POLITÉCNICA "Ingeniamos el futuro" CAMPUS DE EXCELENCIA INTERNACIONAL

UNIVERSIDAD POLITÉCNICA DE MADRID

E.T.S.I. INFORMÁTICOS

MASTER UNIVERSITARIO EN SOFTWARE Y SISTEMAS

TRABAJO DE FIN DE MASTER

Implementing a term rewriting engine for the EasyCrypt framework

Autor: Guillermo Ramos Gutiérrez
Director: Manuel Carro Liñares
Co-director: Pierre-Yves Strub

MADRID, 28 de julio de 2015

Resumen

La sociedad depende hoy más que nunca de la tecnología, pero la inversión en seguridad es escasa y los sistemas informáticos siguen estando muy lejos de ser seguros. La criptografía es una de las piedras angulares de la seguridad en este ámbito, por lo que recientemente se ha dedicado una cantidad considerable de recursos al desarrollo de herramientas que ayuden en la evaluación y mejora de los algoritmos criptográficos. EasyCrypt es uno de estos sistemas, desarrollado recientemente en el Instituto IMDEA Software en respuesta a la creciente necesidad de disponer de herramientas fiables de verificación formal de criptografía.

En este trabajo se abordará la implementación de una mejora en el reductor de términos de EasyCrypt, sustituyéndolo por una máquina abstracta simbólica. Para ello se estudiarán e implementarán previamente dos máquinas abstractas muy conocidas, la Máquina de Krivine y la ZAM, introduciendo variaciones sobre ellas y estudiando sus diferencias desde un punto de vista práctico.

Abstract

Today, society depends more than ever on technology, but the investment in security is still scarce and using computer systems are still far from safe to use. Cryptography is one of the cornerstones of security, so there has been a considerable amount of effort devoted recently to the development of tools oriented to the evaluation and improvement of cryptographic algorithms. One of these tools is EasyCrypt, developed recently at IMDEA Software Institute in response to the increasing need of reliable formal verification tools for cryptography.

This work will focus on the improvement of the EasyCrypt's term rewriting system, replacing it with a symbolic abstract machine. In order to do that, we will previously study and implement two widely known abstract machines, the Krivine Machine and the ZAM, introducing some variations and studying their differences from a practical point of view.

Contents

1. INTRODUCTION

In the last years, society is becoming ever more dependent on computer systems. People manage their bank accounts via web, are constantly in touch with their contacts thanks to instant messaging applications, and have huge amounts of personal data stored in the *cloud*. All this personal information flowing through computer networks need to be protected by correctly implementing adequate security measures regarding both information transmission and storage. Building strong security systems is not an easy task, because there are lots of parts that must be studied in order to assure the system as a whole behaves as intended.

1.1. Cryptography

One of the most fundamental tools used to build security computer systems is **cryptography**. As a relatively low-level layer in the security stack, it is often the cornerstone over which all the system relies in order to keep being safe. Due to its heavy mathematical roots, cryptography today is a mature science that, when correctly implemented, can provide strong security guarantees to the systems using it.

At this point, one could be tempted of just "using strong, NIST-approved cryptography" and focusing on the security of other parts of the system. The problem is that correctly implementing cryptography is a pretty difficult task on its own, mainly because there is not a one-size-fits-all construction that covers all security requirements. Every cryptographic primitive has its own security assumptions and guarantees, so one must be exceptionally cautious when combining them in order to build larger systems. A given cryptographic construction is usually well suited for some kind of scenarios, and offers little to no security otherwise. In turn, this can produce a false sense of security, potentially worse that not having any security at all.

1.2. Formal Methods

In order to have the best guarantee that some cryptographic construction meets its security requirements, we can use use formal methods to prove that the requirements follow from the assumptions (scenario).

While mathematical proofs greatly enhance the confidence we have in that a given cryptosystem behaves as expected, with the recent increase in complexity it has become more and more difficult to write and verify the proofs by hand, to the point of being practically non-viable. In the recent years there has been an increasing effort in having computers help us write and verify this proofs.

There are various methods and tools for doing this, but one of the most versatile and powerful are the **proof assistants**, which are tools designed to help users develop formal proofs interactively. A proof assistant usually follows the rules of one or more **logics** to derive theorems from previous facts, and the user helps it by giving "hints" on how to proceed. This is in contrast to some other theorem provers that use little or no help from the user, making them easier to use but fundamentally more limited. Coq[1] and Isabelle[2] are examples of widely used proof assistants.

One downside of proof assistants is that they require a considerable amount of knowledge from the user, making them difficult to use for people that is not somewhat fluent with theoretical computer science and logic. This is a significant obstacle to the application of this technologies to other scientific fields that could benefit from adopting the formal methods approach to verification.

1.3. EasyCrypt

EasyCrypt [1] is a toolset conceived to help cryptographers construct and verify cryptographic proofs. It is an open source project[3] being developed currently at IMDEA Software Institute and Inria. It is the evolution of the previous CertiCrypt system [2].

EasyCrypt's works as an interpreter of its own **programming language**, in which the programmer can express all that's needed in order to develop the proofs. At every step of the evaluation, EasyCrypt can output some information regarding the state of the system so that external tools can parse and show it to the user. Together with the fact that the evaluation steps can be reversed, this forms the basis of the

[1] `http://coq.inria.fr/`
[2] `http://isabelle.in.tum.de/`
[3] `https://www.easycrypt.info`

interactivity of the EasyCrypt system: the user can evaluate the program step by step, and if needed, undo it and re-evaluate in the fly.

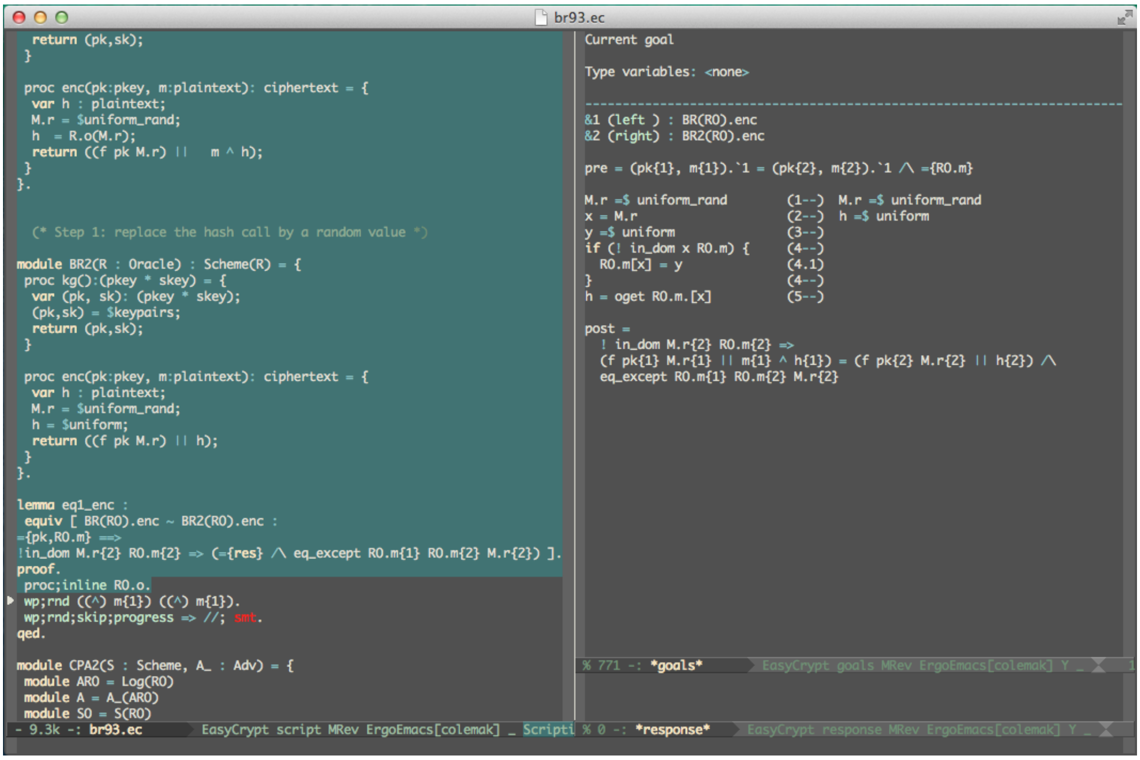

Figure 1.1.: EasyCrypt

The preferred way of working with EasyCrypt is using the **Emacs**[1] text editor, with the **Proof General**[2] interface to proof assistants (figure 1.1). This interface shows both the source code and the EasyCrypt output at the point of evaluation (the already evaluated code is displayed in a different color), and offers standard key combinations for forward/backward code stepping.

As we'll see later (section 2.4), EasyCrypt has different sub-languages for working with different things, e.g., representing games, developing the proofs, etc. One of them is specially relevant in this thesis: the **expression language**. It is the language EasyCrypt uses to define typed values, like quantified formulas, arithmetic expressions, functions, function application and such, and developing proofs relies heavily on the manipulation of this expressions.

[1] `http://www.gnu.org/software/emacs/`
[2] `http://proofgeneral.inf.ed.ac.uk/`

1.4. Contributions

In this work we will study and implement some well-known abstract machines to improve the EasyCrypt's current term rewriting engine. As we will see in the corresponding section (6), the current implementation is an ad-hoc solution that works well, but is monolithic, difficult to extend and somewhat inefficient. Before that, we will introduce some theory to the field of term rewriting and implement both the Krivine Machine and the ZAM, as a way to understand their differences and which is the best reference to improve the EasyCrypt's engine.

Part I.

STATE OF THE ART

2. CRYPTOGRAPHY

In this chapter we will review some concepts related to how cryptographic proofs are built, in order to understand how EasyCrypt works and how proofs are written in it.

2.1. Public-key Encryption

Here we will introduce some basic concepts in asymmetric cryptography, as they will be useful to understand the next sections on EasyCrypt's proof system and sequences of games (section 2.3).

Asymmetric cryptography (also called **Public Key cryptography**), refers to cryptographic algorithms that make use of two different keys, pk (public key) and sk (secret key). There must be some mathematical relationship that allows a specific pair of keys to perform dual operations, e.g., pk to encrypt and sk to decrypt, pk to verify a signature and sk to create it, and so on. A pair of (public, secret) keys can be generated using a procedure called **key generation** ($\mathcal{KG}$).

The encryption ($\mathcal{E}$) and decryption ($\mathcal{D}$) functions work in the following way:

$$\mathcal{E}(pk, M) = C$$
$$\mathcal{D}(sk, C) = M$$

That is, a message (M) can be encrypted using a public key to obtain a ciphertext (C). In turn, a ciphertext (C) can be decrypted using a private key to obtain a message (M). Any **complete** encryption algorithm must satisfy the following property, given that pk and sk were obtained by a call to $\mathcal{KG}$:

$$\mathcal{D}(sk, \mathcal{E}(pk, M)) = M$$

2.2. Proofs by reduction

In cryptography it is usually not possible to prove **perfect security**, as the only possible way to archieve it would be using keys as long as the message (Shannon's theory of information). So, the usual approach is to prove that some cryptographic protocol's security can be **reduced** to the security of some well-known primitive that is believed to be **computationally untractable**. That is, the security relies on the unability of any human being to solve some computationally hard problem. The overall structure of this proofs is represented in the figure 2.1.

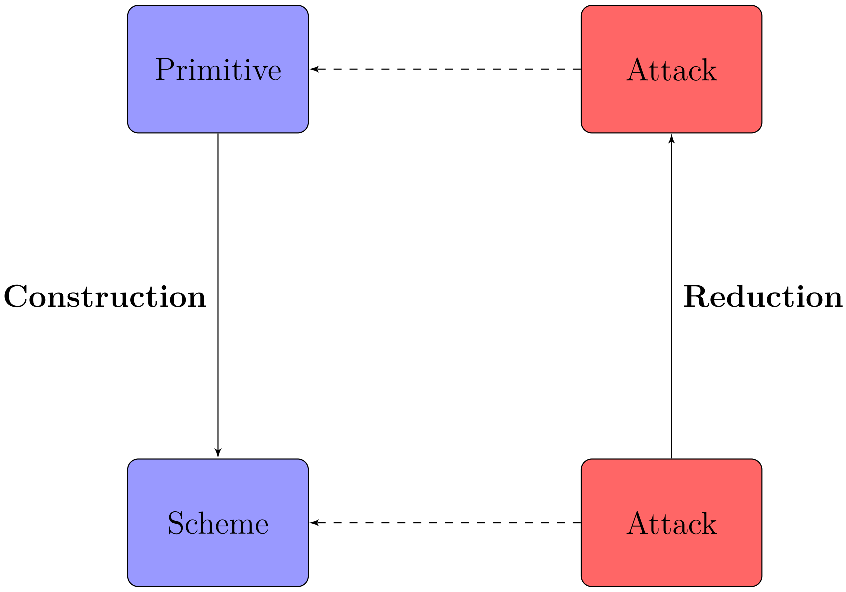

Figure 2.1.: Proofs by reduction

One of the most famous hard problems in cryptography is **integer factorization**. It can be proven that the computational power needed to factor the product of two big primes grows exponentially on the size of the primes, making it a practically impossible task to archieve for sufficiently big prime numbers. The RSA cryptosystem, for example, can be proven secure because it can be reduced to the integer factorization problem.

