Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali

Corso di Laurea Triennale in Informatica

Sul costo della trasformazione CPS[1]

Tesi di Laurea in Paradigmi di Programmazione

Relatore:
Chiar.mo Prof.
Simone Martini

Presentata da:
Andrea Simonetto

Sessione II
Anno Accademico 2007/2008

[1] On the cost of the CPS translation

La gioia di scrivere.
Il potere di perpetrare.
La vendetta d'una mano mortale.
(Wislawa Szymborska, *La gioia di scrivere*)

Dedicato ai miei genitori.

Chapter 1

Introduction

The earliest programming languages were developed with one simple goal in mind: to provide a vehicle through which one could control the behavior of computers. Not surprisingly, the early languages reflected the structure of the underlying machines fairly well. Although at first blush that goal seems eminently reasonable, the viewpoint quickly changed for two very good reasons. First, it became obvious that what was easy for a machine to reason about was not necessarily easy for a human being to reason about. Second, as the number of different kinds of machines increased, the need arose for a common language with which to program all of them.

Thus from primitive assembly languages (which were at least a step up from raw machine code) there grew a plethora of high-level programming languages, beginning with FORTRAN in the 1950s. The development of these languages grew so rapidly that by the 1980s they were best characterized by grouping them into families that reflected a common computation model or programming style. Debates over which language or family of languages is best will undoubtedly persist for as long as computers need programmers.

The class of functional, or applicative, programming languages, in which computation is carried out entirely through the evaluation of expressions, is one such family of languages, and debates over its merits have been quite lively in recent years [8]. Are functional languages toys? Or are they tools?

Are they artifacts of theoretical fantasy or of visionary pragmatism? Will they ameliorate software woes or merely compound them? Whatever answers we might have for these questions, we cannot ignore the significant interest current researchers have in functional languages and the impact they have had on both the theory and pragmatics of programming languages in general. Among the claims made by functional language advocates are that programs can be written quicker, are more concise, are higher level (resembling more closely traditional mathematical notation), are more amenable to formal reasoning and analysis, and can be executed more easily on parallel architectures.

In this thesis we briefly present the formalism underlying the functional programming languages, the *lambda calculus*, originally developed by the logician Alonzo Church in 1941. Indeed, the lambda calculus is usually regarded as the first functional language, although it was certainly not thought of as programming language at the time, given that there were no computers on which to run the programs. In any case, modern functional languages can be thought of as (nontrivial) embellishments of the lambda calculus.

We continue presenting a programming technique noted as *continuation passing style*, which shows some interesting properties of programs. Main contribution of this thesis is a study of the complexity of the *continuation passing transformation*. Plotkin [14] original definition of this transformation makes particularly hard the proof of its main properties, and especially of the computation overhead it introduces during the reduction of a term. Building on results and techinques by Danvy and Filinski [3] – who treated esplicitely only the call-by-value case – we will study the computational overhead of a modified version of the call-by-name transformation. We will prove that the number of reduction steps necessary for the evaluation of a term is comparable to the number of steps necessary to evaluate the translated term (more precisly, growing in a linear way).

Chapter 2

λ-calculus

The λ-calculus is a theory of functions that was originally developed by the logician Alonzo Church [2] as foundation for mathematics. Despite its extreme simplicity, λ-calculus is powerful enough to express every recursive function [9, 1, 13]. The λ-calculus is a notation for defining functions, but it can also be used to represent a wide variety of data and data-structures including numbers, pairs, lists etc. The fundamental computation mechanism is *symbol rewriting*, performed with the β-reduction rule, while the execution flow is controlled by recursion, as application of fixed-points in the pure calculus and as recursive definitions in functional programming languages.

2.1 Syntax

Definition 1. *Assume given an infinite set of* variables*, denoted by x, y, z, etc. The set of lambda terms is given by the following Backus-Naur Form:*

$$M, N ::= x \mid (\lambda x.M) \mid (@MN)$$

A term of the form $(\lambda x.M)$ is an abstraction*; one of the form $(@MN)$ is an* application*. A term is a* value *iff it is not an application. $M \equiv N$ means that M and N are syntactically identical terms (i.e. the same term).*

2.2 Substitution

Free and bound variables

An occurrence of a variable x inside a term of the form $\lambda x.M$ is said to be *bound*. The corresponding λx is called a *binder*, and the subterm M is the *scope* of the binder. A variable occurrence that is not bound is *free*. Thus, for example, in the term:

$$M \equiv @(\lambda x.@xy)(\lambda y.@yz)$$

x is bound, but z is free. The variable y has both a free and a bound occurrence. The set of free variables of M is $\{y, z\}$.

Definition 2. *The set of free variables of a term M is denoted $FV(M)$, and it is defined as follows:*

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \\ FV(@MN) &= FV(M) \cup FV(N) \end{aligned}$$

A term M is closed *iff $FV(M) = \emptyset$, otherwise it is* open.

Definition 3. *The* size $|M|$ *of a term is defined by:*

$$|x| = 1 \qquad |\lambda x.M| = 1 + |M| \qquad |@MN| = 1 + |M| + |N|$$

Terms substitution

Definition 4. *Let M, N and P be terms and x, y variables. $M[x \leftarrow P]$ is called* substitution *and it is inductively defined as follows:*

$$y[x \leftarrow P] \equiv \begin{cases} P & \text{if } x = y \\ y & \text{otherwise} \end{cases}$$

$$(\lambda y.M)[x \leftarrow P] \equiv \begin{cases} \lambda y.M & \text{if } x = y \\ \lambda y.(M[x \leftarrow P]) & \text{if } x \neq y \wedge y \notin FV(P) \\ \lambda z.(M[y \leftarrow z][x \leftarrow P]) & \text{if } x \neq y \wedge y \in FV(P) \end{cases}$$

$$(@MN)[x \leftarrow P] \equiv @(M[x \leftarrow P])(N[x \leftarrow P])$$

(where z is a fresh variable).

Definition 5. α-equivalence *is the smallest congruence relation* $=_\alpha$ *on lambda terms, such that for all terms* M *and all variables* y *that do not occur in* M*:*

$$\lambda x.M =_\alpha \lambda y.(M[x \leftarrow y])$$

α-equivalent terms have identical interpretation and play identical roles in any application of λ-calculus. The notion of α-equivalence is what we usually mean when we refer to *equal terms*; in fact from now on we will write $=$ instead of $=_\alpha$.

