CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Automatically Introducing Tail Recursion in CakeML

Master's thesis in Computer Science – Algorithms, Languages, and Logic

OSKAR ABRAHAMSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Automatically Introducing Tail Recursion in CakeML

OSKAR ABRAHAMSSON

Department of Computer Science and Engineering
Formal Methods Division
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Automatically Introducing Tail Recursion in CakeML
OSKAR ABRAHAMSSON

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: Carlo A. Furia, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Formal Methods Division
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Automatically Introducing Tail Recursion in CakeML
OSKAR ABRAHAMSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this thesis, we describe and implement an optimizing compiler transformation which turns non–tail-recursive functions into equivalent tail-recursive functions in an intermediate language of the CakeML compiler. CakeML is a strongly typed functional language based on Standard ML with call-by-value semantics and a fully verified compiler. We integrate our implementation with the existing structure of the CakeML compiler, and provide a machine-checked proof verifying that the observational semantics of programs is preserved under the transformation. To the best of our knowledge, this is the first fully verified implementation of this transformation in any modern compiler. Moreover, our verification 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.

Keywords: Compiler verification, formal methods, compiler optimizations, functional programming, CakeML, tail recursion

Acknowledgements

I would like to thank my supervisor Magnus Myreen for continuous encouragement, support and helpful ideas throughout my thesis project, as well as for giving me the opportunity to contribute to the CakeML project. I would also like to thank my examiner Carlo A. Furia for providing valuable feedback during the writing of this report. Drafts of this report were read by Maximilian Algehed and Sòlrùn Halla Einarsdòttir.

Oskar Abrahamsson, Gothenburg, July 2017

Contents

Contents **ix**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Preliminaries . . . 1
1.3 Contributions . . . 2
1.4 Thesis structure . . . 3

2 Background **5**
2.1 Notation . . . 5
2.2 Tail-recursive functions . . . 6
2.3 Tail recursion using accumulators . . . 6
2.3.1 Example: List reversal . . . 6
2.3.2 Generalizing the transformation . . . 7
2.4 CakeML . . . 8
2.4.1 The BVI intermediate language . . . 8
2.5 The HOL4 interactive theorem prover . . . 9
2.5.1 Software development in HOL4 . . . 9

3 Transforming BVI functions **11**
3.1 The BVI abstract syntax . . . 11
3.2 Compiling BVI programs . . . 12
3.3 Transforming BVI expressions . . . 12
3.3.1 Example: The factorial in BVI . . . 13
3.3.2 Transforming the tail position . . . 14
3.4 Detecting necessary conditions . . . 20
3.4.1 Inferring the type of BVI expressions . . . 20
3.4.2 Selecting expressions for transformation . . . 21
3.5 Integration with the CakeML compiler . . . 21

4 Proving semantics preservation **25**
4.1 Preliminaries . . . 25
4.1.1 The semantics of BVI . . . 25
4.1.2 Proving theorems about expression semantics . . . 26
4.1.3 Reasoning about divergence . . . 28
4.1.4 Proving theorems about program semantics . . . 29
4.2 Semantics preservation . . . 29
4.2.1 Semantics of programs . . . 31
4.2.2 Semantics of expressions . . . 31
4.2.3 Generalized semantics of expressions . . . 33
4.2.4 Supporting theorems . . . 36
4.3 Limitations . . . 38
4.3.1 The lack of a type system . . . 38
4.3.2 The compiler clock . . . 39

5 Related Work **41**

6 Conclusions and future work **43**

Bibliography **45**

1

Introduction

1.1 Motivation

Modern compilers are complex pieces of software, responsible for translating a large set of input programs to executable machine-code. Optimizing compilers perform a wide range of transformations in order to ensure efficient execution of the resulting machine-code programs. While the desirable outcome of these transformations are usually performance gains, it is also vital that program semantics are preserved, so that the resulting executable code behaves as intended.

CakeML is a functional programming language based on a substantial subset of Standard ML [1]. The CakeML compiler is a verified compiler; that is, for every run of the compiler on a CakeML source document, the compiler has been verified to produce machine-code behaving in accordance to the semantics of the source program. The compiler and its proof of correctness are implemented entirely in the higher-order logic of the HOL4 theorem prover [2].

This thesis describes how a new verified optimization has been added to the CakeML compiler. The optimization is a code transformation, which turns non–tail-recursive functions into tail-recursive functions by automatically introducing accumulator arguments. A tail-recursive function is a function in which all recursive calls are tail calls, i.e. calls that are situated in the value-returning positions of a function body. Although the technique we employ is well known, it is usually performed manually by the programmer at the source level.

As CakeML strives to be the most realistic implementation of a verified modern compiler for a functional programming language [3], efficient and proven-correct optimizations contribute not only towards the project itself, but benefit future efforts in the area of formally verified optimizing compilers.

1.2 Preliminaries

The optimizing transformation described in this thesis allows the CakeML compiler to automatically transform certain recursive functions into equivalent tail-recursive functions. A recursive function is a function which contains a self-reference. In

programming, this entails a function or procedure which calls itself at some point during evaluation. A *tail-recursive* function is a recursive function in which a recursive call is present in a *tail* position of the function definition. Intuitively, a tail position of a function is any part of its body which 'returns' a value. Hence, in a tail-recursive function, the last action performed during evaluation is a recursive call or the evaluation of a non-recursive expression.

Consider two different – but equivalent – recursive definitions of the factorial $n! = n \cdot (n-1) \cdots 2 \cdot 1$ in some fictitious functional language. A recursive definition of the factorial is given by

$$\mathsf{fac}\ n = \mathsf{if}\ n \leq 1\ \mathsf{then}\ 1\ \mathsf{else}\ n \times \mathsf{fac}\ (n-1). \tag{1.1}$$

The self-reference contained in (1.1) exists under a multiplication operator ($\times$). In order for $\mathsf{fac}\ n$ to yield a value for some $n > 1$, we must first compute this multiplication, requiring subsequent recursive computations of fac until the base case $n = 1$ is reached. We give a second definition of the factorial. It is extensionally equivalent to (1.1), but it is defined in such a way that it is tail-recursive:

$$\mathsf{fac'}\ acc\ n = \mathsf{if}\ n \leq 1\ \mathsf{then}\ acc\ \mathsf{else}\ \mathsf{fac'}\ (n \times acc)\ (n-1). \tag{1.2}$$

Evaluating $\mathsf{fac'}\ 1\ n$ yields the same value as the evaluation of $\mathsf{fac}\ n$. However, in (1.2) the recursive call sits in tail position. This turns out to be crucial from a computational standpoint. Evaluating (1.1) for some $n > 1$ demands subsequent recursive evaluations of fac before yielding a value. Each successive recursive call requires a small amount of bookkeeping, which consumes a non-negligible amount of time and stack space. In (1.2) however, the need for bookkeeping is eliminated altogether, as computation of the function arguments can be performed in-place, and the last recursive call to $\mathsf{fac'}$ can return to the original caller. Consequently, our tail-recursive factorial can be evaluated with stack space consumption bounded by a constant, whereas the previous definition consumed additional stack space and time proportional to the number of recursive calls.

1.3 Contributions

In this report, we describe a fully verified implementation of an optimizing code transformation for functional programs, which automatically introduces tail recursion using accumulators. The implementation acts on an intermediate language in the fully verified CakeML compiler. 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. To the best of our knowledge, ours is the first proven-correct implementation of this transformation, existing in a fully verified compiler.

Proving the correctness of the implemented transformation exposes some surprising shortcomings in the techniques used for the verification of the CakeML compiler. 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 the OCaml and Poly/ML compilers in certain benchmarks [4]. However, the verification of the transformation presented in this work reveals drawbacks to this approach. We discuss these drawbacks and suggest workarounds.

1.4 Thesis structure

Chapter 2 gives the relevant background on the topic of the thesis. We start by detailing the notation used throughout this report (Section 2.1) followed by an extended account of tail-recursion (Section 2.2). Following this, we introduce a code transformation for automatically introducing tail recursion (Section 2.3). In addition, we introduce the CakeML language and compiler, as well as the intermediate language BVI of the CakeML compiler, on which our transformation acts (Section 2.4). Lastly, the HOL4 theorem prover within which our work is carried out is described (Section 2.5).

Chapter 3 starts with a description of BVI (Section 2.4). This is followed by a description of the BVI compiler stage (Section 3.2). We then proceed to give a detailed account of how the implementation of the transformation for BVI expressions is carried out in the HOL4 theorem prover (Section 3.3), including the static analysis required to transform expressions (Section 3.4). Finally, we conclude with a description of how the transformation is integrated into the CakeML compiler as a stand-alone compiler stage (Section 3.5).

Following this, the implementation from Chapter 3 is verified correct in Chapter 4. The chapter starts by giving the necessary background for carrying out formal reasoning about the semantics of BVI programs (Section 4.1). In particular, we account for the semantics of BVI (Section 4.1.1), what correctness entails, and how proofs of correctness are carried out (Sections 4.1.2 through 4.1.4). This is followed by a detailed account of the most important correctness theorems for the implementation from Chapter 3, as well as descriptions on how proofs are carried out for these theorems (Section 4.2). We conclude the chapter with a description of some surprising limitations in the verification techniques used which are exposed when carrying out proofs for some of our theorems (Section 4.3).

