Automatically Introducing Tail Recursion in CakeML (Extended Abstract)

Oskar Abrahamsson

Chalmers University of Technology, Gothenburg, Sweden
aboskar@chalmers.se

Abstract. In this paper, we implement an optimizing compiler transformation which turns non–tail-recursive functions into tail-recursive functions in an intermediate language of the CakeML compiler. The implementation is integrated with the existing structure of the compiler as a standalone compiler stage, and is verified to preserve the observational semantics of any transformed program. Moreover, our efforts uncover surprising drawbacks in some of the verification techniques currently employed in several parts of the CakeML compiler. We analyze these drawbacks and discuss potential remedies.

Paper Category: Project.

1 Introduction

Consider the following definition of a function `length` in an ML-like language:

```
fun length []      = 0
  | length (x::xs) = length xs + 1
```

Regardless of what we choose as its name, the purpose of `length` should be immediately clear even to a novice functional programmer – it computes the length of a list. However, this aesthetically pleasing style of programming comes at a price: `length` is not *tail recursive.* Since tail-recursive functions will in general compile to more space efficient and faster code, we give an equivalent tail-recursive definition of `length`:

```
fun length' []      acc = acc
  | length' (x::xs) acc = length' xs (1 + acc)

fun length xs = length' xs 0
```

Functions written using tail calls enable compilers to perform a powerful optimization called tail call elimination. In short, tail call elimination entails the procedure of transforming tail-recursive functions into something which resembles a while loop. Since such a function has a recursive call to itself directly in tail position (i.e. the 'last' position visited when evaluating an expression), no

additional bookkeeping is required to store a return address for the recursive call – once the base case is reached, `length'` may simply return to the function which originally called it. Moreover, when a function performs a tail-call to itself, the locations in memory or registers in which the function arguments are stored can be reused for subsequent recursive calls. The benefits of the procedure are constant stack space usage as well as increased performance due to the reduced amount of bookkeeping.

1.1 Contributions

In this paper, we describe a verified implementation of a code transformation for functional programs, which turns non-tail calls into tail calls by automatically introducing accumulator arguments. Although the technique is well known, it is mostly performed manually by the programmer at the source level. Our implementation of the transformation acts on an intermediate language in the fully verified CakeML compiler.

Section 2 gives a brief introduction to the CakeML project, and the intermediate language BVI on which our transformation acts. The details of the transformation are introduced in Section 3 by means of a worked example in an ML-like language. Moreover, a general description of the transformation is outlined, and we sketch out the details of how the transformation has been implemented for BVI.

Our contributions consist of extending the CakeML compiler with a self-contained phase performing the transformation, as well as a machine-checked proof of semantic preservation. Section 4 gives an overview of the techniques used to verify the transformation on BVI. The style of verification employed in the functional intermediate languages of the CakeML compiler has so far proven successful. In particular, it has enabled the verification of several intricate optimizations that manages to put CakeML in league with OCaml and Poly/ML in certain benchmarks [6]. However, the verification of the transformation presented in this paper reveals some surprising shortcomings to this approach. These shortcomings are discussed in Section 4.2.

Finally, Section 5 puts our work in context with other work done on equivalent or similar transformations. Formal treatments of the transformation described in this paper are sparsely accounted for in literature. In particular, most systematic descriptions focus solely on the removal of list-append, with the introduction of tail-recursion as an implicit side-effect. Additionally, to the best of our knowledge, ours is the first proven-correct implementation, existing in a fully verified compiler.

2 CakeML

CakeML [4] is a strongly typed functional programming language with call-by-value semantics, based on Standard ML. It supports a large subset of the features present in Standard ML, including references, exceptions, modules and I/O.

The CakeML compiler targets several common hardware architectures, including Intel x86, ARM and MIPS. The compiler is implemented in higher-order logic using the HOL4 proof assistant, and comes with a mechanically verified proof of correctness which guarantees that every valid CakeML source program is compiled into semantically compatible machine code.