2.3 β-reduction

β-reduction expresses the idea of *function application as term rewriting*. A term of form $@(\lambda x.M)N$ represents an operator $\lambda x.M$ applied to an argument N. In the informal interpretation of $\lambda x.M$, its value at N is calculated by substituting N for x in M, so $@(\lambda x.M)N$ can be "simplified" to $M[x \leftarrow N]$. This simplification-process is captured by the β-reduction rule. Furthermore, a term on which the β-reduction rule cannot be applied is called a β-normal-form.

In its classical meaning, the β-reduction rule is intrinsically nondeterministic: therefore we will talk about *evaluation strategies*, that is *how* the β-reduction rule must be applied to its arguments. For example, we can say that any application can be reduced at any time (full beta reductions); this means essentially the lack of any particular reduction strategy. Otherwise, we can require that the leftmost, outermost application is always reduced first (normal evaluation order). Anyway, the nondeterminism does not underly a limit for the calculus: in fact, there is a theorem (noted as *Church-Rosser theorem* or *confluence theorem* [9]) which guarantees that if a term M admits a β-normal-form, and M evaluates in a certain number of steps to two different terms P and Q, these terms will merge into a unique β-normal-form N.

However, in order to consider deterministic definitions of this rule, we present the two main evaluation strategies: *call-by-value* and *call-by-name*. Informally, evaluating a function call with call-by-value strategy means to recursively evaluate its arguments in order to obtain values, and then pass these values to the function. On the other side, the call-by-name strategy doesn't evaluate arguments before passing them to the function: it simply bind the formal parameters with the corresponding arguments. In both cases we do never evaluate under a λ-abstraction, because we consider them already as values. Formally:

Definition 6. *One step by-value reduction is defined as follows:*

$$@(\lambda x.M)V \to_v M[x \leftarrow V] \qquad \frac{M \to_v M'}{@MN \to_v @M'N} \qquad \frac{N \to_v N'}{@VN \to_v @VN'}$$

(where V is a value), and similarly one-step by-name reduction:

$$@(\lambda x.M)N \to_n M[x \leftarrow N] \qquad \frac{M \to_n M'}{@MN \to_n @M'N}$$

$M \to_a N$ *if both* $M \to_v N$ *and* $M \to_n N$*. For any of the three reduction relations* $\to$, $\to^+$ *and* $\to^*$ *are its transitive and reflexive-transitive closure, respectively. A λ-term on which β-reduction rule can be applied is called* redex *(abbreviation of* reducible expression*). A lambda term without any redex is said to be in β-*normal-form*. The "β" can be omitted when this causes no confusion.*

We also suppose to have two functions $Eval_v(M)$ and $Eval_n(M)$ that recursively apply the corresponding reduction step to the term M until it reaches a normal form; however if a normal form cannot be reached, the computation never ends and the function never returns. In 1964 Peter Landin proposed an abstract machine, called *SECD machine*, that implements the function $Eval_v$ [10].

2.4 η-reduction

The rule of η-reduction expresses the property that two functions are equal if they give same results when applied to the same arguments. This property is called *extensionality*.

Definition 7. *η-reduction is defined as follow:*

$$M \rightarrow_\eta \lambda x.@Mx \qquad \textit{where } x \notin FV(M)$$

Note that η-reduction produces a redex that is not generally $\alpha\beta$-equivalent to the original one, unless M is (or reduces to) an abstraction. In fact:

$$(\lambda y.M) \rightarrow_\eta \lambda x.@(\lambda y.M)x \rightarrow_\beta \lambda x.M[y \leftarrow x] \qquad \text{where } x \notin FV(M)$$

which is α-equivalent to $\lambda y.M$, while:

$$y \rightarrow_\eta \lambda x.@yx \qquad \text{where } x \neq y$$

aren't α- nor β-equivalent.

Chapter 3

Continuations

A continuation is a data structure which represents the future course of a computation. The use of continuations makes the global context of a computation available in the local context. In the early 1960's, the appearance of Algol 60 inspired a ferment of research on the implementation and formal definition of programming languages. Several aspects of this research were critical precursors of the discovery of continuations [16]. The ability in Algol 60 to jump out of blocks, or even procedure bodies, forced implementors to realize that the representation of a label must include a reference to an environment. According to Peter Naur:

> [...] in order to specify a transfer of control we must in general supply both the static description of the destination [...] and a dynamic description of its environment, the stack reference. This set [...] together define what we call a program point. [11]

In retrospect, a program point was the representation of a continuation.

A more subtle realization was that the return addresses could be treated on the same footing as procedure parameters. With a prescient choice of words, E. W. Dijkstra remarked:

> We use the name "parameters" for all the information that is presented to the subroutine when it is called in by the main program;

> function arguments, if any, are therefore parameters. The data grouped under the term "link" are also considered as parameters; the link comprises all the data necessary for the continuation of the main program when the subroutine has been completed. [5]

Another precursor of continuations occurred in Peter Landin's SECD machine. As captured by the acronym SECD, the state of the interpreter consisted of four components: a *stack*, an *environment*, a *control* and a *dump*. The dump encoded the remaining computation to be executed after the control was exhausted; in retrospect it was another representation of a continuation.

First class continuations

First-class continuations are constructs that give a programming language the ability to save the execution state at any point and return to that point at a later point in the program. They can be used to jump to a function that produced the call to the current function, or to a function that has previously exited. One can think of a first-class continuation as saving the state of the program. However, it is important to note that true first-class continuations do not save program data, only the execution context. This is illustrated by the "continuation sandwich" description:

> Say you're in the kitchen in front of the refrigerator, thinking about a sandwich. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there's a sandwich on the counter, and all the materials used to make it are gone. So you eat it. [1]

[1] cfr. *http://en.wikipedia.org/wiki/Continuation*

Only a few programming languages provide full, unrestrained access to the continuation of a computation step. Scheme was the first full production system, providing *call/cc* [6]. *call/cc* must be passed a procedure p of one argument. *call/cc* constructs a concrete representation of the the current continuation and passes it to p. The continuation itself is represented by a procedure k. Each time k is applied to a value, it returns the value to the continuation of the *call/cc* application. This value becomes, in essence, the value of the application of *call/cc*. Consider the simple examples below:

```
(call/cc
  (lambda (k)
    (* 5 4)))          ⇒ 20

(call/cc
  (lambda (k)
    (* 5 (k 4))))      ⇒ 4

(+ 2
  (call/cc
    (lambda (k)
      (* 5 (k 4)))))   ⇒ 6
```

In the first example, the continuation is obtained and bound to k, but k is never used, so the value is simply the product of 5 and 4. In the second, the continuation is invoked before the multiplication, so the value is the value passed to the continuation, 4. In the third, the continuation includes the addition by 2; thus, the value is the value passed to the continuation, 4, plus 2.