Chapter 5 puts our contributions in context with related work done on similar compiler optimizations. 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.

Finally, we conclude our report in Chapter 6 with a discussion of our results, and suggest suitable topics for future work.

2

Background

This chapter provides the necessary background for the work carried out in this thesis. We start by describing the notation employed in the remainder of the report (Section 2.1). The concept of tail recursion, as well as the benefits of tail-recursive functions when compiling functional programs, is further elaborated on (Section 2.2). We introduce a transformation for automatically introducing tail-recursion by means of an example, and then proceed with a more general description in an algorithmic fashion (Section 2.3). This is followed by a description of the CakeML language and compiler (Section 2.4), as well as an introduction to the BVI intermediate language which our implementation will target (Section 2.4.1). Finally, we introduce to the HOL4 proof assistant, in which our implementation and verification efforts will be carried out (Section 2.5).

2.1 Notation

The notation we employ is as follows. CakeML code is typeset in sans-serif with comments enclosed by (* ... *). Functions written in CakeML are declared using the keyword fun:

fun < identifier > [arguments] = <body>

The majority of the source code listings in this report consists of function definitions and theorems in higher-order logic (HOL) (see Section 2.5). These are typeset automatically using the LaTeX generation facilities of the HOL4 theorem prover. The syntax of HOL closely resembles that of ML-style languages: constructors, keywords and function names are typeset in sans-serif. Variables are written in *italic*. Records are declared using

my_record =
  <| field1 := $v1$; field2 := $v2$; ... |>

and use . (dot) for projection and with for update. Logical equivalence is denoted by $\iff$. Implication and case-style pattern matching is denoted by $\Rightarrow$. All other logical connectives retain their usual meaning.

2.2 Tail-recursive functions

A tail-recursive function is a function which performs a recursive call to itself as its final action before returning a value. In functional programming, this entails that the function contains a recursive call in one of its *tail positions*, that is, those positions in the function definition which evaluate to a value.

Evaluating a recursive call in tail position enables reuse of the current stack frame to a greater extent, as no additional bookkeeping needs to be performed to ensure that the function returns 'to the right place'; one can simply branch unconditionally into the body of the callee. Moreover, the locations of any arguments to the function can potentially be reused instead of being pushed onto the stack or placed in registers to satisfy calling conventions. When this is the case, the function itself can be compiled into a corresponding loop statement, keeping stack usage constant throughout its execution. This approach is commonly referred to as *tail-call elimination*. It is highly advantageous when applicable, as it allows a single function to perform recursive calls without additional stack space consumption. This advantage is even more pronounced in implementations of functional programming languages, where recursive functions act as the drop-in replacement for loops.

2.3 Tail recursion using accumulators

In this section, we describe a code transformation for automatically transforming recursive functions into tail-recursive functions. Although the transformation is well-known [5], it is usually performed by the programmer at the source level. We start by providing an informal description of the procedure through a worked example in Section 2.3.1. The example is generalized to an algorithmic description of the steps of the transformation in Section 2.3.2.

2.3.1 Example: List reversal

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 apply the inductive argument once more, 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 []
```

2.3.2 Generalizing the transformation

The transformation steps applied in Section 2.3.1 can be generalized to work with any operation in tail position, so long as it is associative and has an identity element. Let $+$ 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

$$f\ x + a \tag{2.1}$$

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

$$f'\ x\ a = f\ x + a\ . \tag{2.2}$$

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 $e := f\ x + y$ for some x, y, replace e by $f'\ x\ (y + a)$, where f' is an unused function name.
2. For all other expressions e in tail position, replace them with the expression $e' := e + a$.

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 $+$ by setting $f\ x = f'\ x\ 0$.

We will return to this transformation in Chapter 3 as we provide an implementation in higher-order logic, and describe the steps of its subsequent inclusion in the CakeML compiler.

2.4 CakeML

CakeML [6] 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, MIPS and RISC-V. 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.

2.4.1 The BVI intermediate language

The CakeML compiler recently received a new backend [3] 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 first-order functional language with de Bruijn-indices.[1] Like all other ILs in the new CakeML compiler backend, its formal semantics is specified in terms of a functional big-step style [7].

The transformation described in this thesis 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*. Our motivations 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.

[1] The usage of de Bruijn-indices is common to compilers, as it eliminates the need for variable renaming.

- 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 entries in the code table.

Chapter 3 gives an account of the BVI abstract syntax (Section 3.1). In addition to this, Chapter 4 includes a description of the semantics of the language (Section 4.1.1).

2.5 The HOL4 interactive theorem prover

The implementation of the CakeML compiler as well as its proof of correctness is carried out wholly within the HOL4 interactive theorem prover [2], and by extension, so is all work presented in this thesis. The HOL theories representing the additions to the CakeML compiler resulting from this work can be found at the CakeML GitHub repository at

https://github.com/cakeml/cakeml.

The HOL4 system implements the basic inference rules of higher-order logic as a library in the ML programming language. Proofs are produced by applying so-called *proof tactics*; that is, functions in the ML language which decompose the proof goal into a list of sub-goals, and provide a 'joining' function which produces a proof for the original goal given proofs of the sub-goals. In this way, the theorem is proven somewhat recursively by the user, by decomposing the goal into manageable pieces, and proving these separately.

2.5.1 Software development in HOL4

The higher-order logic of HOL4 supports typed functions, datatype declarations and pattern matching. As such, it can be utilized as a purely functional programming language: programs are written inside the logic and interpreted using an interpreter which implements the semantics of the logic.

The CakeML software ecosystem includes a library which performs proof-producing synthesis of CakeML code from the higher-order logic [8]: functions in the logic are converted to CakeML functions using a fully verified procedure, enabling the user to use theorems about the HOL functions to reason about their corresponding CakeML counterparts. This implies that although the compiler is implemented almost entirely in the logic, it is able to bootstrap itself (i.e. compile itself) by first producing fully verified CakeML expressions from its own definitions, and then use these expressions as input to the compiler functions in the

logic. The result is a fully verified binary of the compiler, compiled by a fully verified compiler.

The proof-producing synthesis facilities allow for all ML-like functions in HOL to be implemented as pure CakeML programs, and proven correct inside the confines of the logic. In addition to this, CakeML recently received support for so-called *characteristic formulae* [9] (a form of Hoare-triples for ML programs). The addition of characteristic formulae enables verification of impure CakeML programs within the logic, arguably making the CakeML software suite the most sophisticated system for writing fully verified functional programs to date.

3

Transforming BVI functions

This chapter describes an implementation of the transformation from Section 2.3.2 in higher-order logic, and how it fits into the existing structure of the CakeML compiler. We start with a description of the BVI abstract syntax (Section 3.1), followed by a description of the BVI stage of the CakeML compiler (Section 3.2). A detailed description of the implementation of our transformation transformation is given (Section 3.3), and we construct a simple check which allows us to determine at compile-time which expressions are eligible for transformation (Section 3.4). Finally, we show how the complete implementation is turned into a stand-alone compiler stage for inclusion into the existing BVI phase of the CakeML compiler (Section 3.5).

3.1 The BVI abstract syntax

The abstract syntax of the BVI language is shown in Figure 3.1. The type `num` corresponds to natural numbers, and `op` to one of the languages primitive operations.

```
exp =
   Var num                          (* de Bruijn-variable *)
 | If exp exp exp                   (* If-then-else *)
 | Let (exp list) exp               (* Let-binding *)
 | Raise exp                        (* Raise exception *)
 | Tick exp                         (* Decrement semantics clock *)
 | Call num (num option)
      (exp list) (exp option)       (* Function call *)
 | Op op (exp list)                 (* Primitive operation *)
```

Figure 3.1: The abstract syntax of BVI.

The meaning of the BVI expressions is as follows: Var i denotes a variable with de Bruijn-index i, Raise *exc* raises an exception *exc*, and Op *op xs* denotes a primitive operation *op* on the expressions *xs*. The expressions If and Let have their usual meaning. A Call expression is of the form Call *ticks dest args hdl*, where *dest* denotes an address in the code table to the function being called, and *args* the function arguments. Optionally, the address *hdl* to a function acting as an exception handler is present. The *ticks* parameter to Call, and the Tick expression are related to the verification of semantics preservation, and are in practice no-ops. Thus, we defer their treatment until Chapter 4.

3.2 Compiling BVI programs

During the BVI stage of the compiler, programs are stored in an immutable code store, or *code table*. All entries in the compiler code-table are BVI functions. Each entry is defined by a tuple

$$((nm : \texttt{num}),(ar : \texttt{num}),(exp : \texttt{exp})) \ .$$

Here, *nm* is an address used to index into the table, *ar* denotes the arity of the function, and *exp* is the BVI expression constituting its body.

In general, most compiler optimizations will not require access to the code table, as they transform programs on an *expression-by-expression* basis. However, the optimization discussed in this thesis requires

(i) access to function addresses, to detect recursion.

(ii) access to the code table, to insert auxiliary definitions.

We will therefore implement it as a stand-alone compiler stage that transforms an entire BVI program 'at once'. The implementation of the transformation itself is described in Sections 3.3 through 3.4. It is then made to act on the entire code table in Section 3.5.

3.3 Transforming BVI expressions

This section provides an implementation of the transformation from Section 2.3.2 which rewrites BVI expressions. The implementation supports on associative integer arithmetic (i.e. addition and multiplication), since these operations are primitive to the BVI language, and thus easily detected compile-time. However, the implementation can be extended to work with any associative operation detectable at compile-time. For the remainder of this section we will assume that the following is known whenever an expression is transformed:

- It is known which operator sits in tail position. If there are several, we know of *one*, and this operator is fixed. Clearly, there may be different operators in different tail positions, and as such, we must parametrize our transformation on a single one.

- We have some form of assurance that all tail-positions return a value of the correct type, i.e. an integer.

In Section 3.4, we will ensure that these assumptions hold for the expressions on which we apply the transformation, by means of static analysis.

3.3.1 Example: The factorial in BVI

Recall the factorial fac and fac' defined in Section 1.2. We give CakeML implementations of fac and fac':

