Verified Extraction from Coq to a Lambda-Calculus

Yannick Forster, Fabian Kunze

Abstract

We present a framework to export programs in Coq to a weak call-by-value lambda calculus. The calculus can be seen as a very basic functional programming language, featuring abstraction, match-constructs based on Scott's encoding and full recursion.
The extraction from Coq is not verified itself, but produces proofs for the correctness of each single extracted program semi-automatically. The frameworks builds on the Coq plugin Template Coq by Gregory Malecha [3]. It eliminates the non-computational parts of the Gallina program and produces a lambda-term and the corresponding correctness statement, which in turn can be verified using several provided automation tactics.

There are various target languages Gallina code can be extracted to, but we are not aware of a verified extraction method. We present a novel approach: We extract definitions in Coq to a lambda calculus that is formalised in Coq itself and prove them to be correct using strong automation tactics. This approach has several advantages over a global approach where the verification procedure itself is verified. First, we can give the correctness proofs in Coq rather than in a different system that is able to formalize Gallina. Second, there is no need to formalize Gallina, because only a formalisation for the target language is necessary. Third, we can use powerful Coq automation tactics to semi-automatically generate the correctness proofs.

Our target language is a weak call-by-value lambda calculus, formalized with de Bruijn indices. With weak in this context we mean that no reduction under binders is allowed. We use Scott's encoding to implement algebraic datatypes [1], which was used to encode lambda-terms by Mogensen [4].

To extract a program one first has to define the Scott-encodings of the inductive involved types. We do this by defining a typeclass that contains an encoding function, allowing to write `enc a` for the Scott encoding of a term `a` of registered type.

```
Class registered (X : Type) :=mk_registered
{
  enc_f : X → term ; (* the encoding function for X *)
  proc_f : ∀ x, closed_abstraction (enc_f x)
}.
```

The registration of terms has to be done by hand (i.e. one has to define the Scott encoding). We conjecture that this could also be done automatically, again using template-coq.

Our framework takes a Gallina term and generates an inductive representation of the Gallina term using Template Coq [3]. This intermediate representation can be used to execute certain optimisations, for instance proof elimination. We then generate a lambda-calculus term from this intermediate representation and prove its correctness statement.

Correctness statements for an extracted function can in general be generated by recursion over the type of the Gallina function. However, recursion over types is not possible in Coq.

One could generate a correctness statement for every extracted program dynamically. But, to check the correctness of several extracted functions one would have to check every single correctness statement to express the right property. We thus use a HOAS representation of types as follows:

```
Inductive TT : Type → Type :=
  TyB t (H : registered t) : TT t
| TyElim (t:Type) : TT t
| TyAll t (ttt : TT t) (f : t → Type) (ftt : ∀ x, TT (f x))
  : TT (∀  (x:t), f x).
```

We then define a correctness property by recursion over an element `tt` of this type[1]:

The recursive function defining what it means for an extract to be correct reads:

[1] One can use the very same trick for similar things, for instance to define what it means for two functions to be extensionally equal.

```
Definition internalizesF (p : Lambda.term) t (ty : TT t) (f : t) : Prop.
revert p. induction ty as [ t H p | t H p | t ty internalizesHyp R ftt internalizesF']; simpl in *;intros.
- exact (p >* enc f).
- exact (p >* I).
- exact (∀  (y : t) u, closed_abstraction u → internalizesHyp y u → internalizesF' _ (f y) (app p u)).
Defined.
```

For a term of base type `f`, i.e. a registered, Scott encodable type, we require the extract `p` to reduce to the encoding `enc f`. For an eliminated term, i.e. a term of type `Prop`, we require the extract to reduce to the trivial term $I = \lambda x.x$. For a term `f` of function type `∀ x : t, R x`, we require that the extract `p` applied to a correct extract `u` for a type term `y` of type `t` fulfills the correctness condition for type `R x`.

Applied to a lambda-term `a` and the Coq function `add : ℕ→ ℕ→ ℕ` this yields

```
∀ x y u1 u2, (u1 >* enc x) → (u2 >* enc y) → a u1 u2 >* enc (x+y)
```

which is equivalent to

```
∀ x y, a (enc x) (enc y) >* enc (x+y)
```

The extraction function from our intermediate representation to the lambda-calculus is written in Ltac and very straightforward. To reuse previously extracted functions we again use a typeclass:

```
Class internalizedClass (X : Type) (ty : TT X) (x : X):=
{ internalizer : term ;
  proc_t : closed_abstraction internalizer ;
  correct_t : internalizesF internalizer ty x
}.

Definition int (X : Type) (ty : TT X) (x : X) (H : internalizedClass ty x) :=internalizer.
Global Arguments int {X} {ty} x {H} : simpl never.
```

The extraction is – apart from the registration of inductive base types – fully automatic. The verification of an extract is semi-automatic.

We provide strong automation tactics that can solve reduction sequences. A user only has to provide the necessary `induction`- or `destruct`-tactics following the structure of the Gallina program. Our automation is based on reflection and a hand-written, bottom-up, linear rewrite tactic.

The extraction of Coq's plus then only looks as follows[2]:

```
Instance term_add : internalized add.
Proof.
internalizeR. abstract (induction y; recStep P; crush).
Defined.
```

As a case study, we extract several functions that are usually used in computability theory, for instance a self-interpreter for the target lambda-calculus. In fact, the framework to extract programs was first developed to ease the formalisation of computability theory [2], where previously every program had to be defined as a Coq function first, then as a term in the lambda-calculus with a simulation proof. Using the framework, writing terms in the lambda-calculus is not necessary anymore, and even complicated developments can be carried out by writing Gallina only. We conjecture that the same approach works for more interesting target languages and can be used to generate verified developments. For the formalisation of computability theory, our approach is approximately 40% as long (counting specification + proof lines) as the direct approach, where every function has to be written twice.

References

[1] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic: Volume II*. North-Holland Publishing Company, 1972.

[2] Yannick Forster. A formal and constructive theory of computation, Bachelor thesis, 2014. Available electronically at `https://www.ps.uni-saarland.de/~forster/bachelor.php`.

[3] Gregory Malecha. *Template Coq*, 2015. Available electronically at `https://github.com/gmalecha/template-coq/`.

[4] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

[2] the use of abstract is technically not necessary and is only used to speed up further proofs.