Bruce Duba introduced *call/cc* into SML. Some Smalltalk and Python implementations provide similar access to continuations, though nothing as systematic as Scheme. Many programming languages exhibit first-class continuations under various names, specifically:

- C: *setcontext* et al. (UNIX System V and GNU libc);
- Factor: *callcc0* and *callcc1*;
- Parrot: Continuation *PMC*; uses continuation passing style for all control flow;
- Perl: *Coro* and *Continuity*;
- Rhino: *Continuation*;
- Ruby: *callcc*;
- Scala: *Responder*;
- Scheme: *call-with-current-continuation* (commonly shortened to *call/cc*);
- Smalltalk: *Continuation currentDo*; in most modern Smalltalk environments continuations can be implemented without additional VM support;
- Standard ML of New Jersey: *SMLofNJ.Cont.callcc*;
- Unlambda: *c*, the flow control operation for call with current continuation;
- Pico: *call(exp())* and *continue(aContinuation, anyValue)*.

3.1 Usage in programming and compilers

In order to simplify the compilation process, many compilers for higher-order languages use the *continuation passing style transformation* in a first phase to generate an intermediate representation of the source program. The salient aspect of this intermediate form is that all procedures take an argument that represents the rest of the computation. Since the naïve CPS transformation considerably increases the size of programs, CPS compilers perform reductions to produce a more compact intermediate representation.

Although often implemented as a part of the CPS transformation, this step is conceptually a second phase. Finally, code generators for typical CPS compilers treat continuations specially in order to optimize the interpretation of continuation parameters.

Continuation passing style is also used in many application contexts, such as conversion of propositional formulas to conjunctive normal form [17], symbolic convolution [4], coroutines [7], backtracking, and expressing denotational semantics of programming languages. In fact, in standard semantics the concept of a continuation is introduced to represent the function denoting the rest of the program from any given point in its execution. Continuations permit elegant solutions to problems like the denotation of jumps, abnormal exits, and so on. For example, corresponding to the *reminder of the computation*, in Prolog there is the concept of *satisfying the remaining goals* [12].

Another area that has seen practical use of continuations is in Web programming [15]. The use of continuations shields the programmer from the stateless nature of the HTTP protocol. In the traditional model of web programming, the lack of state is reflected in the program's structure, leading to code constructed around a model that lends itself very poorly to expressing computational problems. Thus, continuations enable code that has the useful properties associated with inversion of control, while avoiding its problems. Some of the most popular continuation-aware Web servers are the *PLT Scheme Web Server*, the *UnCommon Web Framework* and *Weblocks Web framework* for Common Lisp, and the *Seaside Web Server* for Smalltalk. The *Apache Cocoon Web application framework* also provides continuations.

3.2 The CPS transformation

Instead of "returning" values as in the more familiar direct style, a function written in continuation-passing style (CPS) takes an explicit continuation argument which is meant to receive the result of the computation

performed within the function. When a subroutine is invoked within a CPS function, the calling function is required to supply a procedure to be invoked with the subroutine's "return" value. Expressing code in this form makes a number of things explicit which are implicit in direct style. These include:

- procedure returns, which become apparent as calls to a continuation; for example:

$$\lambda x.x \quad \leadsto \quad \lambda x.\lambda\kappa.@\kappa x$$

- intermediate values, which are all given names, and order of argument evaluation, which is made explicit:

$$\lambda x.@foo(@bar\ x) \quad \leadsto \quad \lambda x.\lambda\kappa.@(@bar'\ x)(\lambda m.@(@foo'\ m)\kappa)$$

- tail recursion, which is simply calling a procedure with the continuation that was passed to the caller: in fact, we note that in the example above, both foo' and bar' are tail-recursive.

In CPS, each procedure takes an extra argument representing what should be done with the result the function is calculating. This, along with a restrictive style prohibiting a variety of constructs usually available, is used to expose the semantics of programs, making them easier to analyze. This style also makes it easy to express unusual control structures, like *catch/throw* or other non-local transfers of control.

The key to CPS is to remember that (a) every function takes an extra argument, its continuation, and (b) every argument in a function call must be either a variable or a lambda expression (not a more complex expression). This has the effect of turning expressions "inside-out" because the innermost parts of the expressions must be evaluated first. Some examples of code in direct style and the corresponding CPS style appear below. These examples are written in the Scheme programming language:

Direct Style	**Continuation Passing Style**

(define (pyth x y)
  (sqrt (+ (* x x) (* y y))))
	(define (pyth x y k)
	  (* x x (lambda (x2)
	  (* y y (lambda (y2)
	  (+ x2 y2 (lambda (x2py2)
	  (sqrt x2py2 k))))))))

(define (factorial n)
  (if (= n 0)
  1
  (* n (factorial (- n 1)))))
	(define (factorial n k)
	  (= n 0 (lambda (b)
	  (if b
	  (k 1)
	  (- n 1 (lambda (nm1)
	  (factorial nm1 (lambda (f)
	  (* n f k)))))))))

(define (factorial n) (f-aux n 1))
(define (f-aux n a)
  (if (= n 0)
  a
  (f-aux (- n 1) (* n a))))
	(define (factorial n k)
	  (f-aux n 1 k))
	(define (f-aux n a k)
	  (= n 0 (lambda (b)
	  (if b
	  (k a)
	  (- n 1 (lambda (nm1)
	  (* n a (lambda (nta)
	  (f-aux nm1 nta k))))))))))

In order to call a procedure written in CPS from a procedure written in direct style, it is necessary to provide a continuation. In the example above,

we might call (factorial 10 identity). This will not work directly with the code above, because in the CPS version we are assuming that primitives like + and * are in CPS, so to make the above example work in a Scheme system we would need to write new CPS versions of these primitives and use them instead: cps* instead of *, etc, where (define (cps* x y k) (k (* x y))), etc.

