Functional programs: conversions between deep and shallow embeddings

Magnus O. Myreen

Computer Laboratory, University of Cambridge, UK

Abstract. This paper presents a method which simplifies verification of deeply embedded functional programs. We present a technique by which proof-certified equations describing the effect of functional programs (shallow embeddings) can be automatically extracted from their operational semantics. Our method can be used in reverse, i.e. from shallow to deep embeddings, and thus for implementing certifying code synthesis: we have implemented a tool which maps HOL functions to equivalent Lisp functions, for which we have a verified Lisp runtime. A key benefit, in both directions, is that the verifier does not need to understand the operational semantics that gives meanings to the deep embeddings.

1 Introduction

For purposes of program verification, programs can be represented in theorem provers either in terms of syntax (a deep embedding), e.g. using an abstract datatype or as a string of ASCII characters

```
(defun APPEND (x y)
  (if (consp x)
      (cons (car x) (APPEND (cdr x) y))
      y))
```

or alternatively, directly as functions in the logic of a theorem prover (shallow embeddings),

$$\begin{array}{lcl} \mathsf{append}\ x\ y & = & \mathsf{if}\ \mathsf{consp}\ x \neq \mathsf{nil}\ \mathsf{then} \\ & & \quad \mathsf{cons}\ (\mathsf{car}\ x)\ (\mathsf{append}\ (\mathsf{cdr}\ x)\ y) \\ & & \mathsf{else}\ y \end{array}$$

Shallow embeddings are easier to work with. Consider e.g. proving associativity of `APPEND`. Proving this over the shallow embedding is straightforward.

$$\mathsf{append}\ x\ (\mathsf{append}\ y\ z) \ = \ \mathsf{append}\ (\mathsf{append}\ x\ y)\ z$$

Proving the same for a deep embedding, w.r.t. an operational semantics $\xrightarrow{\mathsf{ev}}$, involves a tedious proof over a transition system: for all res, env, x, y, z,

$$\begin{array}{l} (\mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [x, \mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [y, z]], env, s) \xrightarrow{\mathsf{ev}} (res, s) \iff \\ (\mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [\mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [x, y], z], env, s) \xrightarrow{\mathsf{ev}} (res, s) \end{array}$$

In some cases, proofs over deep embeddings are unavoidable, e.g. if we are to connect the verification proof to the correctness theorem of a verified compiler or runtime, since these are stated in terms of semantics of deep embeddings.

This paper presents a novel proof-producing technique for converting between the two forms of embedding. Our conversions produce a proof for each run; the result is a *certificate theorem* relating the shallow embedding (append) to the deep embedding (`APPEND`) w.r.t. an operational semantics $\xrightarrow{\mathsf{ap}}$ which specifies how deeply embedded programs evaluate. This will be explained in Section 2.

$$\begin{array}{l}\forall x\ y\ state.\\ \quad \mathsf{code\_for\_append\_in}\ state \implies\\ \quad (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], state) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, state)\end{array}$$

The proof-producing translation technique described in this paper means that the verifier can let automation deal with the operational semantics and thus avoid even understanding the definition of the operational semantics. This work has applications in verification and code synthesis.

Program verification: Given a functional programs written in a deep embedding, e.g. ASCII, we can parse this into an abstract syntax tree and use the translation from deep to shallow to simplify verification. We have used this technique to significantly simplify the task of verifying a 2,000-line Lisp program, the Milawa theorem prover [5], w.r.t a semantics of Lisp [6].

Program synthesis. The ability to translate shallow embeddings into certified deep embeddings can be used to implement high-assurance code synthesis. We have implemented such proof-producing code synthesis from HOL4 into our previously verified Lisp runtime [6]. This improves on the trustworthiness of current *program extraction* mechanisms in HOL4, Isabelle/HOL and Coq which merely print functions into the syntax of SML, Ocaml, Haskell or Lisp without any assurance proof or connection to a semantics of the target language.