CakeML recently received a new backend [7] which makes use of 12 intermediate languages (ILs) during compilation. The IL under consideration for our implementation is BVI (Bytecode-Value Intermediate language). BVI is a low-level first-order functional language. Like all other ILs in the new CakeML compiler backend, its formal semantics is specified in terms of a functional big-step style [5]. An overview of the semantics of BVI is given in Section 4.

3 Transforming Recursive Functions in BVI

In this section, we describe a code transformation for automatically introducing tail recursion in the BVI IL. We start by providing an informal description of the procedure through a worked example in Section 3.1. The example is followed by a more general description of the steps of the transformation in Section 3.2. Finally, in Section 3.3 we sketch out the details of an implementation of the transformation for BVI.

3.1 Example

Consider the following naive implementation of a function which reverses a list:

```
fun reverse [] = []                          (* reverse.base *)
  | reverse (x::xs) = reverse xs ++ [x]      (* reverse.rec  *)
```

The tail position in the recursive case of `reverse` contains a list append operation `reverse xs ++ [x]`. We will introduce a function `reverse'` such that for all `xs` and for all `a`, it holds that `reverse' xs a = reverse xs ++ a`. We proceed by specifying the recursive case:

```
fun reverse' (x::xs) a = reverse (x::xs) ++ a
```

Next, we substitute the definition of `reverse.rec` for the call on the right-hand side:

```
fun reverse' (x::xs) a = (reverse xs ++ [x]) ++ a
```

We then utilize the associative property of `(++)`, yielding

```
fun reverse' (x::xs) a = reverse xs ++ ([x] ++ a)
```

Since the property `reverse' xs a = reverse xs ++ a` holds for all choices of `a`, we substitute `reverse' xs []` for `reverse xs` by an inductive argument.

```
fun reverse' (x::xs) a = reverse' xs [] ++ ([x] ++ a)
```

We re-use the inductive argument a second time, this time with `[x] ++ a` for `a`.

```
fun reverse' (x::xs) a = reverse' xs (([x] ++ a) ++ [])
```

The same procedure is applied for the base case of `reverse`. Finally, we give the definition some touch-ups utilizing the definition of `(++)`, and introduce an auxiliary function named so that `reverse'` may be used in place of the original `reverse`:

```
fun reverse' []      a = a
  | reverse' (x::xs) a = reverse' xs (x::a)

fun reverse xs = reverse' xs []
```

3.2 Tail Recursion Using Accumulators

The transformation steps applied in Section 3.1 can be generalized to work with any operation in tail position, so long as it is associative and has an identity element. Let $\oplus$ be an associative operator with identity 0, and let f be some recursive function. The key takeaway from the `reverse` example is that whenever f has an operation

$$f\ x \oplus a \tag{1}$$

in tail position, we can replace this operation by a tail call, by introducing a function f' satisfying

$$f'\ x\ a = f\ x \oplus a\ . \tag{2}$$

The additional argument a to f' is commonly referred to as an *accumulator*, since it accumulates the partial sum of the result computed during the recursion. The production of such a function f' can be performed as follows, by rewriting the existing expression constituting the body of f:

1. For those expressions e in tail position that satisfy the form $e := fx \oplus y$ for some x, y, replace e by $f'\ x\ (y \oplus a)$, where f' is an unused function name.
2. For all other expressions e in tail position, replace them with the expression $e' := e \oplus a$.
3. Finally, rename f to f', and give it an additional argument pointed to by a. The name f is re-used for an auxiliary definition applying f' to the identity of $\oplus$ by setting $f\ x = f'\ x\ 0$.

3.3 Tail Recursion in BVI

Our transformation is to be applied on BVI programs as a standalone stage in the CakeML compiler. At this stage of compilation, the input program has been divided into a list of functions stored in an immutable code store, which we call the *code table*. A code table entry is given by a tuple (nm,ar,exp), where $(nm : \texttt{num})$ is a unique address used for indexing into the table (i.e. the function 'name'), $(ar : \texttt{num})$ defines the arity of the function, and exp the expression which constitutes its body. Our reasons for choosing BVI for this optimization are the following:

- BVI does not support closures. Determining equivalence between values in a language with closures is complicated, since values contain expressions that would be changed by our transformation. Implementing the transformation in a first-order language greatly simplifies verification, as it enables us to use equality as equivalence between values before and after the transformation.
- The compiler stage which transforms a prior higher-level IL into BVI introduces new functions into the compiler code table, and keeps track of what function names are unused. This suits our purposes, since our transformation needs to introduce auxiliary definitions, i.e. using previously unused function names.