In any language which supports closures, it is possible to write programs in continuation passing style and manually implement *call/cc*. This is a particularly common strategy in Haskell, where it is easy to construct a "continuation passing monad" (for example, the *Cont* monad and *ContT* monad transformer in the *mtl* library).

Programs can be automatically transformed from direct style to CPS; so we talk of *continuation passing transformation* or *CPS transformation*. The transformation of a program is the application of the transformed to the *empty continuation* (or *empty context*), that is the one that takes the program return value and returns it back, that is $\lambda x.x$, the identity function. Note that in CPS, there is no implicit continuation, every call is a tail call. There is no "magic" here, as the continuation is simply explicitly passed.

This use of the CPS transformation finds its theoretical grounds and formal definitions in the work of Gordon Plotkin "Call-by-name, call-by-value and the λ-calculus" [14].

3.3 Reduction properties

In 1974 Gordon Plotkin performed a formal study on the differences between the evaluation strategies *call-by-value* and *call-by-name*. He proved some important properties of the CPS transformation, among them: the insensitivity of the transformed term from the evaluation strategy and the possibility of simulating an evaluation mechanism with the other one (and vice-versa) [14].

Definition 8. *Plotkin's CPS transform of λ_v and λ_n-terms, respectively, are*

defined as follows:

$$\begin{array}{rcl} [\![...]\!]_v & : & syntax \to syntax \\ [\![x]\!]_v & \stackrel{def}{=} & \lambda k.@kx \\ [\![\lambda x.M]\!]_v & \stackrel{def}{=} & \lambda k.@k(\lambda x.[\![M]\!]_v) \\ [\![@MN]\!]_v & \stackrel{def}{=} & \lambda k.@[\![M]\!]_v(\lambda m.@[\![N]\!]_v(\lambda n.@(@mn)k)) \end{array}$$

$$\begin{array}{rcl} [\![...]\!]_n & : & syntax \to syntax \\ [\![x]\!]_n & \stackrel{def}{=} & x \\ [\![\lambda x.M]\!]_n & \stackrel{def}{=} & \lambda k.@k(\lambda x.[\![M]\!]_n) \\ [\![@MN]\!]_n & \stackrel{def}{=} & \lambda k.@[\![M]\!]_n(\lambda m.@(@m[\![N]\!]_n)k) \end{array}$$

Definition 9. *The Ψ function sends values to values and it is defined by:*

$$\Psi(x) \stackrel{def}{=} x \qquad \Psi(\lambda x.M) \stackrel{def}{=} \lambda x.[\![M]\!]_v$$

Theorem 1 (Indifference)**.** *The transformed term is independent from the evaluation strategy.*

$$Eval_v(@[\![M]\!]_v(\lambda m.m)) \backsimeq Eval_n(@[\![M]\!]_v(\lambda m.m))$$

This result is very powerful: for example, it allows to define a meta-circular interpreter (i.e. an interpreter for the language $\mathcal{L}$ written in $\mathcal{L}$) without worrying about *how* it will be evaluated. In fact, we just have to transform it with the $[\![\cdot]\!]_v$ function, in order to force (in every interpretation) the *call-by-value* evaluation strategy (or with $[\![\cdot]\!]_n$ if we want to force the *call-by-name* strategy).

Theorem 2 (Simulation)**.** *Evaluating by-value the original term is the same (via the Ψ function) of evaluating the transformed term in the empty continuation.*

$$\Psi(Eval_v(M)) \backsimeq Eval_v(@[\![M]\!]_v(\lambda m.m))$$

Analogous theorems can be stated for the $[\![\cdot]\!]_n$ transformation. We will state them in the next chapter, where they are a major subject of study.

Chapter 4

Complexity of the CPS transformation

Taken literally, the original translation yields many artificial "administrative" redexes that must be post-reduced in a second pass; only then do we obtain a result in what is commonly recognized as "continuation passing style".

Let's suppose we have an intepreter for the λ_n language (pure lambda calculus with call-by-name evaluation strategy) capable of handling first-class continuations. We note that an overly enthusiastic post-reducer is likely to perform too many reductions: while this may be useful in its own right, it should not automatically be considered a part of the CPS transformation proper. In particular, excessive post-reduction can lead to uncontrolled "code duplication" in the result or, in the untyped case, even nontermination of the simplification. For example:

$@[\![@(\lambda x.@xx)(\lambda x.@xx)]\!](\lambda v.v)$

$$
\begin{aligned}
&\to_a \quad @[\![\lambda x.@xx]\!](\lambda m.@(@m[\![\lambda x.@xx]\!])(\lambda v.v)) && (4.1)\\
&\to_a \quad @(\lambda m.@(@m[\![\lambda x.@xx]\!])(\lambda v.v))(\lambda x.[\![@xx]\!]) && (4.2)\\
&\to_a \quad @(@(\lambda x.[\![@xx]\!])[\![\lambda x.@xx]\!])(\lambda v.v) && (4.3)\\
&\to_a \quad @([\![@xx]\!][x{\leftarrow}[\![\lambda x.@xx]\!]])(\lambda v.v) && (4.4)\\
&\to_a \quad (@[\![x]\!](\lambda m.@(@m[\![x]\!])(\lambda v.v)))[x{\leftarrow}[\![\lambda x.@xx]\!]] && (4.5)\\
&= \quad (@x(\lambda m.@(@mx)(\lambda v.v)))[x{\leftarrow}[\![\lambda x.@xx]\!]] && (4.6)\\
&= \quad @[\![\lambda x.@xx]\!](\lambda m.@(@m[\![\lambda x.@xx]\!])(\lambda v.v)) && (4.1)\\
&\to_a \quad \ldots
\end{aligned}
$$

Our intepreter will work on two logical levels:

- the level of "dynamic" terms, i.e. the terms that are parts of the program defined by the user; therefore the evaluation of these terms produces unpredictable results;
- the level of "static" terms, i.e. the administrative terms that are introduced by the CPS translation.