2 From deep to shallow embeddings

For purposes of brevity and clarity we will base our examples in this paper on the following abstract syntax for Lisp programs. This is a subset of the input language of our verified Lisp runtime [6].

$$\begin{array}{lcl} term & ::= & \mathsf{Const}\ sexp \mid \mathsf{Var}\ string\\ & & \mid \mathsf{If}\ term\ term\ term \mid \mathsf{App}\ func\ (term\ \mathsf{list}) \mid \ldots\\ func & ::= & \mathsf{PrimitiveFun}\ prim \mid \mathsf{Fun}\ string \mid \mathsf{Funcall} \mid \mathsf{Define} \mid \ldots\\ prim & ::= & \mathsf{Cons} \mid \mathsf{Car} \mid \mathsf{Cdr} \mid \mathsf{Add} \mid \ldots\\ sexp & ::= & \mathsf{Val}\ nat \mid \mathsf{Sym}\ string \mid \mathsf{Dot}\ sexp\ sexp \end{array}$$

We define the operational semantics for this language using inductively defined relations: apply $\xrightarrow{\mathsf{ap}}$ and eval $\xrightarrow{\mathsf{ev}}$. Term *exp* evaluates, in environment *env*, to x (of type *sexp*) if $(exp, env, state) \xrightarrow{\mathsf{ev}} (x, new\_state)$; and the application of function f to arguments xs evaluates to x if $(f, xs, state) \xrightarrow{\mathsf{ap}} (x, new\_state)$. Certain functions, e.g. Define, Print and Error, alter *state*.

2.1 Method

When converting deep embeddings into shallow embeddings our task is to derive a definition of a function append and prove a connection between them, e.g.

$$(\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], state) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, state)$$

The method by which we accomplish this has two phases. The first phase derives a theorem of the following form, for some *hypothesis* and *expression*.

$$hypothesis \implies (body, env, state) \xrightarrow{\mathsf{ev}} (expression, state)$$

This derivation proceeds as a bottom-up traversal of the abstract syntax tree for the *body* of the function we are extracting. At each stage a lemma is applied to introduce the relevant syntax in *body* and, at the same time, construct the corresponding shallowly embedded operations in *expression*.

The second phase defines a shallow embedding using *expression* as the right-hand side of the definition and discharges (most of) the *hypothesis* using the induction that arises from the termination proof for the shallow embedding.

There is no guess work or heuristics involved in this algorithm, which means that well-written implementations can be robust.

2.2 Example: append function

An example will illustrate this algorithm. Consider `APPEND` from above. For the *first phase*, we aim to derive a theorem describing the effect of evaluating the body of the `APPEND` function, i.e.

$$\begin{array}{l}\mathsf{If}\ (\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Consp})\ [\mathsf{Var}\ \texttt{"X"}]) \\ \quad (\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Cons})\ [\ldots, \mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [\ldots]]) \\ \quad (\mathsf{Var}\ \texttt{"Y"})\end{array} \tag{1}$$

Our bottom-up traversal starts at the leaves. Here we have variable look-ups and thus instantiate v to `"X"` and `"Y"` in the following lemma to get theorems describing the leaves of the program.

$$v \in \mathsf{domain}\ env \implies (\mathsf{Var}\ v, env, state) \xrightarrow{\mathsf{ev}} (env\ v, state)$$

Now that we have theorems describing the leaves, we can move upwards and instantiate lemmas for primitives, e.g. for Cdr using modus ponens against:

$$\begin{array}{l}(hyp \implies (x, env, state) \xrightarrow{\mathsf{ev}} (exp, state)) \implies \\ (hyp \implies (\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Cdr})\ [x], env, state) \xrightarrow{\mathsf{ev}} (\mathsf{cdr}\ exp, state))\end{array}$$

When we encounter the recursive call to `APPEND` we, of course, do not have a description yet. In this case, we insert a theorem where *hypothesis* makes the assumption that some function variable *append* describes this application.

$$\begin{array}{l}(\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], state) \xrightarrow{\mathsf{ap}} (append\ x\ y, state) \implies \\ (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], state) \xrightarrow{\mathsf{ap}} (append\ x\ y, state)\end{array}$$

The result of the *first phase* is a theorem of the form

$$\mathit{hypothesis} \implies (\mathit{body}, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathit{expression}, \mathit{state})$$

Here *body* is the abstract syntax tree for the body of `APPEND`; and *expression* is the following, if $\mathit{env} = \{\texttt{"X"} \mapsto x, \texttt{"Y"} \mapsto y\}$,

$$\begin{array}{l}\mathsf{if\ consp}\ x \neq \mathsf{nil}\ \mathsf{then}\\ \quad \mathsf{cons}\ (\mathsf{car}\ x)\ (\mathit{append}\ (\mathsf{cdr}\ x)\ y)\\ \mathsf{else}\ y\end{array} \tag{2}$$

and, with the same *env* instantiation, *hypothesis* is:

$$\begin{array}{l}\mathsf{consp}\ x \neq \mathsf{nil} \implies\\ (\mathsf{Fun}\ \texttt{"APPEND"}, [\mathsf{cdr}\ x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathit{append}\ (\mathsf{cdr}\ x)\ y, \mathit{state})\end{array}$$