We will now give an outline of how the transformation from Section 3.2 is implemented in the BVI stage of the CakeML compiler. The transformation is restricted to expressions containing associative integer arithmetic and list append in tail position, for the reason that these can be easily detected at compile-time.

1. We search the code table for entries (nm,ar,exp), in which exp contains at least one tail position in the shape of $f\ x \oplus y$, where f is the name of the function at address nm.
2. If the previous check succeeds, we create an expression exp_{OPT} by modifying the tail positions of exp:
   - Any expression $f\ x \oplus y$ is replaced by a function call $f'\ x\ (y \oplus a)$, where f' is a function at the next 'free' address nm' in the code table, and a is a variable pointing at a newly allocated argument of the function.
   - Any other expression y in tail position is replaced by $y \oplus a$.
3. Finally, the transformed expression exp_{OPT} is inserted into the code table as $(nm', ar + 1, exp_{\mathrm{OPT}})$, and an auxiliary expression exp_{AUX} is inserted as $(nm, ar, exp_{\mathrm{AUX}})$. This expression simply calls f' while appending the identity of $\oplus$ to the arguments it was called with.

Lastly, we impose some additional restrictions on which expressions that can be transformed. BVI supports a wide range of operations that read or alter global state. In order for effects to not be evaluated out of order, we require that y in the expression $f\ x \oplus y$ does not access global state in any way.

4 Verification of Semantic Preservation

Like most intermediate languages in the CakeML compiler, the semantics of BVI is defined in a functional big-step style using an interpreter function [5]. This enables us to prove theorems regarding program semantics by induction on the recursive cases of the interpreter function. In general, these correctness theorems state that the semantics of a list of expressions is preserved under some transformation applied directly on these expressions. However, since our transformation works on the entire BVI code table – as opposed to lists of expressions – it does not fit well into any existing stages of in the BVI phase of the compiler. We have thus implemented it as a stand-alone stage. In addition to providing theorems which state that observational semantics are preserved when transforming stand-alone expressions, this also requires us to provide a higher-level semantics theorem, stating that the semantics of all expressions in the code table are preserved under the transformation. In the remainder of this section, we give some details of this theorem and sketch out the process of proving it.

4.1 Semantics Preservation

We state and prove the following semantics-preservation theorem for the compiler stage which performs the transformation.

$$
\begin{array}{l}
\vdash \mathsf{every}\ (\mathsf{free\_names}\ n \circ \mathsf{fst})\ \mathit{prog} \wedge \mathsf{all\_distinct}\ (\mathsf{map}\ \mathsf{fst}\ \mathit{prog}) \wedge \\
\quad \mathsf{compile}\ n\ \mathit{prog} = \mathit{prog}_2 \wedge \\
\quad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog})\ \mathit{start} \neq \mathsf{Fail} \Rightarrow \\
\qquad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog})\ \mathit{start} = \\
\qquad\quad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog}_2)\ \mathit{start}
\end{array}
$$

Concretely, the semantics function describes the observable results of evaluating the program *prog* from an entry point *start* (i.e. an address in the code table), and an FFI state *ffi*. An incorrect program, that is, a program which fails to type check, is represented by the result Fail. Our theorem thus states that the semantics of any non-Fail program should be preserved under the transformation compile, which applies the transformation to the functions in the code table.