We can graphically distinguish the two levels: we will underline the dynamic λ-abstractions and applications and we will overline the static ones.

The intepreter will have to implicitly evaluate the static terms, because these terms are logically part of the translation. When we found a term in this form:

$$\overline{@}(\overline{\lambda}x.M)V$$

we will ignore the fact that a step of β-reduction is necessary for the evaluation of that term; therefore we will write:

$$\overline{@}(\overline{\lambda}x.M)V = M[x{\leftarrow}V]$$

that means that this reduction step will be implicitly performed by the interpreter. On the other hand, in case of a term in this form:

$$\underline{@}(\underline{\lambda}x.M)V$$

the application of the β-reduction rule will have to be explicit. In this way, we will write:

$$\underline{@}(\underline{\lambda}x.M)V \rightarrow M[x \leftarrow V]$$

In the intermediate cases:

$$\overline{@}(\underline{\lambda}x.M)V \qquad \text{and} \qquad \underline{@}(\overline{\lambda}x.M)V$$

the application of the β-reduction will be still explicit. To avoid these situations and factorising all the administrative reductions in a unique phase following the syntactic transformation, we will introduce in the last section a tecnique noted as *one-pass CPS translation*.

For precision and conciseness in the text, let us label the six lambdas and applications of this specification:

$$\begin{aligned} [\![x]\!] &\stackrel{def}{=} x \\ [\![\lambda x.M]\!] &\stackrel{def}{=} \lambda^1 k.@^1 k(\lambda^2 x.[\![M]\!]) \\ [\![@MN]\!] &\stackrel{def}{=} \lambda^3 k.@^2 [\![M]\!](\lambda^4 m.@^3(@^4 m[\![N]\!])k) \end{aligned}$$

We first notice that the only "dynamic" abstraction is λ^2 while the only "dynamic" application is $@^4$, so we can underline them. The result of the translation is a variable in the first case and an abstraction in the other two cases: we can introduce an η-redex in the first case in order to exhibit the application of the continuation:

$$[\![x]\!] \stackrel{def}{=} \lambda^5 k.@^5 xk$$

This guarantees that the application of an η-reduction step to $[\![M]\!]$ builds a redex which is $\alpha\beta$-equivalent to $[\![M]\!]$.

> **Remark:** Such an η-expansion may be felt as a step backwards in optimizing the translation, since η-reduction is usually perceived as an actual optimization step. In fact, the premature optimization in the translation of λ-abstractions contributes to muddying the water in the translated term.

The other abstractions and applications are static, so we can overline them. To conclude, we shows the final transformer, without labels since they were only used for expository purposes.

$$\begin{array}{rcl} [\![...]\!] & : & \text{syntax} \to \text{syntax} \\ [\![x]\!] & \stackrel{def}{=} & \overline{\lambda}k.\overline{@}xk \\ [\![\lambda x.M]\!] & \stackrel{def}{=} & \overline{\lambda}k.\overline{@}k(\underline{\lambda}x.[\![M]\!]) \\ [\![@MN]\!] & \stackrel{def}{=} & \overline{\lambda}k.\overline{@}[\![M]\!](\overline{\lambda}m.\overline{@}(\underline{@}m[\![N]\!])k) \end{array}$$

Our target is to give a valuation of the of the number of β-reduction steps necessary to evaluate $\overline{@}[\![M]\!](\overline{\lambda}m.m)$, in function of those required to simply evaluate M. Using the previous described interpretation, we proceed proving a theorem which captures the 'indifference" and "simulation" results, with evaluation of the reduction steps.

4.1 Properties of new specification

Since we altered the original specification in a meaning-preserving way (by introducing an η-redex) our specification computes a result that is $\beta\eta$-equivalent to Plotkin's one.

Lemma 1. *The transformation is insensitive to η-conversion. In fact, the result of transforming a term is always an abstraction. Formally, let M be any term:*

$$[\![M]\!] = \overline{\lambda}k.\overline{@}[\![M]\!]k$$

Proof. Immediate in all cases. □

Lemma 2 (Substitution)**.** *Let M, P, K be terms and x a variable:*

$$\overline{@}[\![M[x{\leftarrow}P]]\!]K = (\overline{@}[\![M[x{\leftarrow}x']]\!]K)[x'{\leftarrow}[\![P]\!]]$$

where x' is a fresh variable.

Proof. By structural induction on M. Let $y \neq x$:

$$\begin{aligned}
\overline{@}[\![x[x{\leftarrow}P]]\!]K &= \overline{@}[\![P]\!]K = (\overline{@}x'K)[x'{\leftarrow}[\![P]\!]] = (\overline{@}x[x{\leftarrow}x']K)[x'{\leftarrow}[\![P]\!]] \\
&= (\overline{@}[\![x[x{\leftarrow}x']]\!]K)[x'{\leftarrow}[\![P]\!]]
\end{aligned}$$

$$\begin{aligned}
\overline{@}[\![y[x{\leftarrow}P]]\!]K &= \overline{@}[\![y]\!]K = (\overline{@}[\![y[x{\leftarrow}x']]\!]K)[x'{\leftarrow}[\![P]\!]] \\
\overline{@}[\![(\lambda x.M)[x{\leftarrow}P]]\!]K &= \text{same as } y[x{\leftarrow}P]
\end{aligned}$$

$$\begin{aligned}
\overline{@}[\![(\lambda y.M)[x{\leftarrow}P]]\!]K &= \overline{@}[\![\lambda y.M[x{\leftarrow}P]]\!]K = \overline{@}K(\underline{\lambda}y.[\![M[x{\leftarrow}P]]\!]) \\
\text{(by Lemma 1)} &= \overline{@}K(\underline{\lambda}y.\overline{\lambda}k.\overline{@}[\![M[x{\leftarrow}P]]\!]k) \\
\text{(by I.H.)} &= \overline{@}K(\underline{\lambda}y.\overline{\lambda}k.(\overline{@}[\![M[x{\leftarrow}x']]\!]k)[x'{\leftarrow}[\![P]\!]]) \\
&= (\overline{@}K(\underline{\lambda}y.\overline{\lambda}k.\overline{@}[\![M[x{\leftarrow}x']]\!]k))[x'{\leftarrow}[\![P]\!]] \\
\text{(by Lemma 1)} &= (\overline{@}K(\underline{\lambda}y.[\![M[x{\leftarrow}x']]\!]))[x'{\leftarrow}[\![P]\!]] \\
&= (\overline{@}[\![\lambda y.M[x{\leftarrow}x']]\!]K)[x'{\leftarrow}[\![P]\!]] \\
&= (\overline{@}[\![(\lambda y.M)[x{\leftarrow}x']]\!]K)[x'{\leftarrow}[\![P]\!]]
\end{aligned}$$

