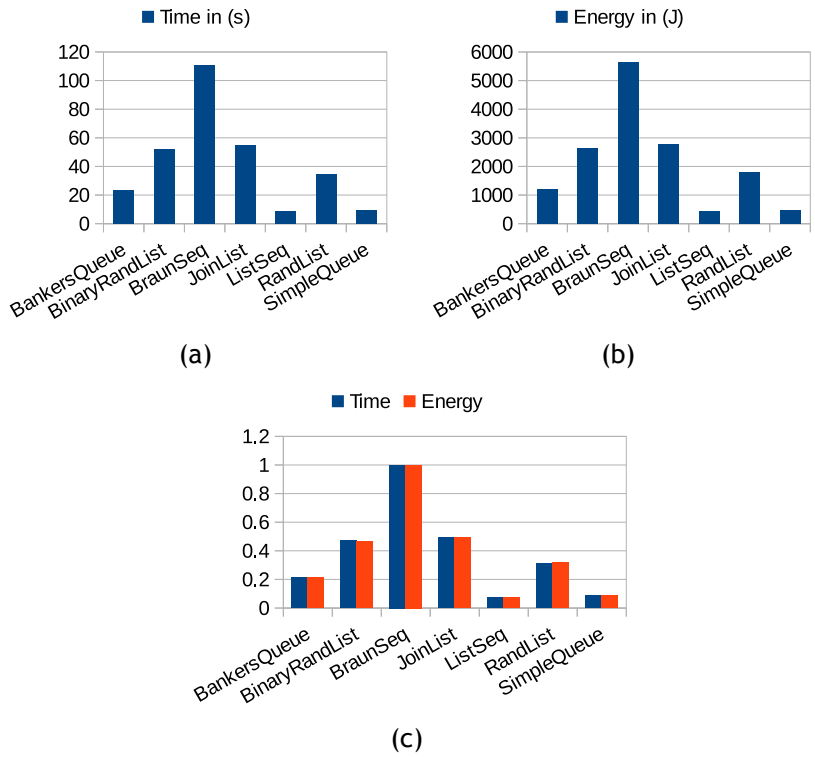


Figure 5.3: Results for the containsAll operation for Sequences.

Note that, for this operation, for the *FingerSeq* implementation, we were unable to get results in a timely manner. Therefore that implementation was discarded from the graphics.

As for the results we do have, there is a clear worst and best performer. The most consuming implementation was the *BraunSeq* implementation. The least consuming was the *ListSeq* implementation. The difference between them was about 93%. The "intermediate" implementations situate themselves in between 8% and 50% of the maximum.

As can be percieved from the graphics, the energy consumption is closely tied to the execution time.

In Figure 5.4 we present the results obtained for the *retainAll* benchmark operation for `Sequences`.

In that figure the *FingerSeq* implementation stands out as the worst performer. The runner-up for that position was the *BraunSeq* implementation, which nonetheless took less than 10% of the resources to complete it's work. The most efficient sequence implementation for the *retainAll* operation was the *ListSeq* implementation.

The results obtained for `Sequences` show that execution time strongly influences energy consumption.

5.2 Collections

In this section the results for the `Collections` data structure abstraction will be presented, in Section 5.2.1 for `Heaps` and in Section 5.2.2 for `Sets`.

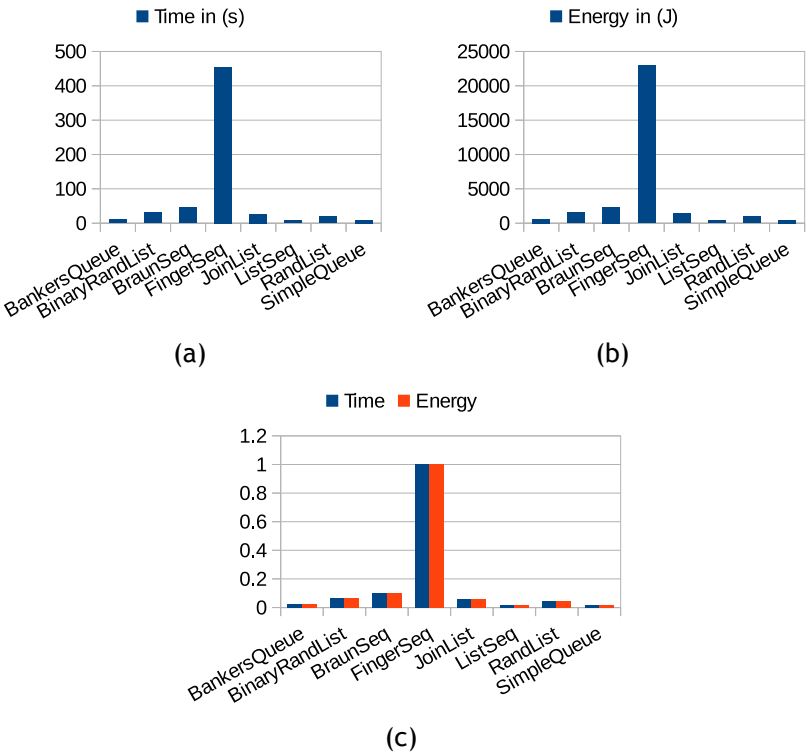


Figure 5.4: Results for the retainAll operation for Sequences.