```
fun fac n = if n ≤ 1 then 1 else fac (n − 1) ∗ n

fun fac' n acc = if n ≤ 1 then acc else fac' (n − 1) (n ∗ acc)
```

The definition of fac compiles into the following equivalent BVI expression:

```
bvifac =
 If (Op LessEq [Var 0; Op (Const 1) []]) (Op (Const 1) [])
  (Op Mult
    [Call 0 (Some 1) [Op Sub [Var 0; Op (Const 1) []]] None;
     Var 0])
```

Here, we have assigned the function bvifac the code table address 1, and represent the single variable n by Var 0. The recursive nature of bvifac is made explicit by the destination Some 1 of the Call sub-expression. In the same way, we give an equivalent BVI expression for fac':

```
bvifac' =
 If (Op LessEq [Var 0; Op (Const 1) []]) (Var 1)
  (Call 0 (Some 2)
    [Op Sub [Var 0; Op (Const 1) []];
     Op Mult [Var 0; Var 1]] None)
```

In this definition, Var 1 represents the accumulating argument. Our goal for the remainder of this section is then to devise an implementation which transforms the expression bvifac into bvifac'.

3.3.2 Transforming the tail position

We return briefly to the transformation outlined in Section 2.3.2. The transformation is applied on the tail positions of a function body, and any tail position occupied by a recursive expression $f\ x + y$ is replaced by a recursive call to a new function f':

$$f\ x + y \mapsto f'\ x\ y. \tag{Rule 3.1}$$

Any tail position containing any other type of expression e is simply transformed according to

$$e \mapsto e + a \tag{Rule 3.2}$$

where a is a variable pointing to the accumulating argument. From this it seems that a natural starting point for our implementation is to provide a function which performs the transformations given by Rule 3.1 and Rule 3.2. In order to perform these transformations on BVI expressions we require the following information at hand:

(i) The de Bruijn-index of the accumulating argument.

(ii) The code table address to the function which contains the expression.

(iii) An unused code table address.

(iv) The operation for which the transformation is to be applied.

For reasons of simplicity, we will let the accumulating argument be the last (or rightmost) argument of the function. The index of this argument can be computed by starting from the function arity, incrementing this by one, and subsequently incrementing it each time we introduce new variable binders (i.e by Let expressions).

Transforming non-recursive tail positions

We return to bvifac (Section 3.3.1). The expression has two tail positions; the first is occupied by the integer literal

$$\mathsf{Op}\ (\mathsf{Const}\ 1)\ []$$

which does not contain any recursive calls. It is hence subject to Rule 3.2. We introduce a function apply_op which is parametrized on an operation and two expressions which are to be joined under the operation:

$$\mathsf{apply\_op}\ op\ e_1\ e_2 = \mathsf{Op}\ (\mathsf{to\_op}\ op)\ [e_1;\ e_2]$$

The definition contains an auxiliary function to_op. The reason for this is to limit the number of cases for apply_op generated by HOL; although our transformation only treats addition and multiplication, the BVI language supports in excess of 40 different operations to be used with Op. Parametrizing apply_op on this type would result in the creation a pattern matching case for each of these. For this reason we introduce a binary datatype `assoc_op` and parametrize apply_op on this datatype (see Figure 3.2). In fact, avoiding excessive pattern matching is the reason for using apply_op in the first place.

```
assoc_op = Plus | Times | Noop

to_op Plus = Add
to_op Times = Mult
to_op Noop = Const 0
```

Figure 3.2: The `assoc_op` type and to_op.

Transforming recursive tail positions

The remaining tail position in bvifac is occupied by the expression

$$\mathsf{Op\ Mult\ [Call\ 0\ (Some\ 1)\ [\ldots]\ \ldots;\ Var\ 0]}$$

under which the recursive call sits. In order to transform this expression, we need to extract the Call and Var expressions under the Op, extract the arguments from the Call, and construct a tail call according to Rule 3.1. For these purposes we introduce three functions (see Figure 3.3):

(i) get_bin_args, which extracts the arguments to a binary operation (if any).

(ii) args_from, which extracts the arguments from a Call expression.

(iii) push_call, which given the outputs of get_bin_args and args_from applies the transformation given by Rule 3.1 to produce a Call expression.

We compose these three functions into a function mk_tailcall, which performs the transformation defined by Rule 3.1, as desired:

```
mk_tailcall n op name acc exp =
 case get_bin_args exp of
   None ⇒ dummy_case
 | Some (call,exp') ⇒ push_call n op acc exp' (args_from call)
```

```
get_bin_args (Op v0 [e1; e2]) = Some (e1,e2)
get_bin_args _ = None

args_from (Call t (Some d) as hdl) = Some (t,d,as,hdl)
args_from _ = None

push_call n op acc exp (Some (ticks,dest,args,handler)) =
  Call ticks (Some n) (args ++ [apply_op op exp (Var acc)]) handler
push_call v0 v1 v2 v3 None = dummy_case
```

Figure 3.3: The definitions of get_bin_args, args_from and push_call.

Applying mk_tailcall to the multiplication in the tail position of bvifac with the correct parameters results in an expression corresponding to the second tail position in bvifac':

```
Call 0 (Some 2)
       [Op Sub [Var 0; Op (Const 1) []];
        Op Mult [Var 0; Var 1]] None
```

Finally, since mk_tailcall is applicable only on tail positions, we implement a function rewrite_tail, which given an expression recursively applies mk_tailcall to its tail positions (see Figure 3.4).

Rearranging using associativity and commutativity

Although mk_tailcall seems to mimic closely the transformation described in Chapter 2, it is unnecessarily weak. Consider the following modification to fac, using commutativity:

```
fun fac n = if n ≤ 1 then 1 else n * fac (n − 1)
```

An expression like n * fac (n - 1) will not be accepted by mk_tailcall, as it expects the recursive call to sit at the left-hand side of the operation. However, since multiplication is commutative, we could clearly swap the recursive call with n. Likewise, the (somewhat artificial) function foo defined by

```
fun foo n = if n ≤ 1 then 1 else n * (foo (n − 1) * 1)
```

can be transformed by first rewriting its tail position using commutativity and associativity:

```
fun foo n = if n ≤ 1 then 1 else foo (n − 1) * (n * 1)
```

```
rewrite_tail n op name acc (Let xs x) =
  Let xs (rewrite_tail n op name (acc + length xs) x)
rewrite_tail n op name acc (Tick x) =
  Tick (rewrite_tail n op name acc x)
rewrite_tail n op name acc (Raise x) = Raise x
rewrite_tail n op name acc (If x1 x2 x3) =
  let y2 = rewrite_tail n op name acc x2;
      y3 = rewrite_tail n op name acc x3
  in
    If x1 y2 y3
rewrite_tail n op name acc (Var v) = Var v
rewrite_tail n op name acc (Call t d xs h) =
  Call t d xs h
rewrite_tail n op name acc (Op v21 v22) =
  mk_tailcall n op name acc (Op v21 v22)
```

Figure 3.4: The definition of rewrite_tail.

Care must be taken, however, to not move expressions around that would incur side-effects during evaluation, as changing their order of appearance under the multiplication operator would change the order in which they are evaluated.

We introduce a function rewrite_op which recursively performs the associative and commutative swaps required on BVI expressions, and strengthen mk_tailcall by calling rewrite_op prior to push_call:

```
mk_tailcall n op name acc exp =
  case rewrite_op op name exp of
    (T,exp2) ⇒
      (case get_bin_args exp2 of
         None ⇒ dummy_case
       | Some (call,exp3) ⇒
           push_call n op acc exp3 (args_from call))
  | (F,exp2) ⇒ apply_op op exp2 (Var acc)
```

The definition of rewrite_op (see Figure 3.5) appears slightly involved, although its workings are simple:

1. First, op_eq determines if the expression is an operation of the correct sort. If it is not, we do nothing.

