INF565 Project Report
A correct compilation from the λ-calculus to the Krivine machine

Pedro Henrique Azevedo de Amorim   Chaitanya Leena Subramaniam

February 28, 2016

Abstract

This project is an attempt to prove, in Coq, theorems about the correctness of the execution of the Krivine machine. To do this, we implement the λ-calculus and the Krivine machine in Coq, and describe a compilation from the λ-calculus to the language of the Krivine machine. We then prove a correspondence between reduction of λ-terms using β-reduction and the execution of the Krivine machine.

1 Introduction

The Krivine machine [Kri07] is a well known description of a machine that runs programs in the λ-calculus. It uses *weak head reduction*, i.e. the β-reduction only of those reducible expressions (redex) at the beginning of the λ-term, to execute the terms of the λ-calculus. It reduces, using this execution, terms that are reducible to a *weak head normal form* to their weak head normal form. A term is in weak head normal form if it is a λ-abstraction or an expression of the form $\lambda x.E$.

The Krivine machine's execution can be proved to be correct [Wan03] in the following sense: to satisfy the property that if t is a closed term, then the repeated execution of the Krivine machine on t terminates on a state corresponding to u if and only if t reduces to u via β-reduction and u is in weak head normal form.

The Krivine machine is simple enough that it can be practically implemented in the type theory of Coq. This allows, via the Curry-Howard correspondence, for the construction of proofs (terms in the type theory of Coq) corresponding to statements about the correctness of the execution of the Krivine machine.

We try, in this project, to implement the λ-calculus and the Krivine machine in the type theory of Coq, and to prove these theorems. This project is the *mise-en-oeuvre* of the outline described on the webpage of the course INF565 at École Polytechnique[1] (henceforth referred to as the webpage).

The ensemble of the project, and this report can be found at:
`https://github.com/pedrohaa/untypedCoq`

2 The λ-calculus and the Krivine machine

The Krivine machine is well described in many articles (notably [Kri07] and for the sake of brevity, we neither fully describe the semantics and the grammar of our particular implementation, nor the compilation of λ-terms to the language of our Krivine machine, both of which can be found on the webpage.

We implement the λ-calculus using the de Bruijn notation [DB72]. This is crucial to our implementation of Krivine's machine, as it removes the need to rename variables when manipulating λ-terms, and guarantees uniqueness of closed terms.

[1]http://www.enseignement.polytechnique.fr/informatique/INF565/projets/projet-coq.html

3 The implementation in Coq

Implementing the λ-calculus and the Krivine machine in the type theory of Coq thus consists of the steps listed on the webpage. What follows is the outline of our implementation of these steps, i.e. of the definitions and main results.

3.1 The λ-calculus

Our implementation of the λ-calculus in the type theory of Coq is reasonably straightforward.

3.1.1 Terms and substitution

- We first define the type of variables `var` by setting it equal to Coq's type of natural numbers, `nat`. We then inductively define the type `term` of λ-terms. Since `nat` is of type `Set` in Coq, `var` and `term` are too, i.e., `var:Set` and `term:Set`.

  ```
  Definition var := nat.
  Inductive term :=
  | Var : var -> term
  | Lambda: term -> term
  | App: term -> term -> term.
  ```

  Terms in our grammar thus look like the expression `Lambda (App (App (Var 0) (Var 0)) (Var 0))`, that corresponds to the term $\lambda.000$. This seems a little cumbersome at first but makes proofs readable.

- `Fixpoint C: nat -> term -> Prop` is a predicate of closure that, for all terms t and numbers i describes the proposition "the free variables of t are smaller than i". We also prove `C i t -> C (i+1) t`.

- `Fixpoint substitution: nat -> term -> term -> term` is the function of substitution of a free variable of a term by another term, denoted $t[i \leftarrow u]$, where i is the i'th free variable in the environment of the term t. This is not, of course, necessarily the same as the free variable i. For instance, the first free variable in the environment of $\lambda\lambda.1$ is 2, since 0 and 1 are bound by λ-abstractions. We implement this by incrementing i and every free variable of u every time that we pass under a λ-abstraction in t. The latter is done by calling the function `lifting: nat -> nat -> term -> term` on $(1, 0, u)$.

- `Fixpoint multiple_substitution: nat -> term -> list term -> term` substitutes, simultaneously, the free variables from i to $i + \mathit{length}(tl)$ in a term t with the terms in the list tl, denoted $t[i \leftarrow u_0 \dots u_n]$ or $t[i \leftarrow tl]$, with a slight abuse of notation.