5.2.1 Heaps

In Figure 5.5 we present the results obtained for the *add* benchmark operation for `Heaps`.

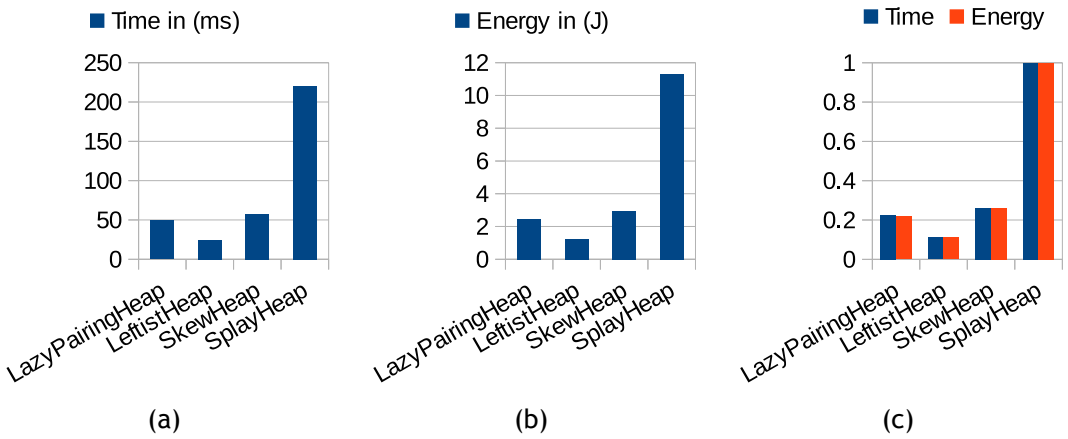


Figure 5.5: Results for the add operation for Heaps.

As can be observed the energy consumption mirrors the execution time. The *LeftistHeap* implementation was the most efficient for this operation, while the *SplayHeap* was the least efficient one. The difference between the two was about 90%. For this operation, for all the implementations the proportions of energy consumption and execution time differ in at most 0.57%.
In Figure 5.6 we present the results obtained for the *containsAll* benchmark operation for `Heaps`. Once again the energy consumption closely "follows" the execution time. The most efficient implementation was the *LazyPairingHeap*, followed by *SkewHeap*, *LeftistHeap*, and the least efficient, was again the *SplayHeap*. The difference between the extreme performers for this

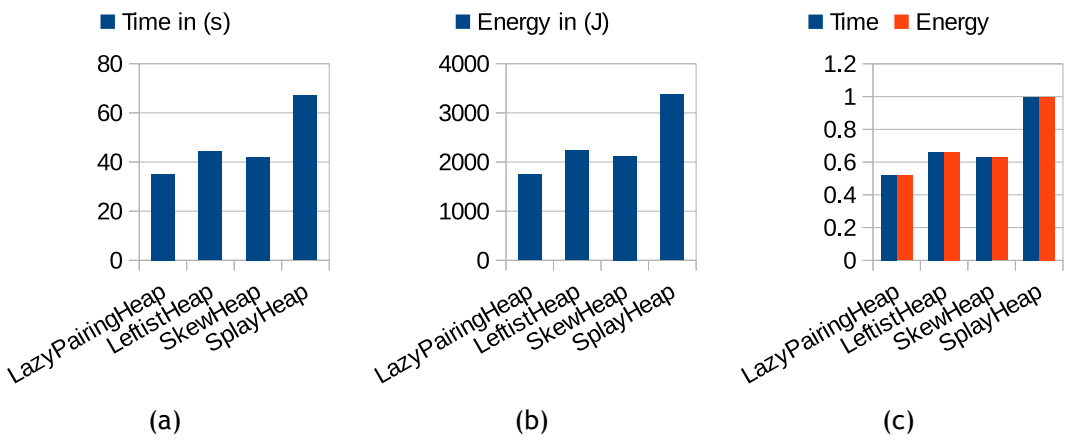


Figure 5.6: Results for the containsAll operation for Heaps.

operation is not as pronounced, as for the *add* operation, weighin in at about 48%. For this operation, for all the implementations the proportions of energy consumption and execution time differ in at most 0.52%.

Actually the pattern of efficiency witnessed for *containsAll* was also observed for three other operations, *clear*, *contains* and *retainAll* (whose concrete results are omitted here).

In Figure 5.7 we present the results obtained for the *addAll* benchmark operation for `Heaps`.