```
rewrite_op op name exp =
 if ¬op_eq op exp then (F,exp)
 else
  case get_bin_args exp of
   None ⇒ (F,exp)
  | Some (x1,x2) ⇒
    let (r1,y1) = rewrite_op op name x1;
        (r2,y2) = rewrite_op op name x2
    in
     case
      (is_rec_or_rec_binop name op y1,
       is_rec_or_rec_binop name op y2)
     of
      (T,T) ⇒ (F,exp)
     | (T,F) ⇒
       if no_err y2 then (T,assoc_swap op y2 y1) else (F,exp)
     | (F,T) ⇒
       if no_err y1 then (T,assoc_swap op y1 y2) else (F,exp)
     | (F,F) ⇒ (F,exp)
```

Figure 3.5: The definition of rewrite_op which rearranges BVI expressions using associativity and commutativity.

2. If get_bin_args returns a positive result, rewrite_op applies itself recursively on its sub-expressions in a bottom-up manner.

3. Applying is_rec_or_rec_binop determines if an expression is one of

   (i) a recursive call

   (ii) an operation conforming to the form of Rule 3.1 from Section 3.3.2.

   Moreover, no_err tries to determine whether or not evaluating an expression can incur side-effects.

4. Finally, assoc_swap utilizes associativity and commutativity to rearrange an expression that is on correct form.

Note that we let rewrite_op return a boolean along with its resulting expression to signify if a rewrite occurred or not. As we will see in Section 3.4, it turns out that

```
op_eq Plus (Op Add v0) ⟺ T
op_eq Times (Op Mult v1) ⟺ T
op_eq _ _ ⟺ F

is_rec_or_rec_binop name op exp ⟺
 is_rec name exp ∨
 op_eq op exp ∧
 case get_bin_args exp of
   None ⇒ F
 | Some (x1,x2) ⇒ is_rec name x1 ∧ no_err x2

assoc_swap op from into =
 if ¬op_eq op into then apply_op op into from
 else
   case get_bin_args into of
     None ⇒ apply_op op into from
   | Some (x1,x2) ⇒ apply_op op x1 (apply_op op from x2)
```

Figure 3.6: Definitions of the auxiliary functions used in rewrite_op.

rewrite_op can also be used to statically determine whether or not an expression is eligible for rewrite in the first place.

3.4 Detecting necessary conditions

Recall the assumptions made in Section 3.3. Firstly, in order to ensure that a BVI expression can be transformed, we need to ensure that its tail positions evaluate to values of the correct type. Additionally, we require a procedure for detecting whether or not an expression is eligible for transformation. Both types of assurances will be given by performing static analysis on the expressions of a BVI program. In Section 3.4.1, we describe a pessimistic procedure for detecting integer expressions in BVI. Following this, we outline how to detect which expressions are eligible for transformation in Section 3.4.2.

3.4.1 Inferring the type of BVI expressions

Since the BVI language has no types, we cannot directly query for the types of expressions. The expressions which are known at compile-time to return integers are

(i) Integer arithmetic, e.g. addition, subtraction, etc.

(ii) Integer literals.

(iii) An expression with any of the above in tail position.

We define a predicate is_ok_type which checks if an expression satisfies the above:

$$\begin{aligned}
&\mathsf{is_ok_type}\ (\mathsf{Op}\ op\ v_0) \iff \mathsf{is_arithmetic}\ op \\
&\mathsf{is_ok_type}\ (\mathsf{Let}\ v_1\ x_1) \iff \mathsf{is_ok_type}\ x_1 \\
&\mathsf{is_ok_type}\ (\mathsf{Tick}\ x_1) \iff \mathsf{is_ok_type}\ x_1 \\
&\mathsf{is_ok_type}\ (\mathsf{If}\ v_2\ x_2\ x_3) \iff \mathsf{is_ok_type}\ x_2 \wedge \mathsf{is_ok_type}\ x_3 \\
&\mathsf{is_ok_type}\ _ \iff \mathsf{F}
\end{aligned}$$

The definition of is_arithmetic allows for operations Add, Sub, Mult, Div, Mod and literals Const i. As a consequence of the above, our transformation will not activate on functions that contain variables in the base case. Chapter 4 contains a discussion on potential solutions to this issue.

3.4.2 Selecting expressions for transformation

In order to decide which BVI expressions are eligible for transformation, we revisit the criteria for the transformation presented in Section 2.3.2. The criteria presented there require only that *some* tail position in the expression under consideration should be on the form of (Rule 3.1). Moreover, the operation in this expression should be associative and have an identity element. Since the integer arithmetic supported by our implementation satisfies the latter, it will suffice to check that tail positions are on a form that can be rewritten.

It turns out that we have already implemented this functionality once in rewrite_op (see Figure 3.5), which returns a boolean declaring if the rewrite succeeded. Based on this, we construct a static check tail_is_ok. Its definition is shown in Figure 3.7. The function tail_is_ok works by recursively traversing all sub-expressions of its input. If it encounters an expression Op *op xs* in tail position, it ensures that

(i) the operation *op* is one of Add or Mult.

(ii) an application of rewrite_op to the expression – parametrized on the operation *op* – succeeds.

Additionally, we decorate the return values of tail_is_ok with a boolean. This boolean which allows us to deduce if the rightmost expression in an If branch was eligible for transformation, and will be helpful when verifying the correctness of our transformation, since there is otherwise no way to decide which branch of the If-expression that resulted in the operation when checked.

Finally, we compose is_ok_type and tail_is_ok into a static check which ensures that we only transform eligible expressions:

```
check_exp name exp =
  if ¬is_ok_type exp then None else tail_is_ok name exp
```

When it succeeds, check_exp returns an operation which can act as input to the rewrite_tail function.

3.5 Integration with the CakeML compiler

So far, we have given an implementation of a code transformation which introduces tail recursion using accumulators, as well as a set of static checks which ensures that the transformation is only applied to expressions which will be correctly transformed, i.e. in such a way that the observational semantics are preserved. Our final order of business before concluding this chapters is the construction of a standalone compiler stage in the CakeML compiler based on our implementation. To this end, we will introduce two new functions:

```
tail_is_ok name (Var v0) = None
tail_is_ok name (Let v1 x1) =
 tail_is_ok name x1
tail_is_ok name (Tick x1) =
 tail_is_ok name x1
tail_is_ok name (Raise x1) = None
tail_is_ok name (If v2 x2 x3) =
 let inl = tail_is_ok name x2;
     inr = tail_is_ok name x3
 in
  case (inl,inr) of
    (None,None) => None
  | (None,Some (v4,iop')) => Some (T,iop')
  | (Some (v6,iop),None) => Some (F,iop)
  | (Some (v6,iop),Some v8) => Some (T,iop)
tail_is_ok name (Call v3 v4 v5 v6) = None
tail_is_ok name (Op op xs) =
 if op = Add ∨ op = Mult then
  let iop = from_op op
  in
   case rewrite_op iop name (Op op xs) of
     (T,v3) => Some (F,iop)
   | (F,v3) => None
 else None
```

Figure 3.7: The definition of tail_is_ok.

```
compile_exp n name num_args exp =
 case check_exp name exp of
  None ⇒ None
 | Some (v1,op) ⇒
    let (_,opt) = rewrite_tail n op name num_args exp;
        aux = let_wrap num_args (id_from_op op) opt
    in
     Some (aux,opt)
```

Figure 3.8: The definition of compile_exp.

(i) A function compile_exp which acts on a single code table entry. Its purpose is to perform the static checks in check_exp, and apply the transformation rewrite_tail when possible.

(ii) A function compile_prog, which maps compile_exp over the entries of the code table.

The definition of compile_exp is shown in Figure 3.8. We let compile_exp return an option value carrying two BVI expressions – an expression transformed by rewrite_tail and an auxiliary definition. The function compile_prog (see Figure 3.9) simply traverses the code table (which is here represented as a list) and inserts the results from compile_exp into the code table, if any. It makes use of a parameter n which is the next 'free' address in the code table. Following the insertion of an additional function into the code table, n is incremented by two, since at this stage of compilation, odd code table entries after n are guaranteed to be free.

Finally, an auxiliary definition is created using a call to let_wrap.

```
let_wrap num_args id exp =
 Let (genlist (λ i. Var i) num_args ++ [id]) exp
```

Here, genlist $(\lambda\, i.$ Var $i)$ k generates a list of variables Var 0, Var 1, ..., Var $(k-1)$ and thus 'copies' the entire environment.

Although it would suffice for let_wrap to simply create a function call to the optimized expression (see Figure 3.9), we are unable to do so at this point, since a direct proof for the correctness of this approach leads to some surprising difficulties – the particularities are described in Section 4.3. The current definition of let_wrap simply extends the environment with the identity for the accumulating variable, and inlines the remainder of the transformed expression. Although not