2.3. Sequences of games

In 2004 [3], Victor Shoup introduced the concept of **sequences of games** as a method of taming the complexity of cryptography related proofs. A **game** is like a program written in a well-defined, probabilistic programming language, and a sequence of games is the result of applying transformations over the initial one.

Every game represents the interaction between a **challenger** and an **adversary**, with the last one being usually encoded as a function (probabilistic program). In the end, we will want the sequence of games to form a proof by reduction (see section 2.2), where the transition of games proves that our system can be reduced, under certain conditions, to some well-known cryptographic primitive. We say that the adversary **wins** when certain event takes place. It generally has to do with his capacity to extract some information or correctly guess some data.

We can define the following game in order to see a practical example of how sequences of games work:

Game 2.1: IND-CPA game (from [2])

$(pk, sk) \leftarrow \mathcal{KG}();$
$(m_0, m_1) \leftarrow A_1(pk);$
$b \stackrel{\$}{\leftarrow} \{0, 1\};$
$c \leftarrow \mathcal{E}(pk, m_b);$
$\tilde{b} \leftarrow A_2(c)$

The game 2.1[1] can be used to define the IND-CPA property of public key encryption schemes. **IND-CPA** (Indistinguishability Under Chosen Ciphertext Attacks) means that the adversary is unable to distinguish between pairs of ciphertexts. The IND-CPA game encodes this fact by letting the adversary chose two messages, encrypting one of them, and making him guess which one was encrypted. In this case, the event that makes the adversary win is that he correctly guesses which of his plaintexts was encrypted. In a full sequence of games, we would start with this game and apply transformations over it trying to preserve the probability of that event (the adversary winning) constant. The final game would hopefully be one in which the calls

2.4. Verification: EasyCrypt

EasyCrypt allows the encoding and verification of game-based proofs, but it has different languages to perform different tasks:

[1] $\mathcal{KG}$ and $\mathcal{E}$ are the key generation and encryption functions provided by the encryption algorithm, respectively, and A_1 is the encoding of the adversary

2.4.1. Specification languages

This are the languages EasyCrypt uses to declare and define types, functions, axioms, games, oracles, adversaries and other entities involved in the proofs.

Expressions

The main specification language of EasyCrypt is the expression language, in which **types** are defined together with **operators** that can be applied to them (or be constant). EasyCrypt follows the usual semicolon notation [4] to denote the typing relationship: «$a : T$» means "a has type T". EasyCrypt has a type system supporting parametric polymorphism: «*int list*» represents a list of integers.

The operators are functions over types, defined with the keyword «**op**» (e.g., «**op** even : nat -> *bool*»). An operator can be applied to some argument by putting them separated by a space: «even 4». Operators can be abstract, i.e., defined without any actual implementation; with semantics given by the definition of axioms and lemmas that describe its observational behavior. Operators are also *curried*, so they support multiple arguments by returning new functions that consume the next one. For example, $f : (A \times B \times C) \to D$ would be encoded as $f : A \to (B \to (C \to D))$, or, by associativity, $f : A \to B \to C \to D$.

In this example (from the current EasyCrypt library) we can see the how actual types and operators are defined in the EasyCrypt's expression language:

Code listing 2.1: Lists (expression language)

```
type 'a list = [
  | "[]"
  | (::) of 'a & 'a list ].

op hd: 'a list -> 'a.
axiom hd_cons (x:'a) xs: hd (x::xs) = x.

op map (f:'a -> 'b) (xs:'a list) =
  with xs = "[]" => []
  with xs = (::) x xs => (f x)::(map f xs).
```

The first line defines the «*list*» type as a sum type with two constructors (cases): the *empty list* and the *construction* of a new list from an existing one and an element that will appear at its head position. The rest of the code defines operators working with lists.

The next line abstractly defines the operator «hd», together with its type. The axiom following it partially specifies the behavior of the «hd» when applied to some list: if the list has the form «x::xs» (element «x» followed by «xs»), the return value is «x». The other case (empty list) is left unspecified.

The last line defines the «map» operator directly, using pattern matching. This operator receives a function and a list, and returns the list consisting of the results of evaluating the function over each element of the list, preserving its order.

Probabilistic expressions Additionally, EasyCrypt defines some standard types and operators to work with probabilistic expressions. The type «'a distr» represents discrete sub-distributions over types. The operator «mu» represents the probability of some event in a sub-distribution:

```
op mu : 'a distr -> ('a -> bool) -> real
```

For example, the uniform distribution over booleans is defined in the EasyCrypt's standard library as follows:

Code listing 2.3: Uniform distribution over bool

```
op dbool : bool distr.
axiom mu_def : forall (p : bool -> bool),
  mu dbool p =
    (1/2) * charfun p true +
    (1/2) * charfun p false.
```

pWhile language

Expression languages are usually not adequate to define games and other data structures as cryptographic schemes and oracles, due to the stateful nature of sequential algorithms. That's why EasyCrypt uses a different language called **pWhile** [5] (probabilistic while) to define them:

Grammar 2.1: pWhile language

$$
\begin{aligned}
\mathcal{C} ::=\ & \mathbf{skip} \\
\mid\ & \mathcal{V} \leftarrow \mathcal{E} \\
\mid\ & \mathcal{V} \overset{\$}{\leftarrow} \mathcal{D}\mathcal{E} \\
\mid\ & \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{C}\ \mathbf{else}\ \mathcal{C} \\
\mid\ & \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C} \\
\mid\ & \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, ..., \mathcal{E}) \\
\mid\ & \mathcal{C};\ \mathcal{C}
\end{aligned}
$$

2.4.2. Proof languages

Judgments

Whenever there is some statement that we want to prove, it must be written as a judgment in some **logic**. Apart from the first order logic expressions, EasyCrypt supports judgments in some logics derived from Hoare logic:

- Hoare Logic (*HL*). These judgments have the following shape:

$$c : P \Longrightarrow Q$$

  where P and Q are assertions (predicates) and c is a statement or program. P is the **precondition** and Q is the **postcondition**. The validity of this kind of Hoare judgment implies that if P holds before the execution of c and it terminates, then Q must also hold.

- Probabilistic Hoare Logic (*pHL*). This is the logic resulting from assigning some probability to the validity of the previously seen Hoare judgments. The probability can be a number or an upper/lower bound:

$$[c : P \Longrightarrow Q] \leq \delta$$
$$[c : P \Longrightarrow Q] = \delta$$
$$[c : P \Longrightarrow Q] \geq \delta$$

- Probabilistic Relational Hoare Logic (*pRHL*). These have the following shape:

$$c_1 \sim c_2 : \Psi \Longrightarrow \Phi$$

  In this case, the pre and postconditions are not standalone predicates, but **relationships** between the memories of the two programs c_1 and c_2. This judgment means that if the precondition Ψ holds before the execution of c_1 and c_2, the postcondition Φ will also hold after finishing its execution.

  This logic is the most complete and useful when developing game-based reduction proofs, because it allows to encode each game transition as a judgment. Twe two games are c_1 and c_2 respectively, and the pre/postconditions refer to the probability of the adversary winning the games.

Tactics

If the judgment is declared as an axiom, it is taken as a fact and does not need to be proven. Lemmas, however, will make EasyCrypt enter in "proof mode", where it stops reading declarations, takes the current judgment as a goal and and starts accepting **tactics** until the current goal is trivially true. Tactics are indications on what rules EasyCrypt must apply to transform the current goal.

This is a simplified example of proof from the EasyCrypt's library, where we can see the tactics applied between the «proof» and «qed» keywords:

Code listing 2.4: Tactics usage

```
lemma cons_hd_tl :
  forall (xs:'a list),
    xs <> [] => (hd xs)::(tl xs) = xs.
proof.
  intros xs.
  elim / list_ind xs.
  simplify.
  intros x xs' IH h {h}.
  rewrite hd_cons.
  rewrite tl_cons.
  reflexivity.
qed.
```

3. TERM REWRITING

3.1. Introduction

In computing and programming languages it is common to encounter scenarios where objects (e.g., code) get transformed gradually for simplification, to perform a computation, etc. The transformations must obey some rules that relate “input” and “output” objects, that is, how to make the transition from one object to the other. When we take both the objects and the rules and study them as a whole, the result is an **abstract reduction system** [6].

This is a very general framework, but for what this work is concerned, we are specially interested in reasoning about rewriting of (λ-)terms. In the end we will want to improve how EasyCrypt is able to reduce terms in its expression language, so we will start by understanding Lambda Calculus and how it is reduced, because it is very similar to how EasyCrypt represents its own terms.

3.2. Lambda Calculus

The **Lambda Calculus** [7] is a formal system developed by Alonzo Church in the decade of 1930 as part of his research on the foundations of mathematics and computation (it was later proven to be equivalent to the Turing Machine). In its essence, the Lambda Calculus is a simple term rewriting system that represents computation through **function application**.

Following is the grammar representing **λ-terms** (lambda-terms, $\mathcal{T}$):

Grammar 3.2: Lambda Calculus

$\mathcal{T} ::= x$	variable
$\mid\ (\lambda x.\mathcal{T})$	abstraction
$\mid\ (\mathcal{T}_1\ \mathcal{T}_2)$	application
$x ::= v_1, v_2, v_3, ...$	(infinite variables available)

Intuitively, the **abstraction** rule represents function creation: take an existing term ($\mathcal{T}$) and parameterize it with an argument (x). The variable x binds every instance of the same variable on the body, which we say are **bound** instances. The **application** rule represents function evaluation ($\mathcal{T}_1$) with an actual argument ($\mathcal{T}_2$).

Seen as a term rewriting system, the Lambda Calculus has some reduction rules that can be applied over terms in order to perform the computation.

3.2.1. Reduction rules

The most prominent reduction rule in Lambda Calculus is the **beta reduction**, or β-reduction. This rule represents function evaluation, and can be outlined in the following way:

$$\beta\text{-RED}\ \frac{}{((\lambda x.\mathcal{T}_1)\ \mathcal{T}_2) \underset{\beta}{\rightsquigarrow} \mathcal{T}_1[x := \mathcal{T}_2]}$$

An application with an abstraction in the left-hand side is called a **redex**, short for “reducible expression”, because it can be β-reduced following the rule[1] . The semantics of the rule match with the intuition of function application: the result is the body of the function with the formal parameter replaced by the actual argument. It has to be noted that even when a term is *not* a redex, it can contain some other sub-expression that indeed is; the problem of knowing where to apply each reduction will be addressed in section 3.4.

The **substitution operation** $\mathcal{T}_1[x := \mathcal{T}_2]$ replaces x by $\mathcal{T}_2$ in the body of $\mathcal{T}_1$, but we have to be careful in its definition, because the “obvious/naïve” substitution process can lead to unexpected results. For example, $(\lambda x.y)[y := x]$ would β-reduce to $(\lambda x.x)$, which is not the expected result: the new x in the body has been **captured** by the argument and the function behavior is now different.

The solution to this problem comes from the intuitive idea that “the exact choice of names for bound variables is not really important”. The functions $(\lambda x.x)$ and $(\lambda y.y)$ behave in the same way and thus should be considered equal. The **alpha equivalence** (α-equivalence) is the equivalence relationship that expresses this idea through another rule: the **alpha conversion** (α-conversion). The basic definition of this rule is the following:

[1]The «$\rightsquigarrow$» symbol means “reduces to”, and «$\overset{\star}{\rightsquigarrow}$» is its symmetric and transitive closure (“reduces in 0 or more steps to”)

$$\alpha\text{-CONV}\ \frac{y \notin \mathcal{T}}{(\lambda x.\mathcal{T}) \underset{\alpha}{\rightsquigarrow} (\lambda y.\mathcal{T}[x := y])}$$

So, to correctly apply a β-reduction, we will do **capture-avoiding substitutions**: if there is the danger of capturing variables during a substitution, we will first apply α-conversions to change the problematic variables by fresh ones.