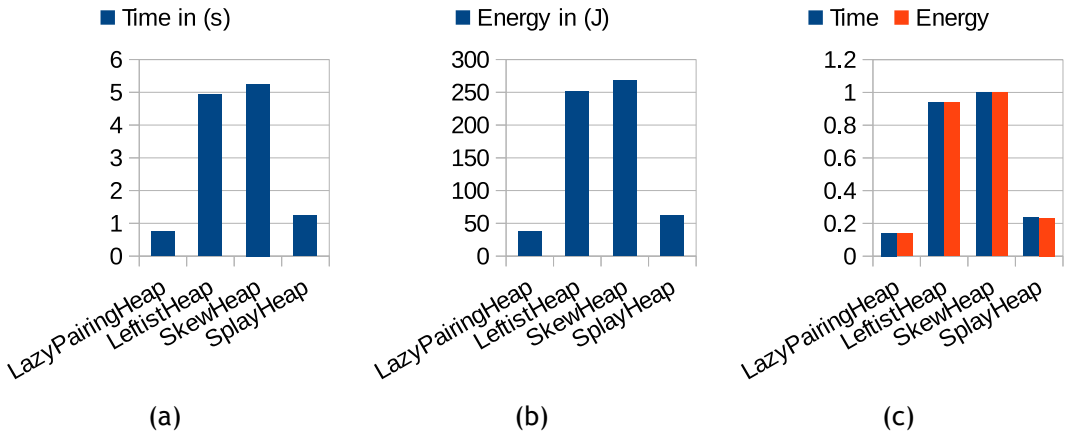


Figure 5.7: Results for the addAll operation for Heaps.

For the *addAll* operation the most efficient implementation was the *LazyPairingHeap*, followed by *SplayHeap*, *LeftistHeap* and finally *SkewHeap*. The differences between the most, and the least, efficient implementations is sizeable, at about 85%. This pattern is also observable for four other operations, *iterator*, *remove*, *removeAll* and *toArray*.

For this data structure abstraction our experiments suggest that energy consumption is proportional to execution time.

Overall, the *LazyPairingHeap* implementation was observed to be the most efficient in all benchmark operations except for *add*; the *SkewHeap* and *SplayHeap* implementations were the least efficient in 5 operations each; and the *LeftistHeap* implementation was consistently the second to last performer (with a single exception being the *add* operation, for which it was the most efficient implementation).

The proportions of runtime and energy consumption differ in at most 2.16% for any operation and implementation of `Heaps`.

Significant differences (a maximum of 90%) were observed for the execution time and energy consumption, among the different implementations. This indicates that there are opportunities for savings to be achieved.

5.2.2 Sets

In Figure 5.8 we present the results obtained for the *containAll* benchmark operation for `Sets`.

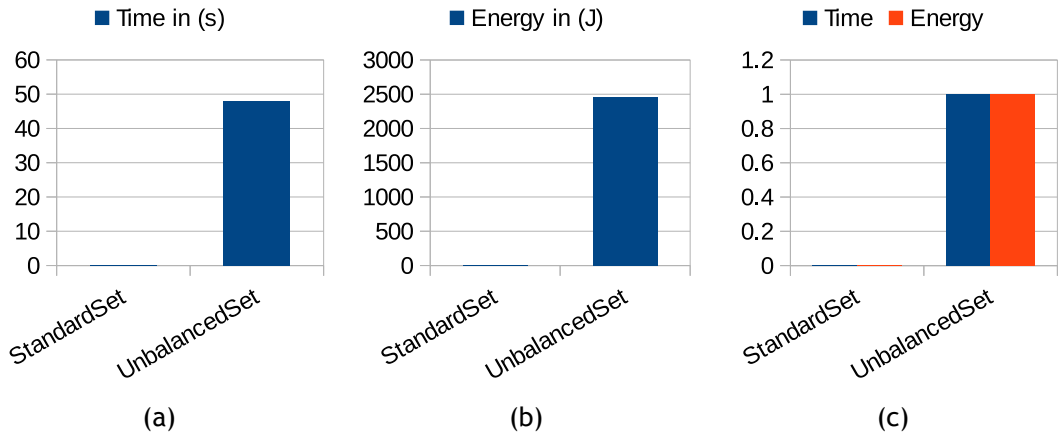


Figure 5.8: Results for the containsAll operation for Sets.

Observable in the graphics in that figure, is an enormous difference in performance between the implementations considered. The *StandardSet* implementation being the most efficient and, *UnbalancedSet* the least efficient. Also easily discernable is the fact that the energy consumption accompanies the execution time.
In Figure 5.9 we present the results obtained for the *toArray* benchmark operation for `Sets`.

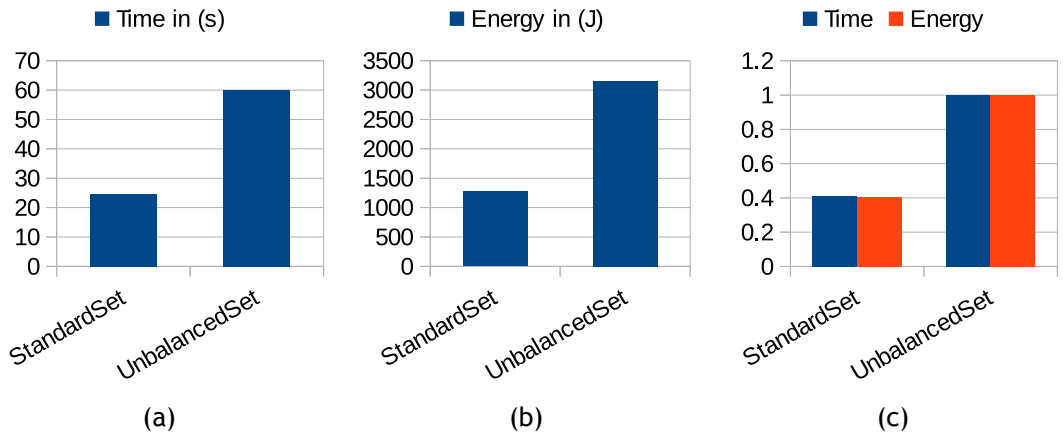


Figure 5.9: Results for the toArray operation for Sets.