Next we enter the *second phase*: we define append so that its right-hand side is (2) with *append* replaced by append. As part of the straightforward termination proof for this definition, we get an induction principle

$$\begin{array}{l}\forall P.\\ \quad (\forall x\ y.\ (\mathsf{consp}\ x \neq \mathsf{nil} \implies P\ (\mathsf{cdr}\ x)\ y) \implies P\ x\ y) \implies\\ \quad (\forall x\ y.\ P\ x\ y)\end{array} \tag{3}$$

which we will use to finalise the proof of the certificate theorem as follows.

For the running example, let P abbreviate the following.

$$\lambda x\ y.\ (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, \mathit{state})$$

We now restate the result of phase one using P and the definition of append:

$$\begin{array}{l}\forall x\ y.\ (\mathsf{consp}\ x \neq \mathsf{nil} \implies \mathsf{P}\ (\mathsf{cdr}\ x)\ y) \implies\\ \qquad (\mathit{body}, \{\texttt{"X"} \mapsto x, \texttt{"Y"} \mapsto y\}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathsf{append}\ x\ y, \mathit{state})\end{array} \tag{4}$$

Let code_for_append_in *state* state that the deep embedding (1) is bound to the name `APPEND` and parameter list `["X", "Y"]` in *state*. Now the operational semantics' rule for function application (Sec. 4.2 of [6]) gives us the following lemma.

$$\begin{array}{l}\forall x\ y.\ (\mathit{body}, \{\texttt{"X"} \mapsto x, \texttt{"Y"} \mapsto y\}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathsf{append}\ x\ y, \mathit{state}) \wedge\\ \qquad \mathsf{code\_for\_append\_in}\ \mathit{state} \implies \mathsf{P}\ x\ y\end{array} \tag{5}$$

By combining (4) and (5) we can prove:

$$\begin{array}{l}\forall x\ y.\ \mathsf{code\_for\_append\_in}\ \mathit{state} \implies\\ \qquad (\mathsf{consp}\ x \neq \mathsf{nil} \implies \mathsf{P}\ (\mathsf{cdr}\ x)\ y) \implies \mathsf{P}\ x\ y\end{array} \tag{6}$$

And a combination of (3) and (6) gives us:

$$\forall x\ y.\ \mathsf{code\_for\_append\_in}\ \mathit{state} \implies \mathsf{P}\ x\ y \tag{7}$$

An expansion of the abbreviation P shows that (7) is the certificate theorem we were to derive for `APPEND`: it states that the shallow embedding append is an accurate description of the deep embedding `APPEND`.

$$\begin{array}{l}\forall x\ y\ \mathit{state}.\\ \quad \mathsf{code\_for\_append\_in}\ \mathit{state} \implies\\ \quad (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, \mathit{state})\end{array}$$

2.3 Example: reverse function

Now consider an implementation for `REVERSE` which calls `APPEND`. In the first phase of the translation, the certificate theorem for `APPEND` (from above) can be used to give a behaviour to Fun `"APPEND"`. The second phase follows the above proof very closely. The result is the following shallow embedding,

$$\begin{array}{lcl} \mathsf{reverse}\ x & = & \mathsf{if\ consp}\ x \neq \mathsf{nil\ then} \\ & & \quad \mathsf{append}\ (\mathsf{reverse}\ (\mathsf{cdr}\ x))\ (\mathsf{cons}\ (\mathsf{car}\ x)\ \mathsf{nil}) \\ & & \mathsf{else\ nil} \end{array}$$

and a similar certificate theorem:

$$\begin{array}{l} \forall x\ \mathit{state}. \\ \quad \mathsf{code\_for\_reverse\_in}\ \mathit{state} \implies \\ \quad (\mathsf{Fun}\ \texttt{"REVERSE"}, [x], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{reverse}\ x, \mathit{state}) \end{array}$$

Here $\mathsf{code\_for\_reverse\_in}\ \mathit{state}$ also requires that code for `APPEND` is present.