Another equivalence relation over lambda terms is the one defined by the **eta conversion** (η-conversion), and follows by the extensional equivalence of functions in the calculus:

$$\eta\text{-CONV}\ \frac{x \notin FV(\mathcal{T})}{(\lambda x.\mathcal{T}\ x) \underset{\eta}{\leftrightsquigarrow} \mathcal{T}}$$

In general, we will treat α-equivalent and η-equivalent functions as interchangeable.

3.3. Normal forms

In abstract rewriting systems, a term a is in **normal form** whenever it can not be reduced any further. That is, there does not exist any other term b such that $a \rightsquigarrow b$.

When a λ-term has no subexpressions that can be reduced, it is already in normal form. There are also three additional notions of normal form in Lambda Calculus:

- **Weak normal form**: λ-terms with form $(\lambda x.\mathcal{T})$ are not reduced
- **Head normal form**: λ-terms with form $(x\ \mathcal{T})$ are not reduced
- **Weak head normal form**: neither λ-terms in weak or head normal form are not reduced

The Lambda Calculus is **not normalising**, so there is not any guarantee that any normal form exists for a given term.

3.4. Reduction Strategies

When reducing λ-terms, a **reduction strategy** [8] is also needed to remove the ambiguity about which sub-expression on a given term should be reduced next. This is usually an algorithm that given some reducible term (redex), points to the redex inside it that sould be reduced next.

Each reduction strategy knows when to stop searching based on some normal form (of the four we've already seen).

Two of the most common reduction strategies, and the ones we will be more concerned with in this work, are the following:

- **Call-by-name** reduces the leftmost outermost redex, unless it is in weak head normal form. Due to the head normal form, the evaluation is non-strict.
- **Call-by-value** reduces the leftmost innermost redex, unless it is in weak normal form. The evaluation is strict.
- **Applicative order** reduces the leftmost innermost redex, unless it is in normal form. The evaluation is strict.

The Lambda Calculus has an interesting property called **confluence**, that means that whenever some term has more than one possible reduction, there exists another term to which both branches will converge in the end:

$$\textsc{Confluence}\ \frac{a \overset{\star}{\rightsquigarrow} b_1 \qquad a \overset{\star}{\rightsquigarrow} b_2}{\exists c.(b_1 \overset{\star}{\rightsquigarrow} c \wedge b_2 \overset{\star}{\rightsquigarrow} c)}$$

What this means is that the reduction order does not really matter unless one of them leads to non-termination (an infinite chain). Non-strict strategies, such as call-by-name, helps avoiding non-terminating reductions thanks to its head normal form (which does not evaluate function arguments until needed).

3.5. Abstract Machines

In order to actually implement the reduction over λ-terms, there are some different ways with different advantages and drawbacks, the most "extreme" being direct interpretation of source code and compilation to native instructions of a real machine.

An intermediate point is simulating an **abstract machine** to which we can feed a sequence of instructions (requiring a previous compilation process) or the original language itself if it is simple enough. This approach is useful because it is more portable than native code generation while being more efficient than plain interpretation.

There can be different abstracts machine for the same language, differing not only in their implementation details but in the reduction strategies they implement, so the output can also be different, with some machines implementing **stronger** reductions than others (i.e., to normal form).

In the next part of the thesis we will study and implement two widely-known abstract machines to reduce λ-terms, and improve the EasyCrypt current reduction machinery by applying the same concepts.

Part II.

IMPLEMENTATION

4. KRIVINE MACHINE

To begin our study of the implementation of abstract machines, we will start with the Krivine Machine. It is a relatively simple and well-known model that will help us see the steps that we need to take in order to implement a real abstract machine. We will be using the OCaml language from now on (not only in this section but also in the next ones), and while snippets of code will be presented to illustrate the concepts, the full code is available in the annex 8.1 for reference.

The Krivine Machine [9] is an implementation of the weak-head call-by-name reduction strategy for pure lambda terms. What that means is that:

- The Krivine Machine reduces pure (untyped) terms in the Lambda Calculus
- The reduction strategy it implements is call-by-name, reducing first the left-most outermost term in the formula
- It stops reducing whenever the formula is in weak-head normal form, that is:
  - does not further reduce abstractions: $(\lambda x.\mathcal{T})$
  - does not reduce arguments before substitution $(x\,\mathcal{T})$

4.1. Target language

The first thing we need to have is an encoding of the language we will be reducing, in this case the Lambda Calculus. We will define a module, **Lambda**, containing the data structure and basic operations over it:

```
type symbol = string * int
let show_symbol (s, _) = s

module Lambda = struct
    type t = Var of symbol | App of t * t | Abs of symbol * t
    let rec show = match m with
      | Var x -> show_symbol x
      | App (m1, m2) -> "(" ^ show m1 ^ " " ^ show m2 ^ ")"
      | Abs (x, m) -> "(λ" ^ show_symbol x ^ "." ^ show m ^ ")"
end
```

(From now on, the auxiliar (e.g., pretty-printing) functions will be ommited for brevity.)

The module Lambda encodes the pure Lambda Calculus with its main type «t» [1]. Variables are just symbols (strings tagged with an integer so that they're unique), Applications are pairs of terms and Abstractions are pairs of symbols (the binding variable) and terms (the body). As the Krivine Machine usually works with expressions in **de Bruijn** notation[2], we'll need to write an algorithm to do the conversion of variables. To be sure we do not accidentally build terms mixing the two notatios, we'll create another module, **DBILambda**, with a different data type to represent them:

```
let rec find_idx_exn x = function
  | [] -> raise Not_found
  | (y::ys) -> if x = y then 0 else 1 + find_idx_exn x ys

module DBILambda = struct
    type dbi_symbol = int * symbol
    type t = Var of dbi_symbol | App of t * t | Abs of symbol * t

    let dbi dbis x = (find_idx_exn x dbis, x)

    let of_lambda =
      let rec of_lambda dbis = function
        | Lambda.Var x -> let (n, x) = dbi dbis x in Var (n, x)
        | Lambda.App (m1, m2) -> App (of_lambda dbis m1, of_lambda dbis m2)
        | Lambda.Abs (x, m) -> Abs (x, of_lambda (x :: dbis) m)
      in of_lambda []
end
```

The new variables, of type «dbi_symbol», store the de Bruijn number together with the previous value of the symbol, to help debugging and pretty-printing. The function «of_lambda» accepts traditional lambda terms (Lambda.t) and returns its representation as a term using de Bruijn notation (DBILambda.t). Now we are ready to implement the actual reduction.

[1] Naming the main type of a module «t» is an idiom when using modules in OCaml

[2] The de Bruijn's notation gets rid of variable names by replacing them by the number of "lambdas" between it and the lambda that is binding the variable. For example, $(\lambda x.(\lambda y.(x\ y)))$ will be written in de Bruijn notation as $(\lambda.(\lambda.(1\ 0)))$

4.2. Reduction

The Krivine Machine has a state (C, S, E) consisting on the **code** it is evaluating (C), an auxiliar **stack** (S) and the current **environment** (E). The code is just the current Lambda expression, the stack holds **closures** (not yet evaluated code + the environment at the time the closure was created), and an environment that associates variables (de Bruijn indices) to values (closures).

The typical description of the Krivine Machine [10] is given by the following set of rules:

$$
\begin{array}{lcl}
(MN, S, E) & \rightsquigarrow & (M, (N, E) :: S, E) \\
(\lambda M, N :: S, E) & \rightsquigarrow & (M, S, N :: E) \\
(i+1, S, N :: E) & \rightsquigarrow & (i, S, E) \\
(0, S, (M, E_1) :: E_2) & \rightsquigarrow & (M, S, E_1)
\end{array}
$$

In the previous diagram, S and E are both described as **lists**, with the syntax (H :: T) meaning "list whose head is H and tail is T". The stack S is a list of closures that implements the push/pop operations over its head. The environment is a list whose i-th position stores the variable with de Bruijn index i.

- In the first rule, to evaluate an application MN, the machine builds a closure from the argument N and the current environment E, pushes it into the stack, and keeps on reducing the function M.

- To reduce an abstraction λM, the top of the stack is moved to the environment and proceeds with the reduction of the body M. What this means is that the last closure in the stack (the function argument) is now going to be the variable in position (de Bruijn index) 0.

- The last two rules rearch through the environment to find the closure corresponding to the current variable. Once it is found, the closure's code M and environment E_1 replace the current ones.

From this specification we can write a third module, **KM**, to define the data structures (state, closure, stack) and implement the symbolic reduction rules of the Krivine Machine:

```
module KM = struct
    open DBILambda

    type st_elm = Clos of DBILambda.t * stack
    and stack = st_elm list

    type state = DBILambda.t * stack * stack

    let reduce m =
      let rec reduce (st : state) : DBILambda.t =
        match st with
        (* Pure lambda calculus *)
        | (Var (0, _), s, Clos (m, e') :: e) -> reduce (m, s, e')
        | (Var (n, x), s, _ :: e) -> reduce (Var (n-1, x), s, e)
        | (App (m1, m2), s, e) -> reduce (m1, Clos (m2, e) :: s, e)
        | (Abs (_, m), c :: s, e) -> reduce (m, s, c :: e)
        (* Termination checks *)
        | (m, [], []) -> m
        | (_, _, _) -> m in
      reduce (m, [], [])
end
```

At this point, we have a working implementation of the Krivine Machine and can execute some tests to see that everything works as expected. We'll write some example λ-terms:

```
let ex_m1 =        (* (λx.((λy.y) x)) *)
  let x = symbol "x" in
  let y = symbol "y" in
  Abs (x, App (Abs (y, Var y), Var x))

let ex_m2 =        (* (((λx.(λy.(y x))) (λz.z)) (λy.y)) *)
  let x = symbol "x" in
  let y = symbol "y" in
  let z = symbol "z" in
  App (App (Abs (x, Abs (y, App (Var y, Var x))), Abs (z, Var z)), Abs (y, Var y))
```

And lastly, a helper function that accepts a λ-term, translates it to de Bruijn notation and reduces it, outputting the results:

```
let dbi_and_red m =
  let dbi_m = DBILambda.of_lambda m in
  print_endline ("# Lambda term:\n    " ^ DBILambda.show dbi_m);
  let red_m = KM.reduce dbi_m in
  print_endline ("# Reduced term:\n    " ^ DBILambda.show red_m);
  print_endline "-----------------------------------------------\n"
```

The results:

```
ocaml> List.iter dbi_and_red [ex_m1; ex_m2];;

# Lambda term:
    (λx.((λy.y) x))
# Reduced term:
    (λx.((λy.y) x))
----------------------------------------------------------

# Lambda term:
    (((λx.(λy.(y x))) (λz.z)) (λy.y))
# Reduced term:
    (λz.z)
----------------------------------------------------------
```

As we can see, the first term is not reduced because it is already in weak head normal form (abstraction). The second term reduces as we would expect. As a sidenote, we can easily tweak the DBILambda module to show the real de Bruijn variables:

```
ocaml> List.iter dbi_and_red [ex_m1; ex_m2];;

# Lambda term:
    (λ.((λ.0) 0))
# Reduced term:
    (λ.((λ.0) 0))
----------------------------------------------------------

# Lambda term:
    (((λ.(λ.(0 1))) (λ.0)) (λ.0))
# Reduced term:
    (λ.0)
----------------------------------------------------------
```

4.3. Extended version

Now that we have a working Krivine Machine over basic Lambda Terms, we want to extend it to work with some extensions: **case expressions** and **fixpoints**.