Once again, the *StandardSet* implementation was more efficient than the *UnbalancedSet* one, and the energy consumption mirrors the execution time. However the difference between the implementations is now less pronounced (about 60%).
The previous results actually form a pattern verified in all but one of the operations. We present the results obtained for that operation next.
In Figure 5.10 we present the results obtained for the *contains* benchmark operation for `Sets`.

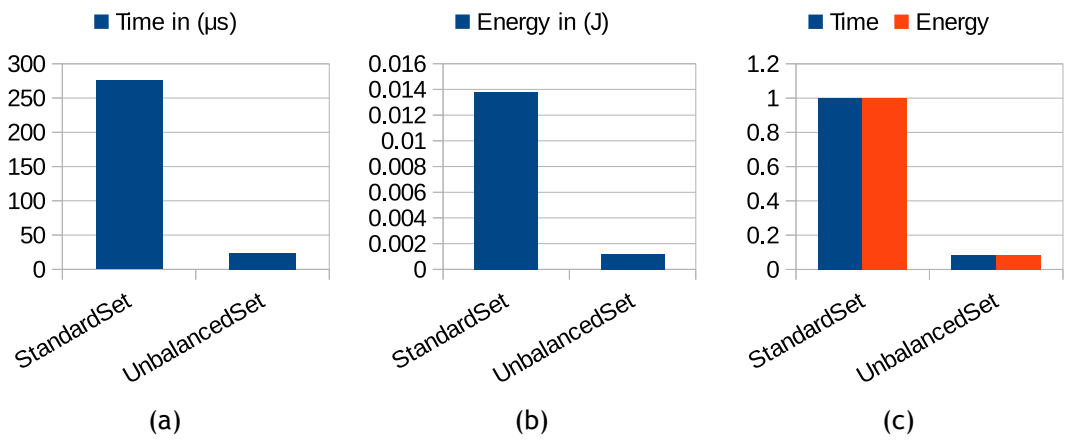


Figure 5.10: Results for the contains operation for Sets.

From the graphic in that figure, for the *contains* operation, apparently the *StandardSet* implementation is worst performing than the *UnbalancedSet* implementation. As mentioned, this differs from the rest of the operations in the benchmark for `Sets`. This led us to investigate the causes.

As defined in Listing 4.9 our *contains* operation implementation uses the *member* function, natively present in `Edison`. For the *StandardSet* implementation, the documentation for *Data.Set*, the underlying implementation, tells us that the asymptotic complexity for that operation is O(log n). For the *UnbalancedSet*, the lack of documentation meant we had to look at the source code.

In Listing 5.1 we can see the definition of the *member* function for the *UnbalancedSet* implementation (for `Sets`).

Listing 5.1 member, function definition for the UnbalancedSet implementation, for the Collection abstraction, for Sets.

```
member _ E = False
member x (T a y b) =
    case compare x y of
        LT -> member x a
        EQ -> True
        GT -> member x b
```

As stated in Section 2.2.2.3, in this implementation a binary search tree order is maintained. If the tree was also a balanced tree, then the complexity of that *member* function would also be O(log n). But this implementation does not keep the tree balanced.

Also, by decisions taken earlier in the development process, the way elements are added to the base data structure, makes it so that, that base will actually be a completely unbalanced tree (akin to a list). It will have it's 100000 elements hanging on the left subtree while the right subtree will be empty. This toghether with the fact that the *contains* operation will look in a tree for an element a lot bigger than the elements in that tree (another design decision relating to the strictness of functions) leads to a premature ending of that computation.

This realization suggests us that we should have chosen a different strategy for inserting the elements in the base structure. For example, to keep a list of random values in a file and load it to create environments for the benchmarks, as this list would need to be equal for all

implementations.

5.3 Associative Collections

In Figure 5.11 we present the results obtained for the *containsAll* benchmark operation for `Associative Collections`.