```
compile_prog n [] = (n,[])
compile_prog n ((nm,args,exp)::xs) =
 case compile_exp n nm args exp of
  None ⇒
   let (n1,ys) = compile_prog n xs in (n1,(nm,args,exp)::ys)
 | Some (exp_aux,exp_opt) ⇒
   let (n1,ys) = compile_prog (n + 2) xs
   in
    (n1,(nm,args,exp_aux)::(n,args + 1,exp_opt)::ys)
```

Figure 3.9: The definition of compile_prog.

very elegant, at the very least we manage to avoid the extra overhead that an additional function call would incur. We discuss potential approaches to avoid let_wrap in Chapter 6, but leave these as future work.

4

Proving semantics preservation

In this chapter we describe the process of verifying the correctness of the compiler transformation introduced in Chapter 3. The notion of correctness in this context entails the preservation of *observational semantics* under a transformation. We start by providing a foundation of the concepts needed in order to perform formal reasoning about the properties of sentences in the BVI language (Section 4.1). In particular, we give the abstract syntax of BVI (Section 4.1.1), followed by the details on how properties about the semantics of BVI expressions are stated (Section 4.1.2). This is followed by a description of the workings of the CakeML compiler in the BVI stage, as well as the details on how semantic properties of BVI programs are verified (Sections 4.1.3 and 4.1.4). The bulk of the chapter is dedicated to the verification of the semantic preservation of the compiler transformation discussed in this thesis (Section 4.2). We state a number of theorems describing the properties of the transformation when acting upon BVI programs, and describe how these theorems are proven (Sections 4.2.1 through 4.2.4). Lastly, we conclude with the surprising discovery of some drawbacks in the verification methodology applied in this work, as well as in a large part of the CakeML compiler. We detail how this affects our implementation, and discuss potential workarounds to these drawbacks (Section 4.3).

4.1 Preliminaries

4.1.1 The semantics of BVI

Like most intermediate languages in the CakeML compiler, the semantics of BVI is defined in a functional big-step style using an interpreter function [7] called evaluate. Although functional big-step semantics are usually defined as a relation, defining a semantics in this way naturally gives rise to a large number of cases that need to be treated when carrying out proofs. Defining the semantics as an interpreter leads to simpler proofs, as it enables us to prove theorems regarding program semantics by performing induction on the recursive cases of the interpreter function.

The interpreter evaluate (see Figure 4.1) takes as input a list of expressions, an environment, and a compiler *state*. The environment is represented as a list of concrete values. The compiler state is the following record type. Here, refs is a mapping from an identifier to a concrete pointer, and global is a reference to a dynamic array used for storing global variables. The field ffi contains a state which tracks the calls made to the foreign function interface (FFI). Lastly, clock is a natural number used by the semantics to track divergence (see Section 4.1.3).

```
α state =
 <| refs : (num ↦ v ref);          (* pointers to ref:s *)
    clock : num;                   (* the compiler clock *)
    global : (num option);         (* pointer to global variables *)
    code : ((num × exp) num_map);  (* compiler code table *)
    ffi : (α ffi_state)            (* FFI state *)
 |>
```

An application evaluate (*xs*,*env*,*s*) for some expression list *xs*, some environment *env* and some state *s* will have one of two different outcomes: either the evaluation succeeds for all expressions in *xs*, in which case evaluation results in Rval *vs*, where *vs* is a list of concrete values, each one corresponding to an expression in *xs*. Otherwise, should the evaluation fail for some expression in *xs*, the result is Rerr *err*, where *err* is the error from the expression that first failed in *xs*. The errors carried by the Rerr constructor are divided into two categories:

(i) Rraise *a*, resulting from an expression which raises an exception. Here *a* is the result of evaluating *exc* in the BVI expression Raise *exc*.

(ii) Rabort *e*, where *e* is one of:

- Rtimeout_error, if the evaluation of some expression in *xs* diverged (see Section 4.1.3).
- Rtype_error for all other types of errors. This includes the results of evaluating ill-typed expressions, out of bounds variable accesses, etc.

In addition to the results Rval and Rerr, evaluate will also return a post-state. Since BVI is an impure language, this state may be different from the pre-state.

4.1.2 Proving theorems about expression semantics

Theorems about the semantics of expressions are stated with evaluate and proven using recursive induction on the cases of evaluate. In general, such theorems will

```
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 v10,s1) => (Rerr v10,s1)
evaluate ([Var n],env,s) =
 if n < length env then (Rval [el n env],s)
 else (Rerr (Rabort Rtype_error),s)
evaluate ([If x1 x2 x3],env,s) =
 case evaluate ([x1],env,s) of
  (Rval vs,s1) =>
   if Boolv T = hd vs then evaluate ([x2],env,s1)
   else if Boolv F = hd vs then evaluate ([x3],env,s1)
   else (Rerr (Rabort Rtype_error),s1)
 | (Rerr v7,s1) => (Rerr v7,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 (v3,v4) => (Rval [v3],v4)
    | Rerr e => (Rerr e,s'))
 | (Rerr v10,s') => (Rerr v10,s')
evaluate ([Tick x],env,s) =
 if s.clock = 0 then (Rerr (Rabort Rtimeout_error),s)
 else evaluate ([x],env,dec_clock 1 s)
```

Figure 4.1: Select cases of the interpreter evaluate, which defines the semantics of BVI.

be of the form

$$\begin{aligned} \vdash\ &\mathsf{evaluate}\ (xs,env,s) = (r,t) \wedge \\ &\quad r \neq \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error}) \Rightarrow \\ &\quad\ \mathsf{evaluate}\ (\mathsf{map}\ f\ xs,env,s) = (r,t) \end{aligned}$$

where f is some transformation on BVI expressions. The above theorem states the following.

> When the expressions xs are evaluated with the environment env from the state s, and this evaluation terminates with result r and post-state t, and r is not a Rtype_error, then the expressions xs evaluate to the same result and state under the transformation f, within the same environment and from the same state.

If the above holds, we say that the semantics of the expressions xs are preserved under the transformation f. Note that the theorem assumes that the result r is not an error of the type Rtype_error. The reason for this is that Rtype_errors do not manifest themselves in well-typed programs. Since the type inference algorithm used in the CakeML compiler comes with proofs of soundness and completeness [10], the possibility of encountering such expressions is eliminated in a proof for a prior stage of compilation.

4.1.3 Reasoning about divergence

Allowing computations to diverge (i.e. 'loop forever' without terminating) is arguably a desirable feature for any programming language to be usable in practice. However, the presence of divergence incurs some additional difficulties when proving the semantic equivalence of programs. In particular, the semantics of the language needs to correctly reflect whether or not a computation diverges. The CakeML compiler verification handles this by keeping a logical counter – the *clock* – in the semantic state of each intermediate language. The approach is described in detail in [7]. We give a short summary here.

Since BVI does not support any loop-type constructs, the only way to achieve divergence is by performing recursive calls endlessly. Hence, whenever a function call is evaluated in the semantics, the compiler clock is decremented. When the clock reaches zero, the evaluation terminates with an Rabort Rtimeout_error, signaling divergence. The initial value of the clock is determined implicitly in the top-level proofs; to prove termination it suffices to show the existence of *some* clock value for which the program does not diverge. This type of reasoning has the unpleasant side effect that any removal or introduction of function calls – for instance by inlining, or introducing an auxiliary wrapper as by the transformation

in Chapter 2 – changes the program's evaluate semantics, even though it does not affect its observational semantics, i.e. semantics (see Figure 4.2).

As inlining of function definitions is a common compiler optimization, the BVI language includes an additional expression Tick. The expression Tick *exp* is semantically equivalent to the expression *exp* apart from the fact that it decrements the compiler clock. In this way, the semantics of any inlined definition is preserved by wrapping it in a Tick constructor. The other way around is less trivial. Whenever additional function calls are introduced, additional clock ticks are consumed during evaluation. However, since the amount of introduced function calls has to be finite, we allow for a finite increase in clock ticks when evaluating a transformed expression, so long as the results of the evaluation are preserved. Proofs involving an increase in clock ticks are greatly complicated; the goal involves an existential quantifier, requiring that we provide a witness declaring by how many ticks the clock should be increased. In general, this prevents us from fully expanding the goal before a witness is provided. In addition, it is possible to create situations where an expanded goal is required for a witness to be provided (Section 4.3). As a consequence, we avoid theorems involving clock increments when possible.

4.1.4 Proving theorems about program semantics

So far we have discussed the semantics of BVI expressions, and the verification of the semantics of expressions in isolation. However, the transformation from Chapter 3 is implemented as a stand-alone compiler stage which acts on the entire BVI code table. Hence, in addition to providing theorems which reason about the transformation of single expressions, we are also required to provide a higher-level semantics theorem, stating that the semantics of any *program* is preserved under our compiler stage.

A BVI program is defined as a compiler code table together with a starting address (the program entry-point) and an initial state. Program semantics are described using the semantics function semantics (see Figure 4.2). Although an in-depth explanation of semantics lies outside the scope of this thesis, we note that it is defined in terms of the evaluate function: the result of any application of evaluate on an expression which calls the entry-point of the program is denoted Fail if it results in an error other than a Rtimeout_error. In practice, this entails that any ill-typed program will evaluate to Fail.

4.2 Semantics preservation

In this section we will state and prove a theorem which allows us to guarantee that the semantics of any BVI program with non-Fail semantics is preserved under the transformation presented in Chapter 3. In Section 4.2.1, we describe this theorem,