2.4 More advanced language features

The most advanced feature our Lisp language supports is dynamic function calls using Funcall: the name of the function to be called is the first argument to Funcall. The equivalent in ML is a call to a function variable. The difference is that Funcall is potentially unsafe, e.g. if called with an invalid function name or with the wrong number of arguments. (ML's type system prevents such unsafe behaviour in ML.) We can support Funcall as follows. First two definitions:

$$\begin{array}{rl} \mathsf{funcall\_ok}\ \mathit{args}\ \mathit{state} = & \exists v.\ (\mathsf{Funcall}, \mathit{args}, \mathit{state}) \xrightarrow{\mathsf{ap}} (v, \mathit{state}) \\ \mathsf{funcall}\ \mathit{args}\ \mathit{state} = & \varepsilon v.\ (\mathsf{Funcall}, \mathit{args}, \mathit{state}) \xrightarrow{\mathsf{ap}} (v, \mathit{state}) \end{array}$$

We use the following lemma in the first phase of the translation algorithm whenever Funcall is encountered.

$$\mathsf{funcall\_ok}\ \mathit{args}\ \mathit{state} \implies (\mathsf{Funcall}, \mathit{args}, \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{funcall}\ \mathit{args}\ \mathit{state}, \mathit{state})$$

The result from phase two is a certificate theorem containing a side-condition which collects the hypothesis that the induction is unable to discharge, e.g. if we were translating a function `CALLF` that uses Funcall then we get:

$$\begin{array}{l} \forall x\ \mathit{state}. \\ \quad \mathsf{code\_for\_callf\_in}\ \mathit{state} \wedge \mathsf{callf\_side}\ x\ \mathit{state} \implies \\ \quad (\mathsf{Fun}\ \texttt{"CALLF"}, [x], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{callf}\ x\ \mathit{state}, \mathit{state}) \end{array}$$

So far we have only considered pure functions, i.e. functions that don't alter *state*. Impure functions are also supported: they translate into shallow embeddings that take the *state* as input and produce a result pair: the return value and the new state, e.g. $(\mathsf{Fun}\ \texttt{"IMPURE\_FUN"}, [x], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{impure\_fun}\ x\ \mathit{state})$.

3 From shallow to deep embeddings

The description above explains how proof-producing translations from deep to shallow embeddings can be performed. The same algorithm can be used for translations in the opposite direction: start by inventing a deep embedding corresponding to the given shallow embedding and, at phase two, refrain from inventing a shallow embedding, instead use the given shallow embedding and its induction principle.

4 Summary and related work

This paper has presented a proof-producing algorithm for translating between shallow and deep embeddings of untyped first-order Lisp programs.

Trustworthy program synthesis is one application area of this work. Li et al. [4] have worked on compiling shallowly embedded functions into assembly code directly from HOL. In this paper we instead establish a connection between HOL and a high-level language (which has a verified runtime). Work by Hardin et al. [2] on decompiling Guardol programs has similar goals.

Program verification is another application area of this work. In this area, Charguéraud [1] has proposed a completely different way of verifying deep embeddings of functional programs. Charguéraud proposes that reasoning is to be carried out using *characteristic formulae* for functional programs. These formulas provide a way of unrolling the operational semantics without dealing with the operational semantics directly. His approach does not require functions to be total, unlike our approach. However, his technique provides relations, while our approach produces equations which fit better with powerful rewriting tactics.

The algorithm presented here bears some resemblance to work by Krauss et al. [3] on constructing termination proofs from termination of rewriting systems.

Ack. I thank Mike Gordon for commenting on drafts and EPSRC for funding.

References

1. Arthur Charguéraud. Program verification through characteristic formulae. In *International Conference on Functional Programming (ICFP)*. ACM, 2010.
2. David Hardin, Konrad Slind, Michael W. Whalen, and Tuan-Hung Pham. The Guardol language and verification system. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS. Springer, 2012.
3. Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle functions via termination of rewriting. In *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2011.
4. Guodong Li, Scott Owens, and Konrad Slind. A proof-producing software compiler for a subset of higher order logic. In *European Symposium on Programming (ESOP)*, LNCS. Springer, 2007.
5. Magnus O. Myreen and Jared Davis. `http://www.cl.cam.ac.uk/~mom22/jitawa/`.
6. Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2011.