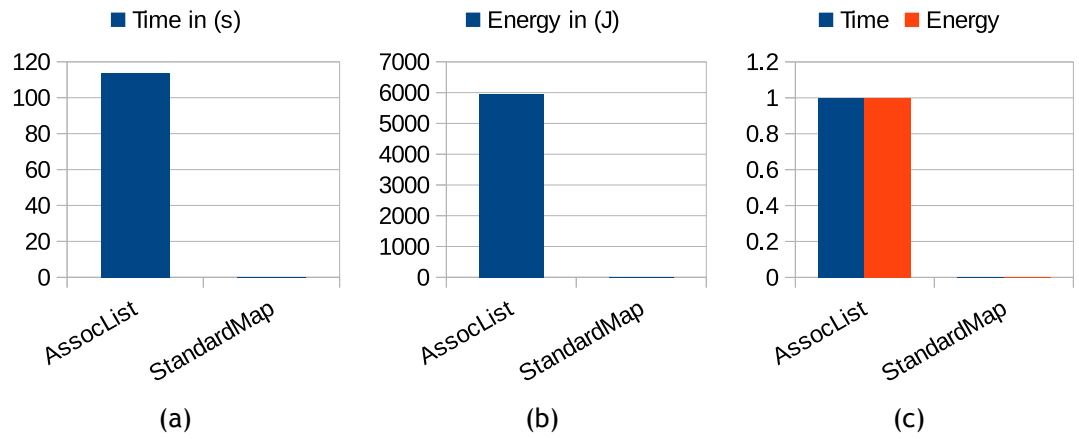


Figure 5.11: Results for the containsAll operation for Associative Collections.

As can be seen from the graphics in that figure the energy consumption is proportional to the execution time. This was true for all the `Associative Collections` implementations and for all operations, as the reader will be able to see from the graphics to come.
Also the *StandardMap* implementation was, for this operation (and for 7 out of 10 operations), immensely more efficient than the *AssocList* implementation.
Only for the *addAll* operation was the *StandardMap* implementation time/energy "expenditure" similar to that of the *AssocList* implementation; the difference being of about 9% in favor of *StandardMap*.
As stated the *StandardMap* implementation was for most cases more efficient than the *AssocList* implementation. There were, however, two operations for which the results showed an inverse relation. These were the *add* and *iterator* operations.
In Figure 5.12 we present the results obtained for the *add* benchmark operation for `Associative Collections`.
As can be observed from the graphics in that figure the efficiency relation between *AssocList* and *StandardMap* is reversed. *AssocList* was approximately 40% more efficient than *StandardMap*.
One may wonder why that is so. In order to try and understand the reason for this inverse relation we turned to the code and documentation of those implementations.
As already shown in Listing 4.12 the *add* operation for `Associative Collections` is implemented using the *insert* native `Edison` function.
The *StandardMap* implementation is based on the *Data.Map* standard `Haskell` library. For this library, asymptotic time complexities are provided in the documentation. For the *insert* native function the listed complexity is O(log n).
For the *AssocList* implementation no asymptotic time complexities are given, so we had to turn to the code implementing it. In Listing 5.2 the definition of the *insert* function for the *AssocList* implementation is shown.

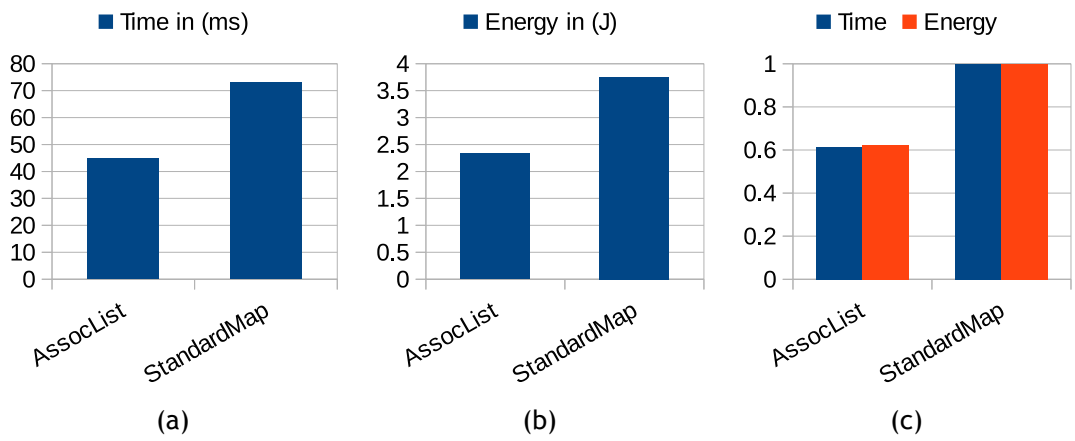


Figure 5.12: Results for the add operation for Associative Collections.

Listing 5.2 insert, function definition, for the AssocList implementation of the Associative Collections abstraction.

```
insert = I
```

As can be seen this function just, resorting to the *I* data constructor (see Section 2.3.2), puts, whatever it's arguments are, an key/value "pair" at the front of the list. This has O(1) complexity.

This, of course, can explain the inverted relation between the two implementations. Constant time outperforms logarithmic time.

The other operation for which the results showed an inverse relation was the *iterator* operation.

In Figure 5.13 we present the results obtained for the *iterator* benchmark operation for `Associative Collections`.