4.3.1. Case expressions

First, we will need to extend our definition of the Lambda Calculus to support constructors and case expressions, both in Lambda and DBILambda. Constructors are just atomic symbols, optionally parameterized by some arguments, used to encode arbitrary data. Case expressions are used to “destructure” constructors and extract the value of their parameters:

```
module Lambda = struct
    type t = Var of symbol | App of t * t | Abs of symbol * t
             (* constructors / case expressions *)
             | Constr of t constr
             | Case of t * (symbol constr * t) list
    and 'a constr = symbol * 'a list

    (* ... *)
end

module DBILambda = struct
    type t = Var of dbi_symbol | App of t * t | Abs of symbol * t
             (* constructors / case expressions *)
             | Constr of t constr
             | Case of t * (symbol constr * t) list
    and 'a constr = symbol * 'a list

    let of_lambda =
      let rec of_lambda dbis = function
        (* ... *)
        | Lambda.Constr (x, ms) -> Constr (x, List.map (of_lambda dbis) ms)
        | Lambda.Case (m, bs) -> Case (of_lambda dbis m,
                                       List.map (trans_br dbis) bs)
      in of_lambda []

    (* ... *)
end
```

The basic approach to implement the reduction will be the same as when reducing applications in λ-terms: when encountering a case expression, create a custom closure “CaseCont” containing the branches and push it into the stack. When a constructor is encountered, if there is a CaseCont closure in the stack, the machine will iterate over the branches in the closure until it finds one whose symbol matches with the constructor, and evaluate the body:

```ocaml
module KM = struct
    (* ... *)
    type st_elm = Clos of DBILambda.t * stack
                (* Specific closure for case expressions *)
                | CaseCont of (symbol DBILambda.constr * DBILambda.t) list * stack

    let reduce m =
      let rec reduce (st : state) : DBILambda.t =
        match st with
        (* ... *)
        (* Case expressions (+ constructors) *)
        | (Case (m, bs), s, e) -> reduce (m, CaseCont (bs, e) :: s, e)
        | (Constr (x, ms), CaseCont (((x', args), m) :: bs, e') :: s, e)
             when x == x' && List.length ms == List.length args ->
           reduce (List.fold_left (fun m x -> Abs (x, m)) m args,
                   map_rev (fun m -> Clos (m, e)) ms @ s, e')
        | (Constr (x, ms), CaseCont (_ :: bs, e') :: s, e) ->
           reduce (Constr (x, ms), CaseCont (bs, e') :: s, e)
        | (Constr (x, ms), s, e) ->
           Constr (x, List.map (fun m -> reduce (m, s, e)) ms)
      reduce (m, [], [])
end
```

Note that te last rule reduces the terms inside a constructor even when it is not being pattern matched. It can be deleted to more closely resemble a weak-head normal form behavior.

With this new functionality we can encode Peano arithmetic by using the constructors «z» and «s», and try some examples of pattern matching:

```ocaml
(* Peano arithmetic helpers *)
let z = symbol "z"
let s = symbol "s"

let rec peano_add n x =
  if n == 0 then x else peano_add (n-1) (Constr (s, [x]))

let peano_of_int ?(base=Constr (z, [])) n = peano_add n base
```

```
# Lambda term:
    ((λc.(case c of (Some(x) → x)
                    (None() → c)))
     Some(s(z())))
# Reduced term:
    s(z())
----------------------------------------------------------

# Lambda term:
    ((λc.(case c of (triple(x,y,z) → y)))
     triple(s(z()), s(s(z())), s(s(s(z())))))
# Reduced term:
    s(s(z()))
----------------------------------------------------------
```

4.3.2. Fixpoints

The second extension we are going to implement in our Krivine Machine is the ability to support fixpoints, that is, recursion. Again, we extend the data structures of the Lambda Calculus. As the extension to DBILambda follows trivially, we will omit it here:

```
module Lambda = struct
    type t = Var of symbol | App of t * t | Abs of symbol * t
           | Constr of t constr
           | Case of t * (symbol constr * t) list
           (* fixpoints *)
           | Fix of symbol * t
    (* ... *)
end
```

The "Fix" constructor is parameterized by a symbol and another lambda term. The symbol is the name of the fixpoint, and will expand to itself when referenced inside the term. Let's see an example:

```
fix(λf.λc. case c of (s(x) → s(s(f x)))
                     (z → z))
    s(s(s(z)))
```

Here, the name of the fixpoint is «f», and is an implicitly-passed argument that references to the term itself («fix(...)»). So, in this case, «f» will be bound to «fix(λf. λc. case c of ...)» and «c» will be bound to «s(s(s(z)))» initially.

The reduction itself is simpler than the previous case:

```
module KM = struct
    (* ... *)
    type st_elm = Clos of DBILambda.t * stack
                | CaseCont of (symbol DBILambda.constr * DBILambda.t) list * stack
                (* Specific closure for fixpoints *)
                | FixClos of symbol * DBILambda.t * stack

    let reduce m =
      let rec reduce (st : state) : DBILambda.t =
        match st with
        (* ... *)
        (* Fixpoints *)
        | (Var (0, _), s, FixClos (f, m, e') :: e) ->
           reduce (m, s, FixClos (f, m, e') :: e')
        | (Fix (x, m), s, e) -> reduce (m, s, FixClos (x, m, e) :: e)
      reduce (m, [], [])
end
```

Again, here we create a new closure when reducing the Fix constructor. Then, whenever some variable refers to a fixpoint closure, the result is its body, with the side effect of keeping the closure it in the environment (so that it is automatically bound to the first argument «f» of itself).

The fixpoint tests:

```
# Lambda term:
    (fix(λf.(λc.(case c of (s(x) → s(s((f x))))
                          (z() → z())))))
     s(s(s(z()))))
# Reduced term:
    s(s(s(s(s(s(z())))))))
-----------------------------------------------------------

# Lambda term:
    ((fix(λf.(λg.(λc.(case c of (s(x) → (g ((f g) x)))
                               (z() → z()))))))
         (λy.s(s(s(y)))))
     s(s(s(z()))))
# Reduced term:
    s(s(s(s(s(s(s(s(s(z()))))))))))
-----------------------------------------------------------
```

The first test is the example we have previously seen; the fixpoint receives a number in Peano notation and multiplies it by two by iterating over its structure. The second one generalizes it by accepting a function «g» that is applied once for each iteration over the number: in this case, «g» adds 3 to its argument, so the whole term is a “by 3” multiplier.

5. ZAM

The ZAM (ZINC Abstract Machine) [11] is a call-by-value variant of the Krivine Machine, and currently powers the bytecode interpreter of the Caml Light and OCaml languages. We will implement the version introduced in [12] as a previous step before tackling the implementation of EasyCrypt's new reduction machinery. Again, the full code is available in the annex 8.2.

As with the Krivine Machine before, it will be able to handle extended λ-terms with case expressions and fixpoints, but this time the machine will interpret actual abstract code instead of implementing symbolic reduction, so we will need an extra step to compile the the λ-terms to machine code. Also, we will skip the progressive exposition of the basic and extended machine, as it has already been done in the previous section, and just implement the final version. To finish, we will show how to use it to archieve **strong reduction**.

5.1. Target language

As the reference work [12] aimed to improve the performance of strong reduction in proof assistants, this version of the ZAM works over type-erased terms of the Calculus of Constructions [13], adding **inductive types** (for our purposes, equivalent to the previously implemented algebraic data types) and **fixpoints**.

For this task, we will define a module called **CCLambda** with a type encoding the terms of our language:

```
module CCLambda = struct
    type t = Var of symbol | App of t * t | Abs of symbol * t
            | Constr of t constr | Case of t * (symbol constr * t) list
            | Fix of symbol * symbol list * symbol * t
    and 'a constr = symbol * 'a list

    (* ... *)
end
```

The only difference we can see here with respect to the encoding of terms in the K-Machine (section 4.1) is the more elaborate fixpoints. Even though our λ-terms are not typed, the Calculus of Constructions' fixpoints need an argument ("guard") to structurally to the recursion and prevent infinite unrolling. We will represent fixpoints as «Fix (f, xs, c, m)», where «f» is the symbol (bound in «m») referring to the fixpoint itself, «xs» is the argument list, «c» is the guard (a constructor), and «m» is the λ-term.

5.2. Compilation

Unlike the previous implementation, in this case we are going to implement a more efficient version. Instead of symbolically evaluating the λ-terms we need an extra step to compile them to some intermediate code Now we need to be able to compile λ-terms to instructions targeting the ZAM runtime, which will do the actual reduction.

This is the type encoding the machine instructions:

```
module WeakRed = struct
    open CCLambda

    type dbi = symbol * int
    type instr =
      | ACCESS of dbi
      | CLOSURE of instrs
      | ACCLOSURE of symbol
      | PUSHRETADDR of instrs
      | APPLY of int
      | GRAB of symbol
      | RETURN
      | MAKEBLOCK of symbol * int
      | SWITCH of (symbol constr * instrs) list
      | CLOSUREREC of (symbol * symbol list * symbol) * instrs
      | GRABREC of symbol
    and instrs = instr list
    and mval =
      | Cons of mval constr
      | Clos of instrs * env
      | Ret of instrs * env * int
    and env = mval list
    and stack = mval list
    and state = {
        st_c : instrs;
        st_e : env;
        st_s : stack;
        st_n : int;
    }

    (* ... *)
end
```

The «WeakRed.compile» function in the code does the actual work of translating λ-terms to a sequence of instructions.

5.3. Reduction

The figure 5.1 details the reduction rules. The implementation is straightforward (although contrived) and follows the same structure as the Krivine Machine. I refer the reader to the annex in order to see how this rules are actually encoded in our program.

Code	Environment	Stack	Num. arg.	
$\texttt{ACCESS}(i);c$	e	s	n	
c	e	$e(i).s$	n	
$\texttt{CLOSURE}(c');c$	e	s	n	
c	e	$[T_\lambda : c', e].s$	n	
$\texttt{PUSHRETADDR}(c');c$	e	s	n	
c	e	$\langle c', e, n\rangle.s$	n	
$\texttt{APPLY}(i)$	e	$[T : c', e'] : s$	n	
c'	e'	s	i	
$\texttt{GRAB};c$	e	$v : s$	$n+1$	
c	$v.e$	s	n	
$c_0 = \texttt{GRAB};c$	e	$\langle c', e', n'\rangle.s$	0	
c'	e'	$[T_\lambda : c_0, e] : s$	n'	
$\texttt{RETURN}$	e	$v.\langle c', e', n'\rangle.s$	0	
c'	e'	$v.s$	n'	
$\texttt{RETURN}$	e	$[T : c', e'].s$	n	if $n > 0$
c'	e'	s	n	
$\texttt{ACCU}$	k	$v_1 \dots v_n.\langle c', e', n'\rangle : s$	n	
c'	e'	$[0 : \texttt{ACCU}, [1 : k, v_1, \dots, v_n]].s$	n'	
$\texttt{MAKEBLOCK}(T,m);c$	e	$v_1 \dots v_m.s$	n	
c	e	$[T : v_1, \dots, v_m].s$	n	
$\texttt{SWITCH}(c_1, \dots, c_m)$	e	$[T : v_1, \dots, v_p].s$	n	if $1 \leq T \leq m$
c_T	$v_p \dots v_1.e$	s	0	
$c_0 = \texttt{SWITCH}(c_1, \dots, c_m)$	e	$[0 : \texttt{ACCU}, k].s$	n	
$\texttt{RETURN}$	e	$[0 : \texttt{ACCU}, [2 : k, c_0, e]].s$	0	
$\texttt{CLOSUREREC}(c');c$	e	s	n	
c	e	$v.s$	n	where $v = [T_\lambda : c', v.e]$
$\texttt{GRABREC};c$	e	$[T : \vec{v}].s$	$n+1$	if $T > 0$
c	$[T : \vec{v}].e$	s	n	
$\texttt{GRABREC};c$	e	$[0 : \texttt{ACCU}, k].s$	$n+1$	
$\texttt{RETURN}$	e	$[0 : \texttt{ACCU}, [3 : c, e, k]].s$	n	
$c_0 = \texttt{GRABREC};c$	e	$\langle c', e', n'\rangle.s$	0	
c'	e'	$[T_\lambda : c_0, e].s$	n'	

Figure 5.1.: ZAM reduction rules (from [12])

At this point, we can try some code examples to see the results:

```
WEAK REDUCTION TEST
Lambda term:
    ((λx.x) (λy.(((λz.z) y) (λt.t))))
Reduced term:
    (λy.(((λz.z) y) (λt.t)))
----------------------------------------------------------

WEAK REDUCTION TEST
Lambda term:
    (((λf.(λx.(f (f x)))) (λy.y)) (λz.z))
Reduced term:
    (λz.z)
----------------------------------------------------------

WEAK REDUCTION TEST
Lambda term:
    ((λc.(case c of (Cons(x,xs) → x)
                    (Nil() → Nil())))
     Cons((λm.m), Nil()))
Reduced term:
    (λm.m)
----------------------------------------------------------

WEAK REDUCTION TEST
Lambda term:
    (fix_0(λf.λc. (case c of (s(x) → s(s((f x))))
                  (z() → z())))
     s(s(s(z()))))
Reduced term:
    s(s(s(s(s(s(z()))))))
----------------------------------------------------------
```

5.4. Strong reduction

Once we have the machinery to perform call-by-value reduction (that is, until it reaches a weak normal form) we can use a callback procedure to apply it repeatedly and archieve strong normalization. The actual implementation is in the module «StrongRed» (annex 8.2).