$$\begin{aligned}
\overline{@}[\![(@MN)[x{\leftarrow}P]]\!]K &= \overline{@}[\![@(M[x{\leftarrow}P])(N[x{\leftarrow}P])]\!]K \\
&= \overline{@}[\![M[x{\leftarrow}P]]\!](\overline{\lambda}m.\overline{@}(\underline{@}m[\![N[x{\leftarrow}P]]\!])K) \\
\text{(by I.H. on M)} &= (\overline{@}[\![M[x{\leftarrow}x']]\!](\overline{\lambda}m.\overline{@}(\underline{@}m[\![N[x{\leftarrow}P]]\!])K))[x'{\leftarrow}[\![P]\!]] \\
\text{(by Lemma 1)} &= (\overline{@}[\![M[x{\leftarrow}x']]\!](\overline{\lambda}m.\overline{@}(\underline{@}m(\overline{\lambda}k.\overline{@}[\![N[x{\leftarrow}P]]\!]k))K))[x'{\leftarrow}[\![P]\!]] \\
\text{(by I.H. on N)} &= (\overline{@}[\![M[x{\leftarrow}x']]\!](\overline{\lambda}m.\overline{@}(\underline{@}m(\overline{\lambda}k.\overline{@}[\![N[x{\leftarrow}x']]\!]k))K))[x'{\leftarrow}[\![P]\!]] \\
\text{(by Lemma 1)} &= (\overline{@}[\![M[x{\leftarrow}x']]\!](\overline{\lambda}m.\overline{@}(\underline{@}m[\![N[x{\leftarrow}x']]\!])K))[x'{\leftarrow}[\![P]\!]] \\
&= (\overline{@}[\![@(M[x{\leftarrow}x'])(N[x{\leftarrow}x'])]\!]K)[x'{\leftarrow}[\![P]\!]] \\
&= (\overline{@}[\![(@MN)[x{\leftarrow}x']]\!]K)[x'{\leftarrow}[\![P]\!]]
\end{aligned}$$

□

Lemma 3. *Let M and N be terms such that $M \to_n N$, and K any term. Then:*

$$\overline{@}[\![M]\!]K \to_a \overline{@}[\![N]\!]K$$

Proof. By induction on the derivation of $\rightarrow_n$:

- Base case: $@(\lambda x.M)N \rightarrow_n M[x \leftarrow N]$.

$$\begin{array}{rcl}
\overline{@}[\![@(\lambda x.M)N]\!]K & = & \overline{@}[\![\lambda x.M]\!](\overline{\lambda}m.\overline{@}(\underline{@}m[\![N]\!])K) \\
 & = & \overline{@}(\underline{@}(\underline{\lambda}x.[\![M]\!])[\![N]\!])K \\
\text{(by Lemma 1)} & = & \overline{@}(\underline{@}(\underline{\lambda}x.\overline{\lambda}k.\overline{@}[\![M]\!]k)[\![N]\!])K \\
 & \rightarrow_a & \overline{@}((\overline{\lambda}k.\overline{@}[\![M]\!]k)[x \leftarrow [\![N]\!]])K \\
 & = & \overline{@}(\overline{\lambda}k.(\overline{@}[\![M[x \leftarrow x']]\!]k)[x' \leftarrow [\![N]\!]])K \\
\text{(by Lemma 2)} & = & \overline{@}(\overline{\lambda}k.\overline{@}[\![M[x \leftarrow N]]\!]k)K \\
 & = & \overline{@}[\![M[x \leftarrow N]]\!]K
\end{array}$$

- Inductive case: $@MN \rightarrow_n @M'N$ because $M \rightarrow_n M'$.

$$\begin{array}{rcl}
\overline{@}[\![@MN]\!]K & = & \overline{@}[\![M]\!](\overline{\lambda}m.\overline{@}(\underline{@}m[\![N]\!])K) \\
\text{(by I.H.)} & \rightarrow_a & \overline{@}[\![M']\!](\overline{\lambda}m.\overline{@}(\underline{@}m[\![N]\!])K) \\
 & = & \overline{@}[\![@M'N]\!]K
\end{array}$$

□

If we restrict evaluation to closed terms, any term is either already a value or contains a redex. However, the results extend easily to open terms, with free variables treated as uninterpreted constants. In this case, there is a third possibility: evaluation may halt at non-value term like $@xy$ from which no further progress is possible.

Following Plotkin, we define:

Definition 10. *A (necessarily open) term S is said to be* stuck *under a given strategy if it is neither a value nor reducible by any of the reduction rules for that strategy. A quick inspection of the Definition 6 shows that such terms must be of the following form (where V is a value and N is any term):*

$$\begin{array}{lrcl}
\textit{(for call-by-value)} & S_v & ::= & @xV \mid @S_vN \mid @VS_v \\
\textit{(for call-by-name)} & S_n & ::= & @xN \mid @S_nN
\end{array}$$

Lemma 4. *Let M, K be terms. If M is stuck under CBN then $\overline{@}[\![M]\!]K$ is stuck under any strategy.*

Proof. By structural induction on the stuck term M.

- Base case, $M = @xN$:

$$\begin{aligned}\overline{@}[\![@xN]\!]K &= \overline{@}[\![x]\!](\overline{\lambda}m.\overline{@}(\underline{@}m[\![N]\!])K) \\ &= \overline{@}x(\overline{\lambda}m.\overline{@}(\underline{@}m[\![N]\!])K)\end{aligned}$$

  which is stuck under any strategy.

- Inductive case, $M = @S_nN$:

$$\overline{@}[\![@S_nN]\!]K = \overline{@}[\![S_n]\!](\overline{\lambda}m.\overline{@}(\underline{@}m[\![N]\!])K)$$

  which is stuck, by the induction hypothesis.