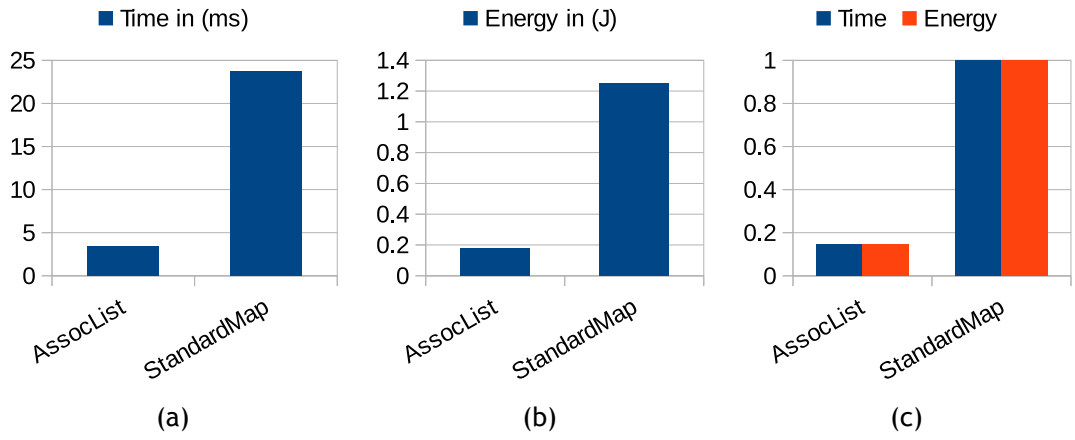


Figure 5.13: Results for the iterator operation for Associative Collections.

Once again we investigated why, for that operation, the *AssocList* implementation consumes less than 20% of the maximum time/energy consumption, by the *StandardMap* implementation.

As presented in Listing 4.13 the *iterator* benchmark operation, for `Associative Collections`, was implemented using the `Edison` native *map* function.

For the *StandardMap* implementation, the asymptotic complexity "listed" in the documentation for the underlying implementation (*Data.Map*) is O(n), that is, linear time.

For the *AssocList* implementation, we had to resort to code inspection. In Listing 5.3 we show the definition of the *map* function for the *AssocList* implementation of the `Associative Collections` abstraction.

Listing 5.3 map, function definition, for the AssocList implementation of the Associative Collections abstraction.

```
map _ E = E
map f (I k x m) = I k (f x) (map f m)
```

As is apparent what that function does is traverse a list end-to-end. This also runs in linear time. Although theoretically both implementations work in linear time, our results indicate that *StandardMap* takes substantially more time than *AssocList*. We haven't yet been able to pinpoint why this is so, but this shows that studies like this one are important in assessing the characteristics of software libraries.

5.4 Final Remarks

In this chapter, we presented the results obtained from our experiments. We have also identified a few glitches in our initial thoughts that may have created some biases in the observed results. Nonetheless, a strong pattern has emerged. And this is that: for the use of the library studied, `Edison`, savings are possible, if different abstraction implementations are employed, according to the different operation usage patterns particular to each application.

Chapter 6

Conclusions

As energy efficiency becomes a popular concern for software developers, we must be aware of the implications of our development decisions in our applications energy footprint. In this work, we analyzed a subset of those decisions for a purely functional programming language, `Haskell`. We started, in Chapter 2, by introducing a `Haskell` software library, `Edison`, that provides a number of different implementations for three data structure abstractions (`Sequences`, `Collections` and `Associative Collections`). Those implementations were listed and organized by data structure abstraction.
Then, in Chapter 3, we reported on our experimental setting. Starting with the depiction of the benchmark utilized to conduct our study, in Section 3.1; then with a illustration/exemplification of the `Criterion` benchmarking library/tool (Section 3.2), which we used to run the benchmark and gather the required measurements; followed by a definition of RAPL (Section 3.3); and ending with the presentation of the underlying test-bed.
In Chapter 4 we elaborated on the methodology we followed in performing our work. We identifyed some "bottlenecks" that meant that some implementations had to be disregarded. Of all the implementations for the three data structure abstractions, *SizedSeq*, *RevSeq* (for `Sequences`) and *MinHeap* (for `Heaps`) were not considered because they are adapters for other implementations; *EnumSet* (for `Sets`) was not considered because it can only model small sets; *PatriciaLoMap* and *TernaryTrie* (for `Associative Collections`) were discarded because they are not totally compatible with the `Associative Collections` API; *MyersStack* (for `Sequences`) was also removed because it could not be fully evaluated in a timely manner (for that same reason, *FingerSeq* for `Sequences`, was excluded from the *containsAll* operation). Still in Chapter 4 we presented our interpretation of the benchmark utilized and exhibited some of our `Haskell` realizations of the benchmark operations.
Finally, in Chapter 5 we presented the results (measurements) obtained from the executions of the benchmark.
Our study was driven by the two following, concrete, research questions:

RQ1. How do different implementations of the same abstractions compare in terms of runtime and energy efficiency?

RQ2. For concrete operations, what is the relationship between their performance and their energy consumption?