The most interesting detail about using the readback procedure is the need to support **accumulators** as another case of λ-terms. Accumulators are just terms that cannot get reduced, and are self-propagating: whenever a function is applied to an accumulator, the result is an accumulator containing the application of the function to the previous accumulator. The callback procedure needs it in order to reach the previously-unreachable terms inside an abstraction. Here are the modifications to the data structures:

```
module CCLambda = struct
    type t = Var of symbol | App of t * t | Abs of symbol * t
           | Constr of t constr | Case of t * (symbol constr * t) list
           | Fix of symbol * symbol list * symbol * t
           | Acc of t    (* Accumulators *)
    and 'a constr = symbol * 'a list

    (* ... *)
end

module WeakRed = struct
    open CCLambda

    type dbi = symbol * int
    type instr =
      | ACCESS of dbi
      | CLOSURE of instrs
      | ACCLOSURE of symbol
      | PUSHRETADDR of instrs
      | APPLY of int
      | GRAB of symbol
      | RETURN
      | MAKEBLOCK of symbol * int
      | SWITCH of (symbol constr * instrs) list
      | CLOSUREREC of (symbol * symbol list * symbol) * instrs
      | GRABREC of symbol
    and instrs = instr list
    and accum =
      | NoVar of symbol
      | NoApp of accum * mval list
      | NoCase of accum * instrs * env
      | NoFix of accum * instrs * env
    and mval =
      | Accu of accum
      | Cons of mval constr
      | Clos of instrs * env
      | Ret of instrs * env * int
    and env = mval list
    and stack = mval list
    and state = {
        st_c : instrs;
        st_e : env;
        st_s : stack;
        st_n : int;
    }

    (* ... *)
end
```

And we pass the tests again:

```
STRONG REDUCTION TEST
Lambda term:
    ((λx.x) (λy.(((λz.z) y) (λt.t))))
Reduced term:
    (λy.(y (λt.t)))
----------------------------------------------------------

STRONG REDUCTION TEST
Lambda term:
    (((λf.(λx.(f (f x)))) (λy.y)) (λz.z))
Reduced term:
    (λz.z)
----------------------------------------------------------

STRONG REDUCTION TEST
Lambda term:
    ((λc.(case c of (Cons(x,xs) → x)
                    (Nil() → Nil())))
     Cons((λm.m), Nil()))
Reduced term:
    (λm.m)
----------------------------------------------------------

STRONG REDUCTION TEST
Lambda term:
    (fix_0(λf.λc. (case c of (s(x) → s(s((f x))))
                             (z() → z())))
     s(s(s(z()))))
Reduced term:
    s(s(s(s(s(s(z()))))))
----------------------------------------------------------
```

As we were expecting, the first term is now **strongly reduced** to normal form.

6. REDUCTION IN EASYCRYPT

The current approach that EasyCrypt uses to reduce terms is the iteration over the formula and application of ad-hoc transformations based on the information provided by its type and global state.

To archieve strong reduction, EasyCrypt uses a similar "read-back" protocol to the one we've already seen in the ZAM: repeatedly application of a function that iterates the current formula and tries to reduce it in a call-by-value fashion. When there is no other expression to be reduced, the read-back procedure stops and the normalized formula is returned.

6.1. Target language

Each formula is composed by some metadata (type, free variables, unique tag) together with a **node** that holds the actual structure of the term:

```
type f_node =
  | Fquant  of quantif * bindings * form
  | Fif     of form * form * form
  | Flet    of lpattern * form * form
  | Fint    of BI.zint
  | Flocal  of EcIdent.t
  | Fpvar   of EcTypes.prog_var * memory
  | Fglob   of EcPath.mpath     * memory
  | Fop     of EcPath.path * ty list
  | Fapp    of form * form list
  | Ftuple  of form list
  | Fproj   of form * int
  | FhoareF of hoareF
  | FhoareS of hoareS
  | FbdHoareF of bdHoareF
  | FbdHoareS of bdHoareS
  | FequivF of equivF
  | FequivS of equivS
  | FeagerF of eagerF
  | Fpr of pr
```

As there are so much types and corner cases, we will briefly explain what are the most important constructors and what are they for:

- **FQuant**: They serve both as universal/existential quantifiers (forall / exists) and lambda abstractions, depending on the value of the «quantif» parameter (Lforall, Lexists, Llambda, respectively).

- **Fif**: Conditionals.

- **Fint**: Literal integers.

- **Flocal**: Local variables.

- **Fop**: Operators: as explained in the introduction (section 2.4.1). The actual code must be obtained by resolving its path.

- **Fapp**: Function application (to multiple arguments).

6.2. Reduction rules

As the term language of EasyCrypt is more complex than standard Lambda Calculus, it has some reduction rules we have not seen befone:

- δ-reduction (**delta**): used to unfold global definitions. Affects operators («Fop»).

- ζ-reduction (**zeta**): used to unfold a let expression in its body. Affects let expressions («Flet»).

- ι-reduction (**iota**): used to unfold a case expression. Affects conditionals («Fif»), operators («Fop»).

- Logical reduction: used to evaluate logic expressions (And, Or, ...). Affects operators («Fop»).

A structure containing information about which of this reductions must be done is passed to every reduction procedure:

```
type reduction_info = {
  beta    : bool;
  delta_p : (path  -> bool);
  delta_h : (ident -> bool);
  zeta    : bool;
  iota    : bool;
  logic   : bool;
}
```

6.3. Reduction

The reduction machinery is implemented in an EasyCrypt module called **EcReduction**. The main entry point is the function «**h_red**», which accepts the target formula and a «reduction_info» structure (see previous section) and returns the reduced formula according to it. An important point is that «h_red» only reduces until **weak normal form**, and there is another callback procedure that calls it repeatedly as we've already seen with the ZAM machine. So, we need to take that function and replace it with a ZAM-like machine to do only the weak reduction.

This is a short fragment of the «h_red» function:

```
let rec h_red_old ri env hyps f =
  match f.f_node with
    (* β-reduction *)
  | Fapp ({ f_node = Fquant (Llambda, _, _)}, _) when ri.beta ->
      f_betared f

    (* ζ-reduction *)
  | Flocal x -> reduce_local ri hyps x

    (* ζ-reduction *)
  | Fapp ({ f_node = Flocal x }, args) ->
      f_app_simpl (reduce_local ri hyps x) args f.f_ty

  (* ... *)
```

Although it is actually a pretty long function (around 220 lines of code), the structure is simple: a pattern matching over the structure of the current formula. We will start by defining the state of the new abstract machine and replacing the pattern matching by a recursive function over an initial state:

```
type st_elm =
  | Clos of form * menv
  | ClosCont of bindings
  | IfCont of form * form
and stack = st_elm list
and menv = (EcIdent.t, form) Hashtbl.t

let rec h_red ri env hyps f =
  let iter st =
    match (st : EcFol.form * stack * menv) with

    (* β-red *)
    | ({ f_node = Fapp (f, fs) }, s, e) when ri.beta ->
       iter (f, List.map (fun f -> Clos (f, e)) fs @ s, e)

  in
  iter (f, [], Hashtbl.create 100)
```

As we can see, the first block is very similar to what we did with the ZAM: define a stack, the types of the closures and an environment (for efficiency, this time it is implemented as a hash map from variables «EcIdent.t» to formulas).

The new «h_red» function creates an initial state composed by the formula to be reduced, an empty stack and an empty environment, and begins the reduction by evaluating the auxiliar «iter» function in a tail-recursive manner. In this example it is included the evaluation of an application: iterate over the arguments, putting in the stack a new closure for every one of them.

Here we have the new code that performs the full β-reduction:

```
let rec h_red ri env hyps f =
  let iter st =
    match (st : EcFol.form * stack * menv) with

    (* β-red *)
    | ({ f_node = Fapp (f, fs) }, s, e) when ri.beta ->
       iter (f, List.map (fun f -> Clos (f, e)) fs @ s, e)
    | ({ f_node = Fquant (Llambda, [], f) }, s, e) ->
       iter (f, s, e)
    | ({ f_node = Fquant (Llambda, (x,_)::bds, f) }, Clos (cf, _) :: s, e) ->
       let e' = Hashtbl.copy e in
       Hashtbl.add e' x cf;
       iter (f_quant Llambda bds f, s, e')

    (* ... *)
```

The second and third cases handle the evaluation of a λ-abstraction: if it has no arguments, just keep going with the function body; if there are arguments and a function closure is present in the stack, bind the function to the closure in the environment and evaluate the function body with one parameter less. (The «f_quant» and «f_lambda» functions are just helpers to build formulas)

In order to do some of the other reductions, as they have nothing to do with the abstract machine but with global state, we simply have to call standard EasyCrypt's functions. For example, to δ-reduce operators and resolve local variables:

```
let rec h_red ri env hyps f =
  let iter st =
    match (st : EcFol.form * stack * menv) with
    (* ... *)

    | ({ f_node = Fop (p, tys) }, s, e) when ri.delta_p p ->
       iter (reduce_op ri env p tys, s, e)

    | ({ f_node = Flocal x }, s, e) -> let f' = if Hashtbl.mem e x
                                               then Hashtbl.find e x
                                               else reduce_local ri hyps x in
                                       iter (f', s, e)

    (* ... *)
```

Once we are done replacing one by one the standard EasyCrypt operations by transitions in the abstract machine, we can see that it works (the formula being reduced appears in the upper right of the screen):

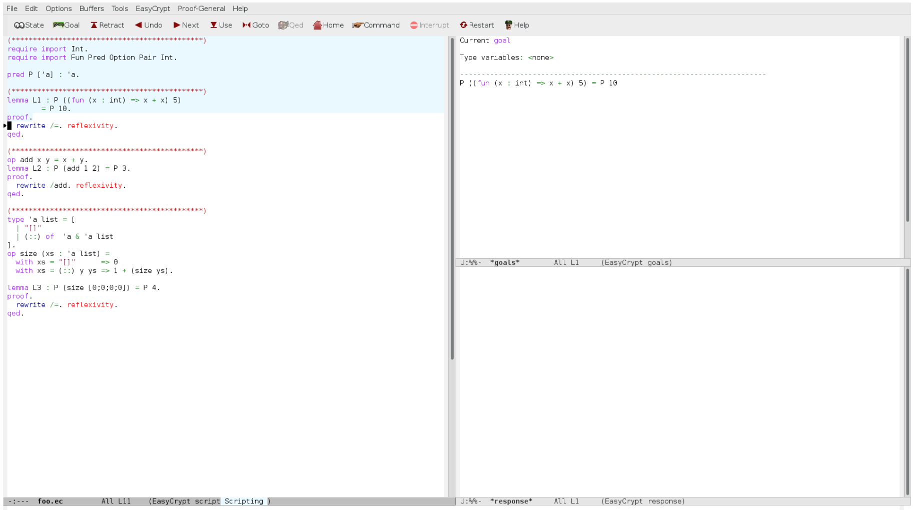

Figure 6.1.: After entering proof mode

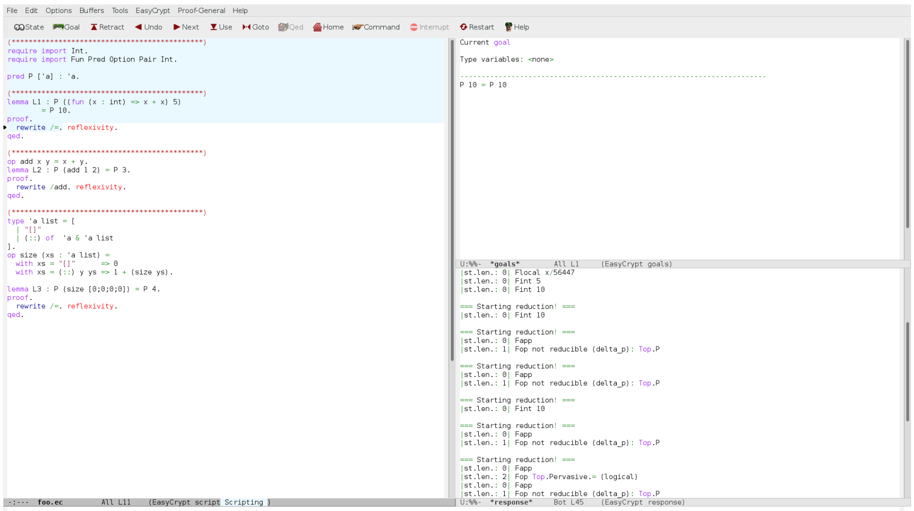

Figure 6.2.: Reduced $((\lambda x.x + x)\ 5) \underset{\beta}{\rightsquigarrow} 10$