4.2 Semantics Preservation, cont'd

The semantics theorem is proven by means of the following lemma which guarantees that the semantics of every list of non-failing BVI expressions is preserved when the transformation is applied to the code table. The semantics function is defined in terms of evaluate (see Figure 1), as described in Tan et al. [7].

```
evaluate ([],env,s) = (Rval [],s)

evaluate (x::y::xs,env,s) =
  case evaluate ([x],env,s) of
    (Rval v1,s1) ⇒
      (case evaluate (y::xs,env,s1) of
         (Rval vs,s2) ⇒ (Rval (hd v1::vs),s2)
       | (Rerr v8,s2) ⇒ (Rerr v8,s2))
  | (Rerr v8,s1) ⇒ (Rerr v8,s1)
...

evaluate ([Op op xs],env,s) =
  case evaluate (xs,env,s) of
    (Rval vs,s') ⇒
      (case do_app op (reverse vs) s' of
         Rval (v,s') ⇒ (Rval [v],s')
       | Rerr e ⇒ (Rerr e,s'))
  | (Rerr v7,s') ⇒ (Rerr v7,s')
```

Fig. 1. The definition of the function evaluate, defining semantics of the BVI language. The majority of cases have been left out for brevity, and are replaced with dots.

```
⊢ evaluate (xs,env1,s) = (r,t) ∧
  env_rel transformed acc env1 env2 ∧ code_rel s.code c ∧
  (transformed ⇒ length xs = 1) ∧
  r ≠ Rerr (Rabort Rtype_error) ⇒
    evaluate (xs,env2,s with code := c) =
      (r,t with code := c) ∧
    (transformed ⇒
      ∀ op n exp ar.
        lookup nm s.code = Some (ar,exp) ∧
        is_transformed_code nm ar exp n c op ∧
        tail_is_ok nm (hd xs) = Some op ⇒
          evaluate
            ([transform_tail n op nm acc (hd xs)],env2,
             s with code := c) =
            evaluate
              ([apply_op op (hd xs) (Var acc)],
               env2,s with code := c))
```

The first six lines of the lemma above are quite common for any verification of an optimization in the CakeML compiler. These lines ensure that the expressions xs are well-typed, and that the code table s.code of the state s is related to some code table c under the relation code_rel. This relation simply states that for any entry in s.code there exist corresponding entries in c with correct arities.

Additionally, we require a relation env_rel between the environments in which the original and transformed expressions are evaluated, such that a location in the latter contains a well-typed value that may be used by the accumulator variable of a potentially transformed expression in xs.

What makes this theorem unusual is the implication *transformed* $\Rightarrow \ldots$, which declares the behavior of transformed expressions. In short, we ensure that any such expressions that are present in the code table s.code are transformed in the code table c. Moreover, we require that any expression that passes a static check (see Section 3.3) will, when transformed, evaluate to a result that is equal to the result of simply applying the expression to the accumulator variable under the operation.

The lemma is proven by recursive induction on the semantics function evaluate (see Figure 1). This evaluate function is defined on sequences of lists, and successful evaluation of one such list results in Rval vs, where vs is a list of values. If an error e is encountered as a result of evaluating some expression, other results are discarded and the function returns Rerr e. In particular, ill-typed expressions evaluate to the error Rabort Rtype_error, and are excluded from all theorems by proof at a prior stage.

Proof of our lemma requires semantics preservation to hold for *all* expressions that we might evaluate; specifically, we must treat any expression in the shape of $f\ x \oplus y$ (cf. Section 3.3). This leads to a proof goal along the lines of

$$f\ x \oplus y \equiv f'\ x\ y \tag{3}$$

where $\equiv$ is taken to mean semantic equality under the evaluate function. The semantics of BVI dictate that the operands xs to an operation expression Op op xs are evaluated in sequence using evaluate (see Figure 1). This necessarily implies that $f\ x$ is evaluated ahead of y. However, on the right-hand side, y is an argument to f' and thus evaluated *prior* to the function call, due to call-by-value semantics. Since y does not depend on global state (cf. Section 3.3), this should not matter, as y is pure and will evaluate to some value. However, guaranteeing that the compound expression $f\ x \oplus y$ is well-typed by assuming that evaluation does not abort with a Rtype_error does not guarantee that the same holds for y, since the evaluation of $f\ x$ can result in any other error. This peculiarity of the verification procedure we use will expose itself whenever we attempt to change the order of evaluation in compound expressions. It is currently circumvented by imposing strong restrictions on y as to provide a static guarantee that it evaluates to a value.