The answers to those questions are:

RQ1.: The comparisons between different implementations of the same abstractions are observable from a number of the proportions graphics presented in the Results chapter (Chapter 5) and available in their entirety in the companion site, green-haskell.github.io. As a summary, we have observed that, for:

`Sequences` the overall most and least efficient implementations were *ListSeq* and *FingerSeq* respectively;

`Heaps` the same is true of the *LazyPairingHeap* and (tied) the *SkewHeap*/*SplayHeap*;

`Sets` the *StandardSet* implementation is generally more efficient than *UnbalancedSet*;

`Associative Collections` the *StandardMap* implementation is generally more efficient than *AssocList*.

RQ2.: For all operations, across all data structure abstractions, the energy consumption is directly proportional to the execution time.

As to a more general research question:

RQ. To what extent can we save energy by refactoring existing `Haskell` programs to use different data structure implementations?

The answer is: Yes we can save (probably a lot of) energy. Those savings vary between, approximately: 38% and 99% for `Sequences`; 2% and 90% for `Heaps`; 60% and 99% for `Sets`; and, 10% and 99% for `Associative Collections`.
We found that for the different data structure implementations available in the `Edison` library (Chapter 2): for particular abstractions implementations and for particular operations, the differences in performance can be very significant, as can be observed in the graphics presented in the Results chapter (Chapter 5). This means, of course, that developers have thus, an opportunity to make an informed exchange of one implementation with some other, for a particular usage pattern, and immediately reap the benefits.
The work described in this thesis consisted in one of the contributions of a published research paper:

- **"Haskell in green land: Analyzing the energy behavior of a purely functional language."**, Luís Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and João Paulo Fernandes, in the proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) 2016, volume 1, pages 517–528.

6.1 Future Work

This work is one study of a particular piece of software, the `Edison` library. Improvements and corrections can of course be made. Also other "base" libraries can be studied to help developers pick the more energy efficient ones.
We are currently preparing to study the behaviour of the different implementations regarding performance "evolution" with input size variation.

Bibliography

[BBV09] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 280–293. ACM, 2009. Available from: http://doi.acm.org/10.1145/1644893.1644927. 1

[Ca14] T Carção. Spectrum-based energy leak localization. Master's thesis, University of Minho, Portugal, 2014. 19

[Cou14] Tracy Counts. Running average power limit - rapl, 2014. Available from: https://01.org/blogs/tlcounts/2014/running-average-power-limit-%E2%80%93-rapl. 19

[CSB92] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power cmos digital design. *Solid-State Circuits, IEEE Journal of*, 27(4):473–484, Apr 1992. 1

[DD09] Srini Devadas and Konstantinos Daskalakis. Balanced binary search trees, 2009. Available from: http://courses.csail.mit.edu/6.006/fall09/lecture_notes/lecture04.pdf. 9

[dee] deepseq package. Available from: http://hackage.haskell.org/package/deepseq. 26

[DGH+10] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194. IEEE, 2010. 20, 22

[Doca] Robert Dockins. Edison, Haskell Communities and Activities Report 2009. Available from: https://www.haskell.org/communities/05-2009/html/report.html. 2

[Docb] Robert Dockins. Edisonapi package. Available from: http://hackage.haskell.org/package/EdisonAPI-1.3. 7

[Docc] Robert Dockins. Edisoncore package. Available from: http://hackage.haskell.org/package/EdisonCore-1.3. 7

[Eri15] Ericsson. Ericsson mobility report, 2015. [Online; accessed 04-October-2016]. Available from: https://www.ericsson.com/news/1925907. 1

[FFMB11] Albrecht Fehske, Gerhard Fettweis, Jens Malmodin, and Gergely Biczok. The global footprint of mobile communications: The ecological and economic perspective. *IEEE Communications Magazine*, 49(8):55–62, 2011. 1

[FZ08] Gerhard Fettweis and Ernesto Zimmermann. Ict energy consumption-trends and challenges. In *Proceedings of the 11th International Symposium on Wireless Personal Multimedia Communications*, volume 2, page 6, 2008. 1

[Gui14] John Guillebaud. There are not enough resources to support the world's population, 2014. [Online; accessed 04-October-2016]. Available from: http://www.abc.net.au/radionational/programs/ockhamsrazor/there-are-not-enough-resources-to-support-the-world's-population/5511900. 1

[HA09] Robert R Harmon and Nora Auseklis. Sustainable it services: Assessing the impact of green computing practices. In *PICMET'09-2009 Portland International Conference on Management of Engineering & Technology*, pages 1707–1717. IEEE, 2009. 1

[HDVH12] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012. 22

[HP06] Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006. 10

[J.95] Gu J. Considerations on some problems of population, resources, and environment., 1995. [Online; accessed 04-October-2016]. Available from: https://www.ncbi.nlm.nih.gov/pubmed/12290013. 1

[KL10] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010. 1

[KLGT09] Mikkel Baun Kjærgaard, Jakob Langdal, Torben Godsk, and Thomas Toftkjær. Entracked: Energy-efficient robust position tracking for mobile devices. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 221–234. ACM, 2009. Available from: http://doi.acm.org/10.1145/1555816.1555839. 1

[KoC15] Shriram Krishnamurthi and The Staff of CSCI 0190. Joinlists, 2015. Available from: https://cs.brown.edu/courses/cs019/2015/Assignments/join-lists.html. 11