- `Lemma mult_sub_inv`. We prove that the order of substitution of the free variables in the $i, \dots, i + \mathit{length}(tl)$ positions in t does not matter if none of the elements of tl contains free variables greater than or equal to i. More precisely, we prove $t[i \leftarrow u :: tl] = t[i+1 \leftarrow tl][i \leftarrow u]$.

3.1.2 β-reduction of terms

- `Inductive reducesInOneTo: term -> term -> Prop` is the predicate that describes the proposition "t reduces to u in one step of β-reduction". This is the implementation of the relation $t \rightarrow u$ of a single step of β-reduction, i.e. the contextual closure over the inductive definition of terms of the relation $(\lambda.t)u \rightarrow t[0 \leftarrow u]$.

- `Inductive reduces: term -> term -> Prop` describes the proposition "t reduces to u using β-reduction". This is defined as the reflexive and transitive closure of the relation $\rightarrow$. It is denoted $\rightarrow^*$.

- `Fixpoint reducesInN: term -> term -> nat -> Prop` describes the proposition "t reduces to u in exactly n steps of $\rightarrow$". We introduced this in order to subsequently try to prove a result of the form "n steps of $\rightarrow$ of t correspond to k steps of the execution of the Krivine machine on $\{compile(t), null, null\}$". This is, however, tricky, since for example, $(\lambda.000)(\lambda.000) \rightarrow (\lambda.000)(\lambda.000)(\lambda.000)$, i.e., in *one* step of β-reduction, while the corresponding number of steps of the Krivine machine to reach a state corresponding to $(\lambda.000)(\lambda.000)(\lambda.000)$ is strictly greater than one.
- We prove that $\rightarrow^*$ satisfies the same properties of contextual closure as $\rightarrow$ by proving that `reducesInN` does so, and then by proving the equivalence
`reduces t u <-> (exists (n: nat), reducesInN t u n).`

This implementation of the λ-calculus with β-reduction allows us to describe a correct compilation from the λ-calculus to the Krivine machine.

3.2 The description of the Krivine machine in the implementation

The Krivine machine is implemented in Coq by defining types corresponding to the instruction, code, environment, stack and state in its description. These are fairly particular to our implementation, thus necessitating a little copying of code from our Coq files.

- The definition of the types of intructions and of code, i.e. the language to which λ-terms are compiled, is given below. It follows directly from the grammar of instructions and of code.

```
Inductive inst :=
| Access: nat -> inst
| Grab: inst
| Push: code -> inst
with code: Type :=
| cnil:   code
| cCons:  inst -> code -> code.
```

- The type of environments is defined as

```
Inductive environment : Type :=
| nul: environment
| cons: code -> environment -> environment -> environment.
```

- We let stacks be of the same type as `environment`. This is justifiable since the type of a stack should correspond to the `list` type of the pair type `code * environment`, and this is the same as `environment` by (un-)currying. This also explains the following definition of states as being expressions of the product type `code * environment * environment`.

```
Definition state := prod (prod code environment) environment.
```

- `exec_inst: state -> state` is the function that describes the semantics of the Krivine machine. It is a straightforward implementation once `state` is defined. There is, however, one subtlety, which has to do with the implementation of the semantics of the machine for those states for which a transition is not defined, e.g., the state $\{Grab; Access\ 0, null, null\}$. What is suggested on the webpage is to implement the transition as `exec_inst: state -> option state` and to have these states evaluate to `None`. We choose, however to have these states evaluate identically to themselves, e.g., `exec_inst (Grab::(Access 0),nul,nul) = (Grab::(Access 0),nul,nul)`. This allows us to more easily prove propositions like "execution of the machine corresponds to $\rightarrow^*$" without treating particular cases (since $\rightarrow^*$ is reflexive, i.e., $t \rightarrow^* t$).
- `Inductive mult_exec_inst: state -> state -> Prop` says that a state can be reduced to another by multiple execution steps.

3.3 Compiling λ-terms into code

3.3.1 Compilation

This involves the simple definition of a compilation function from λ-terms to code.

- `Fixpoint compile: term -> code` compiles λ-terms into code using the same rules described on the webpage applied to the definitions of `term`, `inst` and `code`.

3.3.2 Un-compilation of a state into a λ-term

The definition of this follows exactly from the rules described on the webpage.

- `Fixpoint tau_code: code -> term` is the inverse of `compile`.
- `Fixpoint tau_env: environment -> list term` gives a list of terms corresponding to the values of variables in a term in the environment. We implement the rewriting of a state to a term by substituting the free variables in the term corresponding to the code in the state with the terms in the list corresponding to the environment, i.e., by $\tau(\{c, e, s\}) = (\tau_{code}(c)[0 \leftarrow \tau_{env}(e)])\tau(s)$.
- `Fixpoint tau: state -> term` is defined as described on the webpage. We use `fold_left` to get the correct association of function application.
- `Theorem invert_comp` proves $\tau(\{compile(t), null, null\}) = t$ for all t.