5 Related Work

Burstall and Darlington [1] described a framework for transforming recursive functions into more efficient imperative counterparts. Their approach, however, relies on user-guidance, and is thus not suitable for inclusion in a fully automatic optimizing compiler.

An early systematic account of the transformation described in this paper was given by Wadler [8], with the primary goal of eliminating quadratic list append usage. Since the introduction of tail calls is our primary goal, we have extended it to also treat associative integer arithmetic. A different transformation for introducing accumulators is presented in Kühnemann, et al. [3]. It is, however, limited to unary functions. We are unaware of any compiler which implements this transformation.

Chitil [2] describes an improvement of the short-cut deforestation algorithm which, among other improvements, enables deforestation to act on list producers which consume their own result. It correctly handles the `reverse` example from Section 3.1, but is limited to functions returning lists. As with Kühnemann et al. [3], we are not aware of any compiler which implements it.

Finally, we note that in contrast to other work, our contribution is a fully verified transformation with a machine-checked proof of semantic verification. In addition, it is implemented in a proven-correct compiler, providing not only increased confidence in its correctness, but shows the feasibility of implementing the transformation in practice and integrating it into a larger context.

6 Conclusions

In this paper, we have implemented an optimizing compiler transformation acting on expressions in the BVI intermediate language. The transformation introduces tail recursion in certain recursive functions while ensuring semantic preservation. The implementation has been integrated with the existing structure of the CakeML compiler as a standalone compiler stage. This compiler stage has been verified to not alter the observational semantics of the program under transformation.

To the best of our knowledge, this is the first fully verified implementation of the transformation in any modern compiler. In addition, our contributions make the CakeML compiler the first fully verified compiler which performs this transformation.

Furthermore, our efforts uncover surprising drawbacks in some of the verification techniques currently employed in the BVI compiler phase of the CakeML compiler. The current solution to this issue is unnecessarily restrictive. An alternative approach is to introduce a 'filter' prior to the compiler transformation which recursively replaces all ill-typed expressions with well-typed constants. However, such an approach would introduce new drawbacks, as the filter would traverse parts of the program under compilation while essentially performing no real work. In the future, we will investigate a viable long-term solution to this issue. This would likely require us to come up with a different verification approach for code transformations which alter the order of evaluation.

Acknowledgments. This work was carried out as the author's M.Sc. project at the CSE department of Chalmers University of Technology, Sweden, under the supervision of Magnus Myreen.

State of Formal Proofs. At the time of writing, there are some minor holes in the mechanized proofs. However, the proofs will be completed by the time of TFP. The proofs are available at https://github.com/oskarabrahamsson/cakeml/blob/bvi_tailrec/compiler/backend/proofs/bvi_tailrecProofScript.sml.

References

1. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM (JACM) 24(1), 44–67 (1977)
2. Chitil, O.: Type-inference based short cut deforestation (nearly) without inlining. In: Symposium on Implementation and Application of Functional Languages. pp. 19–35. Springer (1999)
3. Kühnemann, A., Glück, R., Kakehi, K.: Relating accumulative and non-accumulative functional programs. In: International Conference on Rewriting Techniques and Applications. pp. 154–168. Springer (2001)
4. Kumar, R., Myreen, M., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. pp. 179–191. ACM (2014)
5. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: European Symposium on Programming Languages and Systems. pp. 589–615. Springer (2016)
6. Owens, S., Norrish, M., Kumar, R., Myreen, M.O., Tan, Y.K.: Verifying efficient function calls in CakeML. In: ICFP '17: Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming. ACM Press (Sep 2017), to appear
7. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A new verified compiler backend for CakeML (2016)
8. Wadler, P.: The concatenate vanishes. Note, University of Glasgow (1987)