[Lew11] Leo Lewis. Java collection performance, 2011. Available from: http://dzone.com/articles/java-collection-performance. 2, 19

[Lim16] Luís Gabriel Nunes Ferreira Lima. Understanding the energy behavior of concurrent haskell programs. Master's thesis, Federal University of Pernambuco, 2016. 2, 20

[LPL15] Kenan Liu, Gustavo Pinto, and YuDavid Liu. Data-oriented characterization of application-level energy optimization. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*, volume 9033 of *Lecture Notes in Computer Science*, pages 316–331, 2015. 1

[MPC14] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514. ACM, 2014. 19

[Mye83] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, pages 241–248, 1983. 14

[NR72] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72,

pages 137–142. ACM, 1972. Available from: http://doi.acm.org/10.1145/800152.804906. 16, 17

[OG98] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998. 17

[Oka95a] Chris Okasaki. Purely functional random-access lists. In *Functional Programming Languages and Computer Architecture*, pages 86–95. ACM Press, 1995. 12

[Oka95b] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5:583–592, 10 1995. 13

[Oka99] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. 7, 12, 13, 15

[Oka01] Chris Okasaki. An overview of edison. *Electronic Notes in Theoretical Computer Science*, 41(1):60–73, 2001. 2, 7

[O'S09] Bryan O'Sullivan. criterion: Robust, reliable performance measurement and analysis, 2009. Available from: http://www.serpentine.com/criterion/. 20

[PCL14a] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 22–31, New York, NY, USA, 2014. ACM. Available from: http://doi.acm.org/10.1145/2597073.2597110. 1

[PCL14b] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 345–360, New York, NY, USA, 2014. ACM. 1

[PCS+16] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, pages 15–21. ACM, 2016. Available from: http://doi.acm.org/10.1145/2896967.2896968. 19

[PLCL16] Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. A comprehensive study on the energy efficiency of java thread-safe collections. *Journal of Systems and Software*, 2016. To appear. 19

[RNA+12] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, (2):20–27, 2012. 22

[SFKW+] Heinz Schandl, Marina Fischer-Kowalski, James Wes, Stefan Giljum, Monika Dittrich, Nina Eisenmenger, Arne Geschke, and Mirko Lieber. Global material flows and resource productivity. [Online; accessed 04-October-2016]. Available from: http://unep.org/documents/irp/16-00169_LW_GlobalMaterialFlowsUNEReport_FINAL_160701.pdf. 1

[She09] Tim Sheard. Full v.s. complete binary trees, 2009. Available from: http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/FullvsComplete.html. 12

[Sof15] Cluster Green Software, 2015. [Online; accessed 03-October-2016]. Available from: http://www.clustergreensoftware.nl/english/. 1

[SPC14] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 36:1–36:10, 2014. 1

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985. 15

[ST86] Daniel Dominic Sleator and Robert Endre Tarjan. Self adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, February 1986. Available from: http://dx.doi.org/10.1137/0215004. 15

[Sta14] Statista. Number of smartphone users worldwide from 2014 to 2019, 2014. [Online; accessed 04-October-2016]. Available from: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. 1

[STM+14] Cagri Sahin, Philip Tornquist, Ryan Mckenna, Zachary Pearson, and James Clause. How does code obfuscation impact energy usage? In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 131–140, 2014. 1

[SW14a] Robert Sedgewick and Kevin Wayne. Binary search trees, 2014. Available from: http://algs4.cs.princeton.edu/32bst/. 15, 16

[SW14b] Robert Sedgewick and Kevin Wayne. Tries, 2014. Available from: http://algs4.cs.princeton.edu/lectures/52Tries.pdf. 17

[TMOM12] Ramona Trestian, Arghir-Nicolae Moldovan, Olga Ormond, and Gabriel-Miro Muntean. Energy consumption analysis of video streaming to android mobile devices. In *2012 IEEE Network Operations and Management Symposium*, pages 444–452. IEEE, 2012. 1

[TMW94] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, Dec 1994. 1

[VBB+14] Mario Linares Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in android apps: an empirical study. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 2–11, 2014. 1

[VVHC+10] W. Vereecken, W. Van Heddeghem, D. Colle, M. Pickavet, and P. Demeester. Overall ict footprint and green communication technologies. In *4th International Symposium on Communications, Control and Signal Processing (ISCCSP)*, pages 1–6. IEEE, 2010. 1

[Wik16] Wikipedia. Association list — wikipedia, the free encyclopedia, 2016. [Online; accessed 10-June-2016]. Available from: https://en.wikipedia.org/w/index.php?title=Association_list&oldid=700513872. 17

[WJK+12] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. Measuring energy and power with papi. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 262–268. IEEE, 2012. 22

[YN03] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 149–163. ACM, 2003. 1

[YWJ10] Jinglei Yu, Eric Williams, and Meiting Ju. Analysis of material and energy consumption of mobile phones in china. *Energy Policy*, 38(8):4135 – 4141, 2010. Available from: `http://www.sciencedirect.com/science/article/pii/S0301421510002041`. 1