```
semantics init_ffi code start =
 let es = [Call 0 (Some start) [] None]
 in
  if
   ∃k e.
    fst (evaluate (es,[],initial_state init_ffi code k)) = Rerr e ∧
    e ≠ Rabort Rtimeout_error
  then
   Fail
  else
   case
    some res.
     ∃k s r outcome.
      evaluate (es,[],initial_state init_ffi code k) = (r,s) ∧
      (case (s.ffi.final_event,r) of
        (None,Rval v9) ⇒ outcome = Success
       | (None,Rerr v10) ⇒ F
       | (Some e,v3) ⇒ outcome = FFI_outcome e) ∧
      res = Terminate outcome s.ffi.io_events
   of
    None ⇒
     Diverge
      (build_lprefix_lub
        (image
          (λk.
            fromList
             (snd
               (evaluate
                 (es,[],initial_state init_ffi code k))).
             ffi.io_events) 𝒰(: num)))
  | Some res ⇒ res
```

Figure 4.2: The observable semantics for BVI programs.

and outline the steps required in order to prove it. In Section 4.2.2, we will state a number of complementary theorems, and describe how they are proved. In addition, a number of supporting lemmas will be described. Finally, we conclude with the discovery of some surprising drawbacks of the verification methodology employed in most parts of the CakeML compiler in Section 4.3.

4.2.1 Semantics of programs

Inclusion of the compiler pass implemented in Chapter 3 into the BVI stage of the CakeML compiler requires a semantics verification proof stated in terms of the semantics function (see Figure 4.2). We state and prove the following theorem for the function compile_prog which applies our transformation to a BVI program.

Theorem 1. Program semantics are preserved under the transformation compile_prog.

$$
\begin{aligned}
\vdash\ & \mathsf{every}\ (\mathsf{odd\_names\_free}\ n \circ \mathsf{fst})\ \mathit{prog} \wedge \\
& \mathsf{all\_distinct}\ (\mathsf{map}\ \mathsf{fst}\ \mathit{prog}) \wedge \\
& \mathsf{snd}\ (\mathsf{compile\_prog}\ n\ \mathit{prog}) = \mathit{prog}_2 \wedge \\
& \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog})\ \mathit{start} \neq \mathsf{Fail} \Rightarrow \\
& \quad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog})\ \mathit{start} = \\
& \quad\quad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog}_2)\ \mathit{start}
\end{aligned}
$$

Additionally, we condition our theorem on two assumptions that are guaranteed to hold: all_distinct ensures that all addresses in the code table *prog* are distinct (cf. Section 3.2), and every (odd_names_free $n \circ$ fst) ensures that any odd address exceeding the address n is guaranteed to be free in the code table.

Theorem 1 bears close resemblance to most other top-level semantics theorems in the CakeML compiler – in fact, so does its proof. Unlike theorems involving evaluate, theorems like Theorem 1 are not proven by induction; instead the definition of semantics is unfolded, and the remaining goals are proven by repeated applications of lemmas which talk about expression semantics. In the case of Theorem 1 we will make use of one such lemma, which will be described in the upcoming section. The subsequent proof of this lemma, however, is both involved and interesting to such a degree that we dedicate the remainder of this chapter to this goal.

4.2.2 Semantics of expressions

The proof of Theorem 1 requires a theorem which states that the semantics of the expression pointed to by the *start* address in any valid BVI program *prog* are preserved under the compile_prog transformation.

Theorem 2. The semantics of the entry-point expression at address *start* is preserved under compile_prog.

$$
\begin{array}{l}
\vdash \mathsf{every}\ (\mathsf{odd\_names\_free}\ n \circ \mathsf{fst})\ prog \wedge \\
\quad \mathsf{all\_distinct}\ (\mathsf{map}\ \mathsf{fst}\ prog) \wedge \\
\quad \mathsf{evaluate} \\
\quad\ (([\mathsf{Call}\ 0\ (\mathsf{Some}\ start)\ []\ \mathsf{None}],[], \\
\quad\ \mathsf{initial\_state}\ ffi_0\ (\mathsf{fromAList}\ prog)\ k) = (r,s) \wedge \\
\quad 0 < k \wedge r \neq \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error}) \Rightarrow \\
\quad\ \exists\, ck\ s_2. \\
\quad\ \mathsf{evaluate} \\
\quad\ (([\mathsf{Call}\ 0\ (\mathsf{Some}\ start)\ []\ \mathsf{None}],[], \\
\quad\ \mathsf{initial\_state}\ ffi_0 \\
\quad\ (\mathsf{fromAList}\ (\mathsf{snd}\ (\mathsf{compile\_prog}\ n\ prog))) \\
\quad\ (k + ck)) = (r,s_2) \wedge \\
\quad\ \mathsf{state\_rel}\ s\ s_2
\end{array}
$$

Theorem 2 does not adhere to the general style of evaluate-theorems discussed in Section 4.1.2. First, the consequent of the implication contains an existential quantifier, suggesting that we need to provide two witnesses ck and s_2. Here, ck is a potential (finite) increment to the state clock, and s_2 is any post-state related to the post-state s of the non-transformed program. The former is needed since a compiler transformation may potentially introduce additional function calls, which would consume additional clock ticks (although we need no such ticks here, the quantifier is kept to ensure consistency with the definition of semantics). The relation state_rel is defined as follows:

$$
\begin{array}{l}
\mathsf{state\_rel}\ s\ t \iff \\
\quad s.\mathsf{ffi} = t.\mathsf{ffi} \wedge \\
\quad s.\mathsf{clock} = t.\mathsf{clock} \wedge \\
\quad \mathsf{code\_rel}\ s.\mathsf{code}\ t.\mathsf{code}
\end{array}
$$

Here, we require that both compiler states have corresponding FFI states, and that their clocks correspond. Additionally, we require that the code tables are related under a relation code_rel – any requirement that they are the same would fail for the simple reason that compile_prog modifies the code table. The definition of code_rel is shown in Figure 4.3.

The purpose of code_rel is to simplify the statement and subsequent proof of theorems involving evaluate by not explicitly mentioning compile_prog. Instead, it is explicit about where expressions are located before and after transformation of the code table. This relation between code tables turns out to be sufficient for all subsequent proofs of theorems regarding expression semantics. The relation

```
code_rel c1 c2 ⟺
 ∀ name args exp op.
  lookup name c1 = Some (args,exp) ⇒
   (check_exp name exp = None ⇒
     lookup name c2 = Some (args,exp)) ∧
   (check_exp name exp = Some op ⇒
     ∃ n.
      ∀ exp_aux exp_opt.
       compile_exp n name args exp =
        Some (exp_aux,exp_opt) ⇒
        lookup name c2 =
         Some (args,exp_aux) ∧
        lookup n c2 =
         Some (args + 1,exp_opt))
```

Figure 4.3: The definition of the relation code_rel.

defines the results of a lookup into the code table in terms of the results of the static check check_exp (see Chapter 3.4.2). Any expression in c_1 which does not pass the check should appear untouched in c_2. For any expression in c_1 that passes the check, an optimized expression and an auxiliary definition should be present in c_2. In order to use code_rel in place of compile_prog in our theorems we state and prove the following lemma.

Lemma 1. Any BVI programs *prog* and compile_prog n *prog* are related under code_rel.