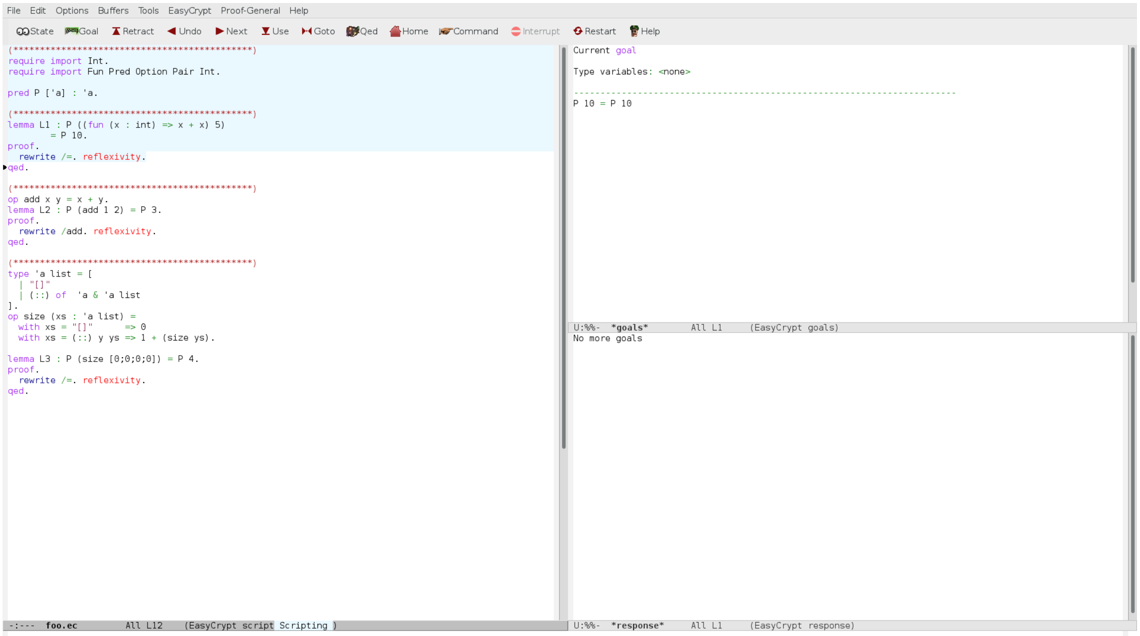

Figure 6.3.: The proof is finished (“no more goals” at bottom right)

Part III.

EPILOGUE

7. CONCLUSIONS

In this work we began by exposing the need to verify cryptographic protocols and the role that the EasyCrypt framework plays in the field. Then we moved on to abstract rewriting systems and how the current machinery that EasyCrypt uses to reduce its formulas could be improved. In order to do that, we implemented two abstract machines with multiple variations (extended lambda terms, strong reduction) and exposed the differences between them: evaluation strategies, symbolic evaluation, bytecode compilation, and so on. Lastly, we continued to the source language and current inner workings of EasyCrypt and proceeded to replace it in a way that closely resembles the work previously done with the abstract machines.

In my opinion, the development of this thesis has resulted in two main contributions belonging to different scopes.

The first one is, obviously, the technical improvement of an existing tool. Although it is not a contribution on features, but the replacing of an existing module for an improved one, we believe it is an important step that had to be taken in order to be able to further expand the system in the future.

The second contribution is the insight given by the actual implementation of two different abstract machines. While none of these by itself was really needed for the task of replacing the EasyCrypt's engine, the research needed to understand the concepts of the abstract machines and correctly implement them has proven crucial when facing a complex system such as EasyCrypt. It might prove to be valuable to some of the interested readers as well, as having the source code of both the abstract machines is a nice way to experiment and compare their behaviors.

Of course, this work can be improved in many ways. The engine is still evaluating code symbolically, which is slower than producing instructions (bytecode) for the machine to evaluate. Some EasyCrypt features make this a feature not trivial to implement (i.e., it would need to decompile bytecode to recover the original terms), but worth keeping it in mind as a possibility for future work.

8. ANNEX

8.1. Krivine Machine source code

```
(*********************************************)
(* (Extended) Krivine Machine implementation *)
(*                                           *)
(* Guillermo Ramos Gutiérrez (2015)          *)
(*********************************************)


(*********)
(* Utils *)
(*********)
let print_with f x = print_endline (f x)
let concat_with sep f xs = String.concat sep (List.map f xs)

let rec find_idx_exn x = function
  | [] -> raise Not_found
  | (y::ys) -> if x = y then 0 else 1 + find_idx_exn x ys

let rec map_rev f xs =
  let rec iter acc = function
    | [] -> acc
    | (x::xs) -> iter (f x :: acc) xs in
  iter [] xs

type symbol = string * int
let show_symbol (s, _) = s
let symbol : string -> symbol =
  let id = ref 0 in
  let gensym c : symbol =
    let newid = !id in
    id := !id + 1;
    (c, newid)
  in
  gensym


(*************************************)
(* (Extended) Untyped Lambda Calculus *)
(*************************************)
module Lambda = struct
    type t = Var of symbol | App of t * t | Abs of symbol * t
             | If of t * t * t | True | False
             | Constr of t constr | Case of t * (symbol constr * t) list
             | Fix of symbol * t
     and 'a constr = symbol * 'a list

    let rec show m =
      let show_branch ((x, args), m) =
        "(" ^ show_symbol x ^ "(" ^ concat_with "," show_symbol args ^ ") → "
        ^ show m ^ ")" in
```

```
  match m with
  | Var x -> show_symbol x
  | App (m1, m2) -> "(" ^ show m1 ^ " " ^ show m2 ^ ")"
  | Abs (x, m) -> "(λ" ^ show_symbol x ^ "." ^ show m ^ ")"
  | If (m1, m2, m3) -> "if " ^ show m1
                      ^ " then " ^ show m2
                      ^ " else " ^ show m3
  | True -> "True" | False -> "False"
  | Constr (x, ms) -> show_symbol x ^ "(" ^ concat_with ", " show ms ^ ")"
  | Case (m, bs) -> "(case " ^ show m ^ " of "
                   ^ concat_with " " show_branch bs ^ ")"
  | Fix (x, m) -> "fix(λ" ^ show_symbol x ^ "." ^ show m ^ ")"
let print = print_with show

(* Constants *)
let none = symbol "None"
let some = symbol "Some"

(* Peano arithmetic helpers *)
let z = symbol "z"
let s = symbol "s"

let rec peano_add n x =
  if n == 0 then x else peano_add (n-1) (Constr (s, [x]))

let peano_of_int ?(base=Constr (z, [])) n = peano_add n base

(* Examples *)

(* (λx.((λy.y) x)) *)
let ex_m1 =
  let x = symbol "x" in
  let y = symbol "y" in
  Abs (x, App (Abs (y, Var y), Var x))

(* (((λx.(λy.(y x))) (λz.z)) (λy.y)) *)
let ex_m2 =
  let x = symbol "x" in
  let y = symbol "y" in
  let z = symbol "z" in
  App (App (Abs (x, Abs (y, App (Var y, Var x))), Abs (z, Var z)), Abs (y, Var y))

(* (λc. case c of (Some(x) → x) (None → c)) Some(s(z))) *)
let ex_case_some =
  let c = symbol "c" in
  let x = symbol "x" in
  App (Abs (c, Case (Var c, [((some, [x]), Var x); ((none, []), Var c)])),
       Constr (some, [peano_of_int 1]))

(* (λc. case c of (Triple(x,y,z) → y)) Triple(1,2,3)) *)
let ex_case_tuple =
  let c = symbol "c" in
  let x = symbol "x" in
  let y = symbol "y" in
  let z = symbol "z" in
  let triple = symbol "triple" in
  App (Abs (c, Case (Var c, [((triple, [x;y;z]), Var y)])),
       Constr (triple, List.map peano_of_int [1;2;3]))

(* fix(λf.λc. case c of (s(x) → s(s(f x))) (z → z)) s(s(s(z))) *)
let ex_fixpt_mul2 =
  let f = symbol "f" in
  let c = symbol "c" in
  let x = symbol "x" in
  App (Fix (f, Abs (c, Case (Var c,
             [((s, [x]), peano_add 2 (App (Var f, Var x)));
              ((z, []), peano_of_int 0)]))),
```

```
           peano_of_int 3)

    (* fix(λf.λg.λc. case c of (s(x) → g (f g x)) (z → z)) (λy.s(s(s(y)))) s(s(s(z))) *)
    let ex_fix_scale =
      let f = symbol "f" in
      let g = symbol "g" in
      let c = symbol "c" in
      let x = symbol "x" in
      let y = symbol "y" in
      App (App (Fix (f, Abs (g, Abs (c, Case (Var c,
                 [((s, [x]), App (Var g, (App (App (Var f, Var g), Var x))));
                  ((z, []), peano_of_int 0)]))))),
                Abs (y, peano_add 3 (Var y))),
           peano_of_int 3)
end


(*************************************************)
(* Untyped lambda calculus with De Bruijn indices *)
(*************************************************)
module DBILambda = struct
    type dbi_symbol = int * symbol
    type t = Var of dbi_symbol | App of t * t | Abs of symbol * t
             | If of t * t * t | True | False
             | Constr of t constr | Case of t * (symbol constr * t) list
             | Fix of symbol * t
     and 'a constr = symbol * 'a list

    let dbi dbis x = (find_idx_exn x dbis, x)

    let output_dbi = false
    let show_dbi_symbol (n, x) =
      if output_dbi then string_of_int n else show_symbol x
    let show_dbi_param x =
      if output_dbi then "" else show_symbol x

    let rec show m =
      match m with
      | Var x -> show_dbi_symbol x
      | App (m1, m2) -> "(" ^ show m1 ^ " " ^ show m2 ^ ")"
      | Abs (x, m) -> "(λ" ^ show_dbi_param x ^ "." ^ show m ^ ")"
      | If (m1, m2, m3) -> "if " ^ show m1
                          ^ " then " ^ show m2
                          ^ " else " ^ show m3
      | True -> "True" | False -> "False"
      | Constr (x, ms) -> show_symbol x ^ "(" ^ concat_with ", " show ms ^ ")"
      | Case (m, bs) -> "(case " ^ show m ^ " of "
                       ^ concat_with " " show_branch bs ^ ")"
      | Fix (x, m) -> "fix(λ" ^ show_symbol x ^ "." ^ show m ^ ")"
    and show_branch ((x, args), m) =
      "(" ^ show_symbol x ^ "(" ^ concat_with "," show_symbol args ^ ") → "
      ^ show m ^ ")"
    let print = print_with show

    let of_lambda =
      let rec of_lambda dbis = function
        | Lambda.Var x -> let (n, x) = dbi dbis x in Var (n, x)
        | Lambda.App (m1, m2) -> App (of_lambda dbis m1, of_lambda dbis m2)
        | Lambda.Abs (x, m) -> Abs (x, of_lambda (x :: dbis) m)
        | Lambda.If (m1, m2, m3) -> If (of_lambda dbis m1,
                                       of_lambda dbis m2, of_lambda dbis m3)
        | Lambda.True -> True | Lambda.False -> False
        | Lambda.Constr (x, ms) -> Constr (x, List.map (of_lambda dbis) ms)
        | Lambda.Case (m, bs) -> Case (of_lambda dbis m,
                                      List.map (trans_br dbis) bs)
        | Lambda.Fix (x, m) -> Fix (x, of_lambda (x :: dbis) m)
      and trans_br dbis ((x, args), m) =
```

```
        let dbis = List.rev args @ dbis in
        ((x, args), of_lambda dbis m) in
      of_lambda []
end


(******************)
(* Krivine Machine *)
(******************)
module KM = struct
    open DBILambda

    type st_elm = Clos of DBILambda.t * stack
                | IfCont of DBILambda.t * DBILambda.t
                | CaseCont of (symbol DBILambda.constr * DBILambda.t) list * stack
                | FixClos of symbol * DBILambda.t * stack
    and stack = st_elm list

    type state = DBILambda.t * stack * stack

    let reduce m =
      let rec reduce (st : state) : DBILambda.t =
        match st with
        (* Pure lambda calculus *)
        | (Var (0, _), s, Clos (m, e') :: e) -> reduce (m, s, e')
        | (Var (0, _), s, FixClos (f, m, e') :: e) ->
           reduce (m, s, FixClos (f, m, e') :: e')
        | (Var (n, x), s, _ :: e) -> reduce (Var (n-1, x), s, e)
        | (App (m1, m2), s, e) -> reduce (m1, Clos (m2, e) :: s, e)
        | (Abs (_, m), c :: s, e) -> reduce (m, s, c :: e)
        (* Conditionals *)
        | (If (m1, m2, m3), s, e) -> reduce (m1, IfCont (m2, m3) :: s, e)
        | (True, IfCont (m2, m3) :: s, e) -> reduce (m2, s, e)
        | (False, IfCont (m2, m3) :: s, e) -> reduce (m3, s, e)
        (* Case expressions (+ constructors) *)
        | (Case (m, bs), s, e) -> reduce (m, CaseCont (bs, e) :: s, e)
        | (Constr (x, ms), CaseCont (((x', args), m) :: bs, e') :: s, e)
             when x == x' && List.length ms == List.length args ->
           reduce (List.fold_left (fun m x -> Abs (x, m)) m args,
                   map_rev (fun m -> Clos (m, e)) ms @ s, e')
        | (Constr (x, ms), CaseCont (_ :: bs, e') :: s, e) ->
           reduce (Constr (x, ms), CaseCont (bs, e') :: s, e)
        | (Constr (x, ms), s, e) ->
           Constr (x, List.map (fun m -> reduce (m, s, e)) ms)
        (* Fixpoints *)
        | (Fix (x, m), s, e) -> reduce (m, s, FixClos (x, m, e) :: e)
        (* Termination checks *)
        | (m, [], []) -> m
        | (_, _, _) -> m in
      reduce (m, [], [])
end


let dbi_and_red m =
  let dbi_m = DBILambda.of_lambda m in
  print_endline ("# Lambda term:\n    " ^ DBILambda.show dbi_m);
  let red_m = KM.reduce dbi_m in
  print_endline ("# Reduced term:\n    " ^ DBILambda.show red_m);
  print_endline "----------------------------------------------------------\n"

let () =
  let open Lambda in
  List.iter dbi_and_red [ex_m1; ex_m2; ex_case_some;
                         ex_case_tuple; ex_fixpt_mul2; ex_fix_scale]
```