□

We can now state the main result, analogous to Plotkin's "Indifference" and "Simulation" theorems:

Theorem 3. *Let M be any term (not necessarily closed) and N a term. If $M \to_n^* N$, then $\overline{@}[\![M]\!](\overline{\lambda}m.m) \to_a^* \overline{@}[\![N]\!](\overline{\lambda}m.m)$ in the same number of steps. Conversely, if M does not evaluate to a value under the call-by-name strategy, then for no strategy will $\overline{@}[\![M]\!](\overline{\lambda}m.m)$ evaluate to one.*

Proof. The first part follows immediately from Lemma 3 applied to every step of the reduction. Conversely, any infinite reduction sequence starting from M gives rise to an infinite, strategy-independent reduction sequence starting from $\overline{@}[\![M]\!](\overline{\lambda}m.m)$. Finally, if the original reduction sequence stops at a CBN-stuck term S, the corrisponding CPS reduction sequence ends in the term $\overline{@}[\![S]\!](\overline{\lambda}m.m)$ which is stuck under any strategy (Lemma 4). □

4.2 A valuation of complexity

The result proved in Theorem 3 can be presented schematically as follows:

$$\begin{array}{ccc} M & \rightarrow_n^i & N \\ \downarrow & & \downarrow \\ \overline{@}[\![M]\!](\overline{\lambda}m.m) & \rightarrow_a^i & \overline{@}[\![N]\!](\overline{\lambda}m.m) \end{array}$$

where i is the number of the reduction steps. It is apparently surprising that the number of the reduction steps necessary to evaluate a term and its transformed are equal: in fact, the sizes of the two terms are very different. The problem is that we have considered equivalent different terms that our interpreter can simplify automatically during the translation. Then, the question remains: how much does the translation $[\![\cdot]\!]$ costs in terms of β-reduction steps? Let's confront the size of the original term with that of the transformed:

$$\begin{array}{rclcrcl} |x| & = & 1 & & |[\![x]\!]| & = & 4 \\ |\lambda x.M| & = & 1+|M| & & |[\![\lambda x.M]\!]| & = & 4+|[\![M]\!]| \\ |@MN| & = & 1+|M|+|N| & & |[\![@MN]\!]| & = & 7+|[\![M]\!]|+|[\![N]\!]| \end{array}$$

We note that the size of the transformed term grows up linearly in function of the size of the original term in a reduction sequence. Furthermore, we observe that the variables in the original term occur only once in the translated term.

> **Remark:** Therefore, we can state that if the original term M reduces to a term N in a finite number of steps n, $\overline{@}[\![M]\!](\overline{\lambda}m.m)$ will evaluate to $\overline{@}[\![N]\!](\overline{\lambda}m.m)$ in a number of steps equal to $O(n)$.

4.3 One-pass CPS translation

We note that the intermediate cases:

$$\overline{@}(\underline{\lambda}x.M)V \qquad \text{and} \qquad \underline{@}(\overline{\lambda}x.M)V$$

are built in a "context-dependent" fashion: to get rid of these situations, in the following, we will analyze the equational specification, identifying where redexes get built independently of any actual source λ-term. When these redex are "context-independent" we reduce them at translation time. When they are "context-dependent" we alter the translation with meaning-preserving transformations to make the construction of redexes context-independent. Such "transformer", performs all the administrative reductions in the beginning and yields terms in proper continuation passing style in one-pass. We use the labelled specification introduced in the beginning of this chapter:

$$\begin{array}{rcl} [\![...]\!] & : & \text{syntax} \rightarrow \text{syntax} \\ [\![x]\!] & \stackrel{def}{=} & \lambda^5 k.@^5 xk \\ [\![\lambda x.M]\!] & \stackrel{def}{=} & \lambda^1 k.@^1 k(\lambda^2 x.[\![M]\!]) \\ [\![@MN]\!] & \stackrel{def}{=} & \lambda^3 k.@^2 [\![M]\!](\lambda^4 m.@^3(@^4 m[\![N]\!])k) \end{array}$$

As can be observed, the result of each elementary transformation (of a variable; of an abstraction; of an application) is an abstraction. *Where can the abstraction λ^5, λ^1, and λ^3 occur in the residual CPS term before post-reduction?* By cases: (a) as the body of λ^2; (b) as the first argument of $@^2$; (c) as the second argument of $@^4$. In case (b) the translation is building a redex that can be simplified by β-reduction. In the other cases no simplification can take place immediately, so we can introduce two η-redexes, one in the definitional translation of the abstraction and one in the application:

$$\begin{array}{rcl} [\![\lambda x.M]\!] & \stackrel{def}{=} & \lambda^1 k.@^1 k(\lambda^2 x.\lambda^6 k.@^6 [\![M]\!]k) \\ [\![@MN]\!] & \stackrel{def}{=} & \lambda^3 k.@^2 [\![M]\!](\lambda^4 m.@^3(@^4 m(\lambda^7 k.@^7 [\![N]\!]k))k) \end{array}$$

The new redexes are safe (in the sense of preserving the operational behaviour under both CBN and CBV) because $[\![M]\!]$ is itself a λ-abstraction (note that applying η-expansion is not in general meaning-preserving, as shown in Section 2.4).

Since the three λ-abstractions λ^5, λ^1, and λ^3 will be reduced at translation time, let us enumerate their possible arguments: *which syntactic constructs can be denoted by k in λ^5, λ^1, and λ^3?* By cases: (a) the second argument of $@^6$ is an identifier k; (b) the second argument of $@^2$ is λ^4; (c) the second argument of $@^7$ is k. Again, the situation is irregular: if the argument of these applications (*i.e.*, the value denoted by k) later gets applied, this application will be reducible only in case (b), *i.e.*, in a context-dependent fashion. We can introduce another two η-redexes in order to exhibit the application of the continuation:

$$\begin{aligned}
[\![\lambda x.M]\!] &\stackrel{def}{=} \lambda^1 k.@^1 k(\lambda^2 x.\lambda^6 k.@^6 [\![M]\!](\lambda^8 m.@^8 mk)) \\
[\![@MN]\!] &\stackrel{def}{=} \lambda^3 k.@^2 [\![M]\!](\lambda^4 m.@^3(@^4 m(\lambda^7 k.@^7 [\![N]\!](\lambda^9 n.@^9 nk)))k)
\end{aligned}$$

Now the different occurences of k are ensured to denote λ-abstractions only.