```
⊢ all_distinct (map fst prog) ∧ every (odd_names_free n ∘ fst) prog ∧
  compile_prog n prog = (n1,prog2) ⇒
   code_rel (fromAList prog) (fromAList prog2)
```

The lemma is proved by recursive induction on the cases of compile_prog.

4.2.3 Generalized semantics of expressions

So far, we have only presented one theorem which explicitly relates the semantics of expressions to their transformed counterparts under compile_prog, namely Theorem 2. Apart from the previously mentioned existential quantifiers present in its consequent, there is another key issue with Theorem 2: it is stated in terms

of a single expression. Recall from Section 4.1.1 that evaluate is defined on lists of expressions. This necessarily implies that we are unable to prove any theorem in which evaluate is applied to a singleton list by recursive induction. Instead, we will generalize Theorem 2 to reason about *any* list of expressions.

Theorem 3. The semantics of any BVI expression evaluated in an environment env_1 and a state s is preserved under the transformation rewrite_tail, if the transformed expression is evaluated in an environment env_2 related to env_1 through env_rel, and a state with the code table c related to s.code through code_rel.

$$
\begin{array}{l}
\vdash \mathsf{evaluate}\ (xs,env_1,s) = (r,t) \wedge \\
\quad \mathsf{env\_rel}\ transformed\ acc\ env_1\ env_2 \wedge \\
\quad \mathsf{code\_rel}\ s.\mathsf{code}\ c \wedge \\
\quad (transformed \Rightarrow \mathsf{length}\ xs = 1) \wedge \\
\quad r \neq \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error}) \Rightarrow \\
\quad\ \mathsf{evaluate}\ (xs,env_2,s\ \mathsf{with\ code}\ :=\ c) = \\
\quad\quad (r,t\ \mathsf{with\ code}\ :=\ c) \wedge \\
\quad\ (transformed \Rightarrow \\
\quad\quad \forall\ op\ n\ exp\ arity. \\
\quad\quad\ \mathsf{lookup}\ nm\ s.\mathsf{code} = \mathsf{Some}\ (arity,exp) \wedge \\
\quad\quad\ \mathsf{optimized\_code}\ nm\ arity\ exp\ n\ c\ op \wedge \\
\quad\quad\ (\exists\ op'\ p. \\
\quad\quad\quad \mathsf{check\_exp}\ nm\ (\mathsf{hd}\ xs) = \mathsf{Some}\ (p,op') \wedge \\
\quad\quad\quad op' \neq \mathsf{Noop}) \Rightarrow \\
\quad\quad\ \mathsf{let}\ (p,x)\ =\ \mathsf{rewrite\_tail}\ n\ op\ nm\ acc\ (\mathsf{hd}\ xs) \\
\quad\quad\ \mathsf{in} \\
\quad\quad\quad \mathsf{evaluate}\ ([x],env_2,s\ \mathsf{with\ code}\ :=\ c) = \\
\quad\quad\quad\ \mathsf{evaluate} \\
\quad\quad\quad\quad ([\mathsf{apply\_op}\ op\ (\mathsf{hd}\ xs)\ (\mathsf{Var}\ acc)],env_2, \\
\quad\quad\quad\quad\ s\ \mathsf{with\ code}\ :=\ c))
\end{array}
$$

The first few lines of Theorem 3 are quite common for any verification of an optimization in the CakeML compiler. Outside of the relation code_rel, we also introduce a relation env_rel between environments. Here, $env_1 \preccurlyeq env_2$ denotes that env_1 is a prefix of env_2.

$$
\begin{array}{l}
\mathsf{env\_rel}\ transformed\ acc\ env_1\ env_2 \iff \\
\quad env_1 \preccurlyeq env_2 \wedge \\
\quad (transformed \Rightarrow \\
\quad\quad \mathsf{length}\ env_1 = acc \wedge \mathsf{length}\ env_2 > acc \wedge \\
\quad\quad \exists k.\ \mathsf{el}\ acc\ env_2 = \mathsf{Number}\ k)
\end{array}
$$

The purpose of env_rel is to *weaken* the requirements of Theorem 3: any expression that exists in a transformed program is likely to involve an additional accumulating

variable. Hence, we require that whenever env_rel $p\ a\ env_1\ env_2$ holds, env_1 is a prefix of env_2, and the accumulating variable points at index a of env_2, at which an integer is required to reside.

What makes Theorem 3 unusual is the boolean variable *transformed* together with 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, as well as requiring that any expression that passes check_exp (see Section 3.4.2) will – when transformed by rewrite_tail – evaluate to a result that is equal to the result of simply applying the expression to the accumulator variable under the operation (cf. Section 2.3.2). In particular, the conjuncts

$$\mathsf{lookup}\ nm\ s.\mathsf{code} = \mathsf{Some}\ (arity, exp)$$

and

$$\mathsf{optimized\_code}\ nm\ arity\ exp\ n\ c\ op$$

together with the definition of optimized_code (see below) ensures that *any* expression present in the original program and for which check_exp succeeded was transformed using compile_exp.

$$\begin{aligned}&\mathsf{optimized\_code}\ name\ arity\ exp\ n\ c\ op \iff\\ &\quad \exists\, exp\_aux\ exp\_opt\ p.\\ &\qquad \mathsf{compile\_exp}\ n\ name\ arity\ exp = \mathsf{Some}\ (exp\_aux, exp\_opt) \wedge\\ &\qquad \mathsf{check\_exp}\ name\ exp = \mathsf{Some}\ (p, op) \wedge\\ &\qquad \mathsf{lookup}\ name\ c = \mathsf{Some}\ (arity, exp\_aux) \wedge\\ &\qquad \mathsf{lookup}\ n\ c = \mathsf{Some}\ (arity + 1, exp\_opt)\end{aligned}$$

Finally, Theorem 3 is proven by strong induction induction on a well-founded relation which is ordered on the size of expressions and the number of remaining ticks in the compiler clock. The reason for which we cannot apply recursive induction is related to the step of our compiler transformation which replaces an Op expression with a tail call to a fresh function definition. Since this tail call is not a sub-expression to the original Op expression, our induction hypothesis would not be applicable, should we attempt a proof based on recursive induction on the cases of evaluate. The proof of Theorem 3 accounts for the bulk of our efforts, as the process is made cumbersome by the stronger induction. Moreover, it requires a series of supporting theorems for ensuring the correctness of various parts of the implementation from Chapter 3. The most important of these theorems are described in Section 4.2.4.

4.2.4 Supporting theorems

We conclude the verification of semantics preservation by describing the most important supporting theorems required for the proof of Theorem 3. Recall the implementation of rewrite_tail (Figure 3.4). The correctness of rewrite_tail relies on the following:

(i) Expression semantics is preserved when transformed by rewrite_op (see Figure 3.5).

(ii) Expressions that satisfy is_ok_type return an integer value when evaluation is successful.

In addition to the above, we are required to eliminate the possibility of reaching the dummy_case cases present in some of the definitions from Chapter 3 (see Figure 3.3 and both definitions of mk_tailcall in Section 3.3.2). The dummy_case in the final definition of mk_tailcall is eliminated by the following lemma for rewrite_op, which ensures that its result is always a binary operation upon success.

Lemma 2. If rewrite_op has transformed an expression then its result is always a binary operation.

$$\vdash \mathsf{rewrite\_op}\ iop\ name\ (\mathsf{Op}\ op\ xs) = (\mathsf{T}, exp) \Rightarrow \\ \exists\, e_1\ e_2.\ \mathsf{get\_bin\_args}\ exp = \mathsf{Some}\ (e_1, e_2)$$

The second dummy_case is present in the definition of push_call (see Figure 3.3) and occurs when the args_from function is called on an expression which is not a Call. The following lemma ensures that this is never the case.

Lemma 3. If rewrite_op has transformed an expression then the left operand of the resulting expression is always a recursive call.

$$\vdash iop \neq \mathsf{Noop} \wedge \\ \mathsf{rewrite\_op}\ iop\ name\ (\mathsf{Op}\ op\ xs) = (\mathsf{T}, \mathsf{Op}\ op\ [e_1;\ e_2]) \Rightarrow \\ \mathsf{is\_rec}\ name\ e_1$$

Both Lemmas 2 and 3 are proven directly in a similar style, by applying a simplification tactic which rewrites the proof goal using the definition of rewrite_op, followed by exhaustive treatment of its resulting cases.

We now turn our attention to the main correctness theorem for rewrite_op, which acts to ensure that it is semantics preserving.

Theorem 4. The transformation rewrite_op is semantics preserving.

$$\begin{array}{l}\vdash \mathsf{evaluate}\ ([exp], env, s) = (r, t) \wedge \\ \quad r \neq \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error}) \wedge \\ \quad \mathsf{rewrite\_op}\ op\ name\ exp = (p, exp_2) \wedge \\ \quad \mathsf{evaluate}\ ([exp], env, s) = (r, t) \Rightarrow \\ \quad\ \mathsf{if}\ \neg p\ \mathsf{then}\ exp_2 = exp \\ \quad\ \mathsf{else}\ \mathsf{evaluate}\ ([exp_2], env, s) = (r, t)\end{array}$$

Although Theorem 4 is concisely stated, its proof is unusually involved. The reasons for this is that it exploits both the associative and commutative properties of integer addition and multiplication. Moreover, the definition of rewrite_op contains a large number of cases. Several of these implicitly make use of commutativity and associativity more than once, requiring multiple instantiations of associativity- and commutativity lemmas. For reasons of brevity, we refrain from including these lemmas here. In addition, the properties of multiplication and addition are already well established. The proof of Theorem 4 is performed by recursive induction on its cases.

We conclude this section with a theorem stating that any expression satisfying the check is_ok_type (see Section 3.4.1) evaluates to an integer.

Theorem 5. Whenever the evaluation of an expression satisfying is_ok_type results in value this value is an integer.

$$\begin{array}{l}\vdash \mathsf{is\_ok\_type}\ exp \wedge \\ \quad \mathsf{evaluate}\ ([exp], env, s) = (\mathsf{Rval}\ r, t) \Rightarrow \\ \quad\ \exists\, n.\ r = [\mathsf{Number}\ n]\end{array}$$

In contrast to all other theorems treated in this chapter so far, the proof of Theorem 5 is performed by induction over the datatype which defines the abstract syntax of BVI expressions (see Figure 3.1 in Section 3.1).

Note on omitted theorems. The entire suite of theorems required for proving the correctness of the implementation of the optimization described in this report is larger than what is presented here. For reasons of brevity, the theorems and lemmas deemed most important were selected for inclusion. For a complete listing of all theorems, as well as the mechanized proofs in all their glory, we refer the reader to the proof theories located in the CakeML GitHub repository:

https://github.com/CakeML/cakeml/blob/master/compiler/backend/proofs/bvi_tailrecProofScript.sml

4.3 Limitations

We return briefly to the transformation described in Chapter 2, and its implementation described in Chapter 3. It should be clear by now that the transformation and its HOL implementation is correct – in fact, we have dedicated this chapter towards giving a formal proof of this claim. However, there are two separate issues that arise when proving the correctness of the transformation, both of which impose restrictions on the efficiency of the transformation.

4.3.1 The lack of a type system

Although it may not be immediately clear, the implementation described in Chapter 3 changes the order of evaluation for transformed expression. Since CakeML is an impure language, expressions may incur side effects, such as I/O events, when evaluated. Hence, changing the order in which expressions are evaluated may potentially result in I/O events appearing out of order, meaning that the observable behavior of a program is changed. Clearly, this is not something we can allow.

The culprit is Rule 3.1 (Section 3.3.2), which swaps the order in which some of the involved expressions are evaluated. To see that this is the case, let f and f' be BVI functions defined such that

$$f\ x + y = f'\ x\ y\ . \tag{4.1}$$

Expressions are evaluated from left to right, leading to the following orders of evaluations for the respective sides in (4.1).

$f\ x + y$	$f'\ x\ y$
x	x
body of f	y
y	body of f'
$\ldots$	$\ldots$

In this situation, the expression $f\ x$ is expected to evaluate to $f'\ x\ 0$. Evaluation of the function bodies at both sides of (4.1) will therefore end up in the same expression. However, the order in which the expression y and the function bodies are evaluated differ from both sides.

A situation closely resembling the one above exposes itself during the proof of Theorem 3; in particular, when proving that the transformation operation Op *op xs* (corresponding to $f\ x + y$) can be transformed to a tail call (corresponding to $f'\ x\ y$). The assumptions of Theorem 3 only guarantee the evaluation of the operation $f\ x + y$ does not result in a Rtype_error. However, this does not rule out the case that $f\ x$ diverges, for instance. In this case, evaluation of $f\ x + y$ will not

result in a type-error, and it will not do so even if the evaluation of y itself results in a type-error! However, in this case, the right-hand side of (4.1) would result in a type-error, and we are left unable to close the proof goal – it is simply not valid.

Clearly, to ensure that the above situation is avoided, we must be able to guarantee that the evaluation of y always result in a concrete value. What's more, we must be able to guarantee that this value is of the correct type. Simply restricting y to be a pure expression and preventing it from accessing global state in any way is not enough, as even a variable expression may evaluate to a Rtype_error (see Figure 4.1).

The current solution to these issues is to enforce strong restrictions on the expression y. In particular, this forces us to disallow variables. However, in a well-typed program, all variables are bound to a correctly typed value in the environment. Hence, the addition of a type system to the BVI language would alleviate the issues described above, as we could simply enforce purity on y and refrain from applying the transformation to ill-typed programs.

The inclusion of a type system in the BVI stage of the compiler is non-trivial, however: it would rely on all prior ILs having type systems. What's more, we would likely have to prove the preservation of soundness at the transition between each pair of type systems. Instead, we suggest a less powerful solution that nevertheless allow us to correctly deduce the type of variables and ensure that they are bound. This would, at the very least, enable the optimization to correctly transform all examples presented in this report. To each value in the environment we assign an element in a *context*, which tells us

(i) whether or not a variable is bound,

(ii) the expected type of a variable, if it is bound.

The above information is collected by recursively traversing BVI expressions in an analysis pass, decorating the context as we encounter expressions that reveal type restrictions on variables (i.e. comparison with a literal, etc.). If we are unable to fully deduce the types of all expressions in tail position, we abstain from transforming a function.

The above approach has the pleasant side-effect that it alleviates the need of the is_ok_type check defined in Section 3.4.1. However, due to timing constraints, we leave the implementation of type approximation using contexts as future work.

4.3.2 The compiler clock

Recall the discussion on the production of auxiliary functions from Section 3.5. In the implementation of compile_exp (see Figure 3.8), auxiliary definitions were

created using a function let_wrap which simply duplicated the optimized function.