8.2. ZAM source code

```
(***********************************)
(* (Extended) ZAM implementation    *)
(*                                  *)
(* Guillermo Ramos Gutiérrez (2015) *)
(***********************************)


(*********)
(* Utils *)
(*********)
let print_with f x = print_endline (f x)
let concat_with sep f xs = String.concat sep (List.map f xs)
let split n xs =
  let rec split_acc n xs ys = match (n, ys) with
    | (0, _) | (_, []) -> (List.rev xs, ys)
    | (n, y :: ys) -> split_acc (n-1) (y::xs) ys in
  split_acc n [] xs
let find_idx a =
  let rec find_acc n = function
    | [] -> raise Not_found
    | (x :: xs) -> if a == x then n else find_acc (n+1) xs in
  find_acc 0
let rec repeat n x = if n == 0 then [] else x :: repeat (n-1) x
let fold_left1 f = function
  | [] -> raise (Invalid_argument "empty string")
  | (x::xs) -> List.fold_left f x xs

type symbol = string * int
let symbol : string -> symbol =
  let id = ref 0 in
  let gensym c : symbol =
    let newid = !id in
    id := !id + 1;
    (c, newid)
  in
  gensym

let dbg_lev = 1
let debug lev spaces s =
  if dbg_lev >= lev
  then print_endline (" -- " ^ String.concat "" (repeat spaces "  ") ^ s)


(* Compile, decompile and reduce errors *)
exception CpErr of string
exception DcErr of string
exception RdErr of string


(***********************************)
(*  Calculus of Constructions terms *)
(***********************************)
module CCLambda = struct
    type t = Var of symbol | App of t * t | Abs of symbol * t
           | Constr of t constr | Case of t * (symbol constr * t) list
           | Fix of symbol * symbol list * symbol * t
           | Acc of t
    and 'a constr = symbol * 'a list

    let show_symbol (s, n) =
      if dbg_lev > 2 then s ^ "/" ^ string_of_int n else s
    let rec show m =
      let show_branch ((x, vs), m) =
```

```
    "(" ^ show_symbol x ^ "(" ^ concat_with "," show_symbol vs ^ ") → "
    ^ show m ^ ")" in
  match m with
  | Var x -> show_symbol x
  | App (m1, m2) -> "(" ^ show m1 ^ " " ^ show m2 ^ ")"
  | Abs (x, m) -> "(λ" ^ show_symbol x ^ "." ^ show m ^ ")"
  | Constr (x, ms) -> show_symbol x ^ "(" ^ concat_with ", " show ms ^ ")"
  | Case (m, bs) -> "(case " ^ show m ^ " of "
                   ^ concat_with " " show_branch bs ^ ")"
  | Fix (f, xs, c, m) ->
     "fix_" ^ string_of_int (List.length xs)
     ^ "(λ" ^ concat_with ".λ" show_symbol (f::xs@[c])
     ^ ". " ^ show m ^ ")"
  | Acc m -> "[" ^ show m ^ "]"
let print = print_with show

(* Auxiliar term-generating functions *)
let identity s =
  let x = symbol s in
  Abs (x, Var x)

let apps = fold_left1 (fun m n -> App (m, n))

let none = symbol "None"
let some = symbol "Some"
let cons = symbol "Cons"
let nil  = symbol "Nil"

(* Peano arithmetic helpers *)
let z = symbol "z"
let s = symbol "s"

let rec peano_add n x =
  if n == 0 then x else peano_add (n-1) (Constr (s, [x]))

let peano_of_int ?(base=Constr (z, [])) n = peano_add n base

(* Examples *)

(* (λx.x) (λy. ((λz.z) y) (λt.t) *)
let ex_m1 =
  let y = symbol "y" in
  App (identity "x",
       Abs (y, App (App (identity "z",
                         Var y),
                    identity "t")))

(* (λf.λx. f (f x)) (λy.y) (λz.z) *)
let ex_id_id =
  let f = symbol "f" in
  let x = symbol "x" in
  App (App (Abs (f, Abs (x, App (Var f, App (Var f, Var x)))),
            identity "y"),
       identity "z")

(* (λc. case c of (Cons(x, xs) → x) (Nil → Nil)) Cons(λx.x, Nil) *)
let ex_case_head =
  let c = symbol "c" in
  let x = symbol "x" in
  let xs = symbol "xs" in
  App (Abs (c, Case (Var c, [((cons, [x;xs]), Var x);
                             ((nil, []), Constr (nil, []))])),
       Constr (cons, [identity "m"; Constr (nil, [])]))

(* fix_0(λf.λc. case c of (s(x) → s(s(f x))) (z → z)) s(s(s(z))) *)
let ex_fixpt_dup =
  let f = symbol "f" in
```

```
      let c = symbol "c" in
      let x = symbol "x" in
      App (Fix (f, [], c, Case (Var c,
                 [((s, [x]), peano_add 2 (App (Var f, Var x)));
                  ((z, []), peano_of_int 0)])),
           peano_of_int 3)
end

(************************************)
(*  (Extended) ZAM - Weak Reduction *)
(************************************)
module WeakRed : sig
    type instrs
    type mval

    val show_instrs : instrs -> string
    val show_mval   : mval   -> string

    val compile   : CCLambda.t -> instrs
    val decompile : instrs -> CCLambda.t

    val am_reduce : instrs -> mval
    val extract   : mval -> CCLambda.t

    val reduce : CCLambda.t -> CCLambda.t
end = struct
    open CCLambda

    type dbi = symbol * int
    type instr =
      | ACCESS of dbi
      | CLOSURE of instrs
      | ACCLOSURE of symbol
      | PUSHRETADDR of instrs
      | APPLY of int
      | GRAB of symbol
      | RETURN
      | MAKEBLOCK of symbol * int
      | SWITCH of (symbol constr * instrs) list
      | CLOSUREREC of (symbol * symbol list * symbol) * instrs
      | GRABREC of symbol
    and instrs = instr list
    and accum =
      | NoVar of symbol
      | NoApp of accum * mval list
      | NoCase of accum * instrs * env
      | NoFix of accum * instrs * env
    and mval =
      | Accu of accum
      | Cons of mval constr
      | Clos of instrs * env
      | Ret of instrs * env * int
    and env = mval list
    and stack = mval list
    and state = {
        st_c : instrs;
        st_e : env;
        st_s : stack;
        st_n : int;
    }

    let show_dbi (x, i) =
      string_of_int i ^
        if dbg_lev > 2 then " (" ^ show_symbol x ^ ")" else ""
    let rec show_instr = function
      | ACCESS dbi -> "ACCESS(" ^ show_dbi dbi ^ ")"
      | CLOSURE is -> "CLOSURE(" ^ show_instrs is ^ ")"
```

```
  | ACCLOSURE x -> "ACCLOSURE([" ^ show_symbol x ^ "])"
  | PUSHRETADDR is -> "PUSHRETADDR(" ^ show_instrs is ^ ")"
  | APPLY i -> "APPLY(" ^ string_of_int i ^ ")"
  | GRAB x -> "GRAB(" ^ show_symbol x ^ ")"
  | RETURN -> "RETURN"
  | MAKEBLOCK (x, n) -> "MAKEBLOCK(#" ^ show_symbol x ^ ", " ^
                         string_of_int n ^ ")"
  | SWITCH bs -> "SWITCH(" ^ concat_with ", " show_branch bs ^ ")"
  | CLOSUREREC (_, is) -> "CLOSUREREC(" ^ show_instrs is ^ ")"
  | GRABREC x -> "GRABREC(" ^ show_symbol x ^ ")"
and show_branch (((c, _), _), is) = c ^ " → " ^ show_instrs is
and show_accum = function
  | NoVar x -> "{NoVar: " ^ show_symbol x ^ "}"
  | NoApp (k, mvs) -> "{NoApp: " ^ show_accum k ^ ", "
                       ^ concat_with ", " show_mval mvs ^ "}"
  | NoCase (k, is, e) -> "{NoCase: " ^ show_accum k ^ ", "
                         ^ show_instrs is ^ ", " ^ show_env e ^ "}"
  | NoFix (k, is, e) -> "{NoFix: " ^ show_accum k ^ ", "
                        ^ show_instrs is ^ ", " ^ show_env e ^ "}"
and show_mval mval = match mval with
  | Accu k -> show_accum k
  | Cons ((s, _), mvs) ->
     "{#" ^ s ^ if List.length mvs == 0 then "}"
                else ": " ^ concat_with ", " show_mval mvs ^ "}"
  | Clos (is, e) -> "{Tλ: (" ^ show_instrs is ^ "), "
                   ^ (if List.length e > 0 && List.hd e == mval
                      then "{Tλ <fix>}::" ^ show_env (List.tl e)
                      else show_env e)
                   ^ "}"
  | Ret (is, e, n) -> "{<RET>: (" ^ show_instrs is ^ "), " ^ show_env e ^ ", "
                     ^ string_of_int n ^ "}"
and show_env e = "[" ^ concat_with ", " show_mval e ^ "]"
and show_instrs is = concat_with "; " show_instr is

let show_stk stk = "| " ^ concat_with "\n     | " show_mval stk
let show_st {st_c; st_e; st_s; st_n} =
  "\n/---------------------------------------------------\n"
  ^ "  C: " ^ show_instrs st_c ^ "\n"
  ^ "  E: " ^ show_env st_e ^ "\n"
  ^ "  S: " ^ show_stk st_s ^ "\n"
  ^ "  N: " ^ string_of_int st_n ^ "\n"
  ^ "\\---------------------------------------------------"

let ret = [RETURN]

let compile (m : CCLambda.t) : instrs =
  let e = [] in
  let rec compile' e (is : instrs) m =
    debug 3 3 ("COMPILING: " ^ show m);
    debug 3 3 ("     IN ENV: " ^ concat_with ", " show_symbol e);
    match m with
    | Var x -> begin
        try ACCESS(x, find_idx x e) :: is
        with Not_found -> raise (CpErr ("Var " ^ fst x ^ " not found"))
      end
    | Abs (x, m) -> CLOSURE(GRAB x :: compile' (x :: e) ret m) :: is
    | App (m1, m2) -> let cont = compile' e [APPLY(1)] m1 in
                     PUSHRETADDR(is) :: compile' e cont m2
    | Constr (x, args) -> let f arg cont = compile' e cont arg in
                          let cont = [MAKEBLOCK (x, List.length args)] in
                          List.fold_right f (List.rev args) (cont @ is)
    | Case (m, bs) ->
       let compile_branch ((c, args), m) =
         let dbi' = List.rev args @ e in
         ((c, args), compile' dbi' ret m) in
       let bs' = List.map compile_branch bs in
       PUSHRETADDR(is) :: compile' e [SWITCH(bs')] m
```

```
    | Fix (f, xs, c, m) ->
       let cont = compile' (c :: List.rev xs @ f :: e) ret m in
       CLOSUREREC((f, xs, c),
                  List.map (fun x -> GRAB x) xs @ GRABREC c :: cont) :: is
    | Acc (Var x) -> ACCLOSURE x :: is
    | Acc _ -> raise (CpErr "Trying to compile non-var accumulator") in
  compile' e ret m

let rec decompile' e s is =
  debug 3 3 ("DECOMPILING: " ^ show_instrs is);
  debug 3 3 ("     IN ENV: " ^ concat_with ", " CCLambda.show e);
  debug 3 3 ("     IN STK: " ^ concat_with ", " CCLambda.show s);
  match is with
  | [] -> List.hd s
  | [RETURN] -> List.hd s
  | (ACCESS (_, i) :: is') -> decompile' e (List.nth e i :: s) is'
  | (CLOSURE c_is :: is') -> decompile' e (decompile' e s c_is :: s) is'
  | (PUSHRETADDR r_is :: is') -> decompile' e (decompile' e s is' :: s) r_is
  | (APPLY i :: is) -> begin
      match (i, s) with
      | (1, a::b::_) -> App (a, b)
      | (n, a::s') -> App (a, decompile' e s' [APPLY (n-1)])
      | _ -> raise (DcErr "Unable to decompile APPLY")
    end
  | (GRAB x :: is') -> Abs (x, decompile' (Var x :: e) s is')
  | (MAKEBLOCK (x, n) :: is') -> let (args, st') = split n s in
                               decompile' e (Constr (x, args) :: s) is'
  | (SWITCH brs :: is') -> begin
      let decompile_br ((c, parms), is) =
        let parms_m = List.map (fun x -> Var x) parms in
        ((c, parms), decompile' (List.rev parms_m @ e) s is) in
      match s with
      | [] -> Case (Var (symbol "_"), List.map decompile_br brs)
      | (a::s') -> Case (a, List.map decompile_br brs)
    end
  | (CLOSUREREC ((f, xs, c), is') :: is) ->
     let m = decompile' (e) s is' in
     decompile' e (Fix (f, xs, c, m) :: s) is
  | (GRABREC x :: is') -> Abs (x, decompile' (Var x :: e) s is')
  | _ -> raise (DcErr "Unable to decompile (unknown instruction)")
let decompile = decompile' [] []

let rec extract mv =
  debug 3 3 ("EXTRACTING: " ^ show_mval mv);
  match mv with
  | Clos (is, _) -> decompile is
  | Cons (x, mvs) -> Constr (x, List.map extract mvs)
  | Accu k ->
     let rec extract_accu = function
       | (NoVar x) -> Acc (Var x)
       | (NoApp (k, mvs)) ->
          Acc (apps (extract_accu k :: List.map extract mvs))
       | (NoCase (k, is, e)) -> begin
           match decompile' (List.map extract e) [] is with
           | Case (m, bs) ->
              let accu = extract_accu k in
              Acc (Case (accu, bs))
           | _ -> raise (DcErr "Invalid decompilation of CASE accum.")
         end
       | (NoFix (k, is, e)) -> begin
           match decompile' (List.map extract e) [] is with
           | Fix (f, xs, c, m) ->
              Acc (Fix (f, xs, c, App (m, extract_accu k)))
           | _ -> raise (DcErr "Invalid decompilation of fixpt accum.")
         end in
     extract_accu k
  | _ -> raise (DcErr "Unable to extract")
```

```
let am_reduce is =
  debug 3 3 ("REDUCING: " ^ show_instrs is);
  let rec eval st =
    debug 4 4 (show_st st);
    let {st_c=c; st_e=e; st_s=s; st_n=n} = st in
    match c with
    | [] -> raise (RdErr "Empty code section")
    | (instr :: c) -> begin
        match instr with
        | ACCESS ((x, _), i) -> begin
            try eval {st with st_c=c; st_s=(List.nth e i :: s)}
            with Not_found -> raise (RdErr ("Var " ^ x ^ " not found"))
          end
        | CLOSURE c' -> eval {st with st_c=c; st_s=(Clos(c', e) :: s)}
        | ACCLOSURE x -> eval {st with st_c=c; st_s=(Accu(NoVar x) :: s)}
        | PUSHRETADDR c' -> eval {st with st_c=c; st_s=(Ret(c', e, n) :: s)}
        | APPLY i -> begin
            match s with
            | (Clos (c', e') :: s) ->
               eval {st_c=c'; st_e=e'; st_s=s; st_n=i}
            | (Accu k :: s) -> begin
                let (args, s') = split i s in
                match s' with
                | (Ret (c', e', n') :: s'') ->
                   eval {st_c=c'; st_e=e';
                         st_s=(Accu(NoApp(k,args)))::s''; st_n=n'}
                | _ -> raise (RdErr "APPLY over accu with invalid stack")
              end
            | _ -> raise (RdErr "APPLY over non-closure mval")
          end
        | GRAB _ -> begin
            if n == 0 then
              match s with
              | (Ret (c', e', n') :: s) ->
                 let clos = Clos (instr :: c, e) in
                 eval {st_c=c'; st_e=e'; st_s=clos::s; st_n=n'}
              | [] -> Clos (instr :: c, e)
              | _ -> raise (RdErr "GRAB (n=0) over non-retval")
            else
              match s with
              | (v :: s) -> eval {st_c=c; st_e=v::e; st_s=s; st_n=n-1}
              | _ -> raise (RdErr "GRAB (n>0) over empty stack")
          end
        | RETURN -> begin
            if n == 0 then
              match s with
              | (v :: Ret (c', e', n') :: s) ->
                 eval {st_c=c'; st_e=e'; st_s=v::s; st_n=n'}
              | [v] -> v
              | _ -> raise (RdErr "RETURN over empty stack or non-retval")
            else
              match s with
              | (Clos (c', e') :: s) ->
                 eval {st_c=c'; st_e=e'; st_s=s; st_n=n}
              | _ -> raise (RdErr "RETURN over empty stack or non-retval")
          end
        | MAKEBLOCK (x, m) -> let (vs, s') = split m s in
                             eval {st_c=c; st_e=e;
                                   st_s=(Cons (x, vs)::s'); st_n=n}
        | SWITCH bs -> begin
            match s with
            | (Cons (x, vs) :: s) ->
               let (_, c') = try List.find (fun ((y, _), _) -> y == x) bs
                             with Not_found ->
                               raise (RdErr "SWITCH constr id not found") in
               let e' = List.rev vs @ e in
```

```
                   eval {st_c=c'; st_e=e'; st_s=s; st_n=0}
                | (Accu k :: s) ->
                   let accu = Accu (NoCase (k, instr :: c, e)) in
                   eval {st_c=ret; st_e=e; st_s=accu::s; st_n=0}
                | _ -> raise (RdErr "SWITCH over empty stack or non-constr")
              end
            | CLOSUREREC (_, c') -> let rec v = Clos (c', v::e) in
                                   eval {st with st_c=c; st_s=v::s}
            | GRABREC _ -> begin
                match (s, n) with
                | ([Accu k], 1) -> Accu (NoFix (k, c, e))
                | (Accu k :: s, n) ->
                   let accu = Accu (NoFix (k, instr :: c, e)) in
                   eval {st_c=ret; st_e=e; st_s=accu::s; st_n=n-1}
                | (Ret (c', e', n') :: s, n) ->
                   let clos = Clos (instr :: c, e) in
                   eval {st_c=c'; st_e=e'; st_s=clos::s; st_n=n'}
                | (mval :: s, n) ->
                   eval {st_c=c; st_e=mval::e; st_s=s; st_n=n-1}
                | ([], 0) -> Clos (instr :: c, e)
                | _ -> raise (RdErr "GRABREC over empty stack or invalid mval")
              end
          end in
      let st = {st_c = is; st_e = []; st_s = []; st_n = 0} in
      eval st

    let reduce m = debug 2 1 ("V( " ^ show m ^ " )");
                   match m with
                   | App (m1, m2) -> m |> compile |> am_reduce |> extract
                   | _ -> m
end

(**************************************)
(*  (Extended) ZAM - Strong Reduction *)
(**************************************)
module StrongRed = struct
    open CCLambda

    let rec extract_unique = function
      | Var x -> x
      | Acc x -> extract_unique x
      | _ -> raise (RdErr "Trying to extract invalid value")

    let unique x = symbol (fst x)

    let rec reduce m =
      debug 2 0 ("N( " ^ show m ^ " )");
      match m with
      | Fix (f, xs, c, m) ->
         let xs' = List.map unique xs in
         let f' = unique f in
         let c' = unique c in
         let accs = List.map (fun x -> Acc (Var x)) (f' :: xs' @ [c']) in
         let m' = reduce (apps (m :: accs)) in
         Fix (f', xs', c', m')
      | _ -> readback (WeakRed.reduce m)
    and readback m =
      debug 2 1 ("R( " ^ show m ^ " )");
      match m with
      | Abs (x, m) -> let u = unique x in
                     Abs (u, reduce (App (Abs (x, m), Acc (Var u))))
      | Constr (x, vs) -> Constr (x, List.map readback vs)
      | Acc k -> readback_acc k
      | App (Fix (f, xs, c, m), p) ->
         App (reduce (Fix (f, xs, c, m)), readback p)
      | _ -> raise (RdErr "Readback of invalid value")
    and readback_acc m =
```

```
      debug 2 2 ("R'( " ^ show m ^ " )");
      match m with
      | Var x -> Var x
      | App (k, v) -> App (readback_acc k, readback v)
      | Acc (Var x) -> Var x
      | Case (k, bs) -> let x = extract_unique k in
                        let b = Abs (x, Case (Var x, bs)) in
                        let rb_branch ((c, xs), m) =
                          let us = List.map unique xs in
                          let acc_us = List.map (fun x -> Acc (Var x)) us in
                          ((c, us), reduce (App (b, Constr (c, acc_us)))) in
                        Case (readback_acc k, List.map rb_branch bs)
      | _ -> raise (RdErr "Readback of invalid accumulator")
end

let weakred m =
  print_endline "WEAK REDUCTION TEST";
  print_endline ("Lambda term:\n    " ^ CCLambda.show m);
  let compiled = WeakRed.compile m in
  let reduced = WeakRed.am_reduce compiled in
  print_endline ("Reduced term:\n    " ^ CCLambda.show (WeakRed.extract reduced));
  print_endline "----------------------------------------------------------\n"

let strongred m =
  print_endline "STRONG REDUCTION TEST";
  print_endline ("Lambda term:\n    " ^ CCLambda.show m);
  print_endline ("Reduced term:\n    " ^ CCLambda.show (StrongRed.reduce m));
  print_endline "----------------------------------------------------------\n"

let () =
  let open CCLambda in
  let examples = [ex_m1; ex_id_id; ex_case_head; ex_fixpt_dup] in
  List.iter weakred examples;
  List.iter strongred examples
```