3.4 Proving the correctness of the compilation

As mentioned in the introduction, the correctness of the compilation of λ-terms should be meant in the following sense (cf. [Kri07], [Wan03]): if t is a closed term, then the repeated execution of the Krivine machine on $\{compile(t), null, null\}$ terminates (loops infinitely) on a state corresponding to u if and only if t reduces to u in n steps of $\rightarrow$ (and not $\rightarrow^*$) and u is in weak head normal form.

We did not prove exactly this in our implementation. What we prove is that for a closed term t, for each execution step of the Krivine machine from $\{compile(t), null, null\}$ to s, either t and the term corresponding to s are identical, or the former reduces to the latter in one step of $\rightarrow$. This implies the previous statement (a pen-and-paper proof can be found in [Wan03]) but poses two difficulties as a formal proof in our implementation. Firstly, we have not defined the notion of a term in weak head normal form. Secondly, termination of the machine corresponds, in our implementation, to an infinite loop on a state. The proof using this notion of termination seems tricky in Coq. We are certain that it can be done, but we haven't yet managed to do it.

We also prove (fairly easily) that for a closed term t, any number of execution steps on $\{compile(t), null, null\}$ results in a state corresponding to a term u such that $t \rightarrow^* u$.

The outlines of the important definitions and proofs are given below to help with reading our code.

- `Inductive correct_stk: environment -> Prop` corresponds, for an environment e, to the proposition "all free variables in $\tau_{code}(c_0)$ are smaller than the length of $\tau_{env}(e_0)$ for all pairs (c_0, e_0) in e".
- `correct_state: state -> Prop` is defined as on the webpage.
- `Theorem correct_invariance` proves that a correct state remains correct after one step of execution.
- `Lemma closed_env` proves that a correct environment corresponds, via τ, to a list of closed terms.
- `Lemma fold_reduce` proves $(t \rightarrow u) \Rightarrow (tu_1 \dots u_n \rightarrow uu_1 \dots u_n)$.
- `Lemma closed_implies_correct` proves that for a closed term t, the state $\{compile(t), null, null\}$ is correct.

- `Theorem correct_reduction` proves that for a correct state s, either $\tau(s)$ remains the same after one execution step, or $\tau(s)$ reduces to $exec(s)$ in one step of $\rightarrow$.

  We note that `closed_implies_correct`, `correct reduction`, and `invert_comp` prove easily the first of the two theorems discussed in the previous section.

- `Lemma mult_exec_red` proves that a term corresponding to a correct state s reduces by $\rightarrow^*$ to a term corresponding to multiple execution steps on s.

- `Theorem compiling_correct` is the second theorem discussed in the previous section, and is proved easily from `mult_exec_red`, `closed_implies_correct` and `invert_comp`.

4 Conclusion

We were able to implement the λ-calculus and the Krivine machine in Coq, and prove important theorems about the correctness of the compilation. While we did not manage to prove the theorem mentioned as the precise meaning of the correctness of the compilation, we believe that it can be done in our implementation with certain additional definitions.

A further development to our project would be the introduction of types for terms and the corresponding reduction of typed terms. This would allow us to try to prove results such as strong normalisation of terms in type theories (e.g. the simply-typed λ-calculus). We worked on this project in the hope of extending it to a project we did in the autumn semester of 2015 where we described a type theory that had a model in the effective topos, and which was a slight extension of the simply-typed λ-calculus. This, however, proved to be too ambitious for the time we had, but remains a very interesting potential development of our project.

Throughout the project, we made many decisions about the implementation that may, in retrospect have not been optimal. They have mostly been mentioned in the description of our implementation, and we summarise them here.

- We chose, in the implementation of the Krivine machine, to have the machine enter an infinite loop on a final state rather than to stop with a default value. This makes some of the proofs easier but poses a difficulty when trying to prove the theorem mentioned in the introduction.

- We spent a lot of time proving `closed_env` since to begin with, we had defined the constructor `cons` of `environment` as being of type `(code*environment) -> environment -> environment`. This is the same as our new definition (by uncurrying) but we could not prove the lemma without changing the definition. We still do not know why this is so.

References

[DB72] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[FGSW07] Daniel P Friedman, Abdulaziz Ghuloum, Jeremy G Siek, and Onnie Lynn Winebarger. Improving the lazy krivine machine. *Higher-Order and Symbolic Computation*, 20(3):271–293, 2007.

[Kri07] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[Wan03] Mitchell Wand. On the correctness of the krivine machine. *submitted for publication*, 2003.