```
let_wrap num_args id exp =
  Let (genlist (λ i. Var i) num_args ++ [id]) exp
```

Clearly, we could have created an auxiliary function by simply performing a function call to the optimized function. The reason that we avoid doing so is related to the discussion on the compiler clock from Section 4.1.3.

```
mk_aux_call name num_args id =
  Call 0 (Some name) (id::genlist (λ i. Var i) num_args) None
```

Introducing auxiliary calls such as in mk_aux_call implies that any function call in the optimized code table are subject to additional indirection; we must potentially perform two function calls where we previously performed one. This has the effect that the compiler clock will need to be increased in the proofs in the way described in Section 4.1.3. In particular, this greatly complicates the proof of Theorem 3.

For the reasons stated here, we leave the proof of an implementation using mk_aux_call over let_wrap for auxiliary function creation as future work. However, we believe that it is possible to include the former in the implementation by first proving instances of the more complicated theorems (such as Theorem 3) that utilize the current version of compile_exp (with let_wrap). In addition to these one would provide a *separate* theorem stating that compile_exp would have preserved semantics with mk_aux_call, so long as the compiler clock is increased. In this way, reasoning about the compiler clock is limited to a simpler theorem.

5

Related Work

Previous work has been done on verified optimized compilers. The CompCert [11] compiler targets the C programming language, and is implemented mainly in the Coq proof assistant [12]. It is notable for showing the feasibility of formally verifying a modern compiler. Examples in the functional programming domain include the Cogent [13] compiler, which produces verified C code from a functional programming language aimed at systems programming, but for this reason lacks features otherwise common to general purpose language implementations, such as automatic memory management. The CakeML [6, 3] compiler is notable since the CakeML language includes a rich subset of Standard ML with many modern language features. Moreover, the compiler is able to bootstrap itself and produces verified machine code for several hardware targets, making it the most realistic verified optimizing compiler for a functional programming language to date [3].

Burstall and Darlington [14] 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 [5], with the primary goal of eliminating quadratic list append usage. Since the introduction of tail calls is our primary goal, we have settled with treating associative integer arithmetic, although it is possible to extend its application to list append. A different transformation for introducing accumulators is presented in Kühnemann, et al. [15]. It is, however, limited to unary functions. We are unaware of any compiler which implements this transformation.

Chitil [16] 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 2.3.1, but is limited to functions returning lists. As with Kühnemann et al. [15], 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 in-

creased confidence in its correctness, but shows the feasibility of implementing the transformation in practice and integrating it into a larger context.

6

Conclusions and future work

In this thesis, we have described and implemented an optimizing compiler transformation in HOL4 acting on expressions in the BVI intermediate language. The transformation introduces tail recursion in certain recursive functions on the integers, while preserving observational semantics.

The implementation has been integrated with the existing structure of the CakeML compiler as a standalone compiler stage. Moreover, we have verified the compiler stage to preserve the observational semantics of the program under transformation. To the best of our knowledge, this is the first fully verified implementation of this transformation in any modern compiler. In addition, our contributions make the CakeML compiler the first fully verified compiler which performs this transformation.

During the course of verifying the transformation, our efforts uncover surprising drawbacks in some of the verification techniques currently employed in the BVI compiler phase of the CakeML compiler. In particular, since BVI lacks a type system, we are left unable to infer the types of variables. This leads to an implementation that is weaker than the one originally envisioned, since the class of expressions handled by the transformation is limited to tail expressions that consist of an operation composed only by a single recursive call and integer literals.

Although the transformation handles *some* expressions, we feel that the current solution is unnecessarily restrictive. We therefore suggest an alternative approach to static analysis which would enable us to infer the types of variables in some expressions. In particular, we show that this class of expressions include the examples shown in Chapters 2 and 3. However, we also note that a solution which would lift these restrictions completely exists, namely giving BVI a type system. This approach, however, has the unwanted side-effect that all ILs prior to BVI would need a type system. In addition, soundness proofs for these type systems between each major stage of compilation would be required.

Due mainly to lack of time, implementations of the suggested improvements in Section 4.3 are left as future work. In addition, other topics suitable for future study includes the addition of support for list append in the transformation. The implementation from Chapter 3 can be easily extended to support append, as fewer

cases need to be considered due to the operation not being commutative. While the original intention was to include an implementation for append, the operation was left out since it is currently not primitive to BVI[1]. We conclude by noting that the benefits of transforming list-append are two-fold. They are made clear by the reverse-example from Section 2.3.1: while the original example performs a quadratic number of operations in proportion to the length of the input, the transformed example is linear in this regards.

[1]The inclusion of list append as a primitive operation in the BVI language is under consideration.

Bibliography

[1] R. Milner, *The Definition of Standard ML: Revised*. MIT press, 1997.

[2] K. Slind and M. Norrish, "A brief overview of HOL4," *Theorem Proving in Higher Order Logics*, pp. 28–32, 2008.

[3] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "A new verified compiler backend for CakeML," 2016.

[4] S. Owens, M. Norrish, R. Kumar, M. O. Myreen, and Y. K. Tan, "Verifying efficient function calls in CakeML," in *ICFP '17: Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming*, ACM Press, Sept. 2017. To appear.

[5] P. Wadler, "The concatenate vanishes," *Note, University of Glasgow*, 1987.

[6] R. Kumar, M. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," pp. 179–191, ACM, 2014.

[7] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan, "Functional big-step semantics," in *European Symposium on Programming Languages and Systems*, pp. 589–615, Springer, 2016.

[8] M. O. Myreen and S. Owens, "Proof-producing synthesis of ML from higher-order logic," in *ACM SIGPLAN Notices*, vol. 47, pp. 115–126, ACM, 2012.

[9] A. Guéneau, M. O. Myreen, R. Kumar, and M. Norrish, "Verified characteristic formulae for CakeML," in *European Symposium on Programming*, pp. 584–610, Springer, 2017.

[10] Y. K. Tan, S. Owens, and R. Kumar, "A verified type system for CakeML," in *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, p. 7, ACM, 2015.

[11] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, p. 363, 2009.

[12] "The Coq proof assistant." `https://coq.inria.fr/`. Accessed 2017-04-13.

[13] L. O'Connor, C. Rizkallah, Z. Chen, S. Amani, J. Lim, Y. Nagashima, T. Sewell, A. Hixon, G. Keller, T. Murray, *et al.*, "COGENT: Certified compilation for a functional systems language," *arXiv preprint arXiv:1601.05520*, 2016.

[14] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 44–67, 1977.

[15] A. Kühnemann, R. Glück, and K. Kakehi, "Relating accumulative and non-accumulative functional programs," in *International Conference on Rewriting Techniques and Applications*, pp. 154–168, Springer, 2001.

[16] O. Chitil, "Type-inference based short cut deforestation (nearly) without inlining," in *Symposium on Implementation and Application of Functional Languages*, pp. 19–35, Springer, 1999.