Bibliography

[1] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *CRYPTO* (P. Rogaway, ed.), vol. 6841 of *Lecture Notes in Computer Science*, pp. 71–90, Springer, 2011. 1.3

[2] G. Barthe, B. Grégoire, and S. Zanella-Béguelin, "Formal certification of code-based cryptographic proofs," in *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pp. 90–101, ACM, 2009. 1.3, 2.1

[3] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," *IACR Cryptology ePrint Archive*, vol. 2004, p. 332, 2004. 2.3

[4] A. Church, "A formulation of a simple theory of types," *Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940. http://www.jstor.org/stable/2266866Electronic Edition. 2.4.1

[5] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Probabilistic relational hoare logics for computer-aided security proofs," in *Mathematics of Program Construction* (J. Gibbons and P. Nogueira, eds.), vol. 7342 of *Lecture Notes in Computer Science*, pp. 1–6, Springer Berlin Heidelberg, 2012. 2.4.1

[6] F. Baader and T. Nipkow, *Term Rewriting and All That.* New York, NY, USA: Cambridge University Press, 1998. 3.1

[7] A. Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, vol. 58, pp. 345–363, April 1936. 3.2

[8] P. Sestoft, "The essence of computation," ch. Demonstrating Lambda Calculus Reduction, pp. 420–435, New York, NY, USA: Springer-Verlag New York, Inc., 2002. 3.4

[9] J.-L. Krivine, "A call-by-name lambda-calculus machine," *Higher-Order and Symbolic Computation*, vol. 20, no. 3, pp. 199–207, 2007. 4

[10] R. Douence and P. Fradet, “The next 700 krivine machines,” *Higher Order Symbol. Comput.*, vol. 20, pp. 237–255, Sept. 2007. 4.2

[11] X. Leroy, “The ZINC experiment: an economical implementation of the ML language,” Technical report 117, INRIA, 1990. 5

[12] B. Grégoire and X. Leroy, “A compiled implementation of strong reduction,” in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP ’02, (New York, NY, USA), pp. 235–246, ACM, 2002. 5, 5.1, 5.1

[13] T. Coquand and G. Huet, “The calculus of constructions,” *Inf. Comput.*, vol. 76, pp. 95–120, Feb. 1988. 5.1