Where do these k occur? By cases: (a) as the second argument of $@^5$; (b) as the first argument of $@^1$; (c) as the second argument of $@^3$. In case (b) the translation is building a redex that can be simplified by β-reduction. In cases (a) and (c) no simplification can take place immediately. As consequence, whether the application of a k is post-reducible is context-dependent since in cases (a) and (b), k does not occur in function position in an application. We can introduce the last two η-redexes in order to exhibit sending the result of an application to the continuation.

$$\begin{aligned}
[\![x]\!] &\stackrel{def}{=} \lambda^5 k.@^5 x(\lambda^{10} a.@^{10} ka) \\
[\![@MN]\!] &\stackrel{def}{=} \lambda^3 k.@^2 [\![M]\!](\lambda^4 m.@^3(@^4 m(\lambda^7 k.@^7 [\![N]\!](\lambda^9 n.@^9 nk)))(\lambda^{11} a.@^{11} ka))
\end{aligned}$$

As consequence, because by construction the translation is building a λ-abstraction which is ensured to occur in function position in an application, we can classify these λ-abstraction and applications to be simplifiable unconditionally.

We can also list the possible argument of the k: they are the second argument of $@^{10}$, $@^1$, and $@^{11}$, *i.e.*, (a) an identifier a; (b) λ^2; (c) an identifier a. These may be bound to m in λ^8, m in λ^4, and n in λ^9.

- In λ^8, m occurs as the first argument of $@^8$.
  - If m is bound to a, no simplification is possible;
  - If m is bound to λ^2 then a β-reduction is possible but it would correspond to a reduction in the original term; therefore we do not want to perform it. Thus $@^8$ must be classified as irreducible and so is λ^2 and λ^6.
- In λ^4, m occurs as the first argument of $@^4$, which is irreducible.
- in λ^9, n occurs as the first argument of $@^9$ that cannot be reduced since the second argument is the identifier k.

As a consequence, the second argument of $@^3$ is irreducible and thus $@^3$ is irreducible. As another consequence, the first argument of $@^3$ must be irreducible.

To conclude, we show the final version of the trasformer, without labels since they were only used for expository purpose:

$$\begin{array}{rcl} [\![...]\!] & : & [\text{syntax} \to \text{syntax}] \to \text{syntax} \\ [\![x]\!] & \stackrel{def}{=} & \overline{\lambda}k.\underline{@}x(\underline{\lambda}a.\overline{@}ka) \\ [\![\lambda x.M]\!] & \stackrel{def}{=} & \overline{\lambda}k.\overline{@}k(\underline{\lambda}x.\underline{\lambda}k.\overline{@}[\![M]\!](\overline{\lambda}m.\underline{@}km)) \\ [\![@MN]\!] & \stackrel{def}{=} & \overline{\lambda}k.\overline{@}[\![M]\!](\overline{\lambda}m.\underline{@}(\underline{@}m(\underline{\lambda}k.\overline{@}[\![N]\!](\overline{\lambda}n.\underline{@}kn)))(\underline{\lambda}a.\overline{@}ka)) \end{array}$$

It can be shown that this new specification produces more η-redexes than Plotkin's one, which are not nearly as hard to get rid of as the β-redexes produced by the original translation. η-redexes are only constructed in tail-contexts, for identifiers occurring as λ-abstraction bodies, and as arguments of functions. Danvy and Filinski proposed to duplicate the rules to account for tail-call contexts [3].

> **Remark:** Despite the increased size of the transformed term, the results on the complexity presented above are still valid. In fact, we altered the translation in a meaning-preserving way, without duplicating variables nor introducing more recursion.

Bibliography

[1] H. P. Barendregt. *The Lambda Calculus: its syntax and semantics*. North-Holland publishing company, 1981.

[2] Alonzo Church. *The calculi of lambda-conversion*. Princeton University press, 1941.

[3] Oliver Danvy and Andrzej Filinski. Representing control - A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[4] Olivier Danvy and Mayer Goldberg. There and back again. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 230–234, New York, NY, USA, 2002. ACM.

[5] Edsger W. Dijkstra. Recursive programming. *Numerische Mathematik*, 2:312–318, 1960.

[6] R. Kent Dybvig. *The Scheme Programming Language*. The MIT Press, 3rd edition, 2003. Entire text available online at http://www.scheme.com/tspl3/.

[7] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298, New York, NY, USA, 1984. ACM.

[8] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[9] Jonathan P. Seldin J. Roger Hindley. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[10] Peter J. Landin. The mechanical evaluation of expressions. *The computer journal*, 6(4):308–320, 1964.

[11] Peter Naur. The design of the GIER ALGOL compiler, part i. *BIT Numerical Mathematics*, 3:124–140, 1963. Reprinted in Goodman, Richard, editor, *Annual Review in Automatic Programming*, Vol. 4, Pergamon Press, Oxford (1964) 49–85.

[12] Tim Nicholson and Norman Foo. A denotational semantics for prolog. *ACM Trans. Program. Lang. Syst.*, 11(4):650–665, 1989.

[13] Piergiorgio Odifreddi. *Classical Recursion Theory*. Elsevier Science Publishers B.V., 1950.

[14] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[15] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, 2003.

[16] John C. Reynolds. The discoveries of continuations. *LISP AND SYMBOLIC COMPUTATION: An internal journal*, 6:233–247, 1993.

[17] Mitchell Wand. Continuation-based program transformation strategies. *J. ACM*, 27(1):164–180, 1980.

Ringraziamenti

Ringrazio innanzitutto i miei genitori, che mi hanno dato la possibilità e gli strumenti per iscrivermi all'Università, e l'incoraggiamento ed il sostegno – nonostante il mio caratteraccio – per completare questo ciclo di studi. Grazie di cuore.

Uno speciale ringraziamento va inoltre a Odeta Qorri, per avermi pazientemente aiutato con l'inglese e corretto la prima stesura.

Infine grazie a Giovanni Moschese, Francesco Lupi e Marco Nardone per avermi sopportato durante tutto il periodo di stesura della tesi.