CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Tail Recursion Modulo Cons Optimization in CakeML

Master's Thesis

RY WIESE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

MASTER'S THESIS 2026

Tail Recursion Modulo Cons Optimization in CakeML

RY WIESE

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Tail Recursion Modulo Cons Optimization in CakeML
RY WIESE

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: Andreas Abel, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Tail Recursion Modulo Cons Optimization in CakeML

RY WIESE
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this project, the Tail Recursion Modulo Cons (TMC) optimization is implemented in the end-to-end verified CakeML compiler. The optimization makes tail recursive those functions which contain a recursive call nested inside one or more constructors. The TMC transformation is applied in the bytecode-value intermediate language (BVI) and rewrites functions in a way that cannot be done manually in CakeML source code.
The syntax and semantics of BVI are extended with three new language constructs, and a new compiler phase is added to rewrite BVI programs using the new constructs. The compiler is implemented and formally verified in the HOL4 interactive theorem prover, and included in the project is a mechanized proof that the transformation produces code that is semantically equivalent to the original.

Keywords: Computer Science, Formal Verification, Verification, Compilers, Optimization, CakeML, HOL4, End-To-End Verification

Acknowledgements

I would like to thank my supervisor, Magnus Myreen, for his incredible guidance and support during this thesis. I would also like to thank my examiner, Andreas Abel, as well as my opponent and the HOL4 community for the lively discussions. Lastly, I would like to thank the new friends who have made me feel at home in Sweden, and my friends and family back home who have supported me from abroad.

RY WIESE, Gothenburg, June 2026

Contents

List of Figures

1

Introduction

CakeML [Kumar et al., 2014] is a functional programming language with an end-to-end verified compiler. The compiler is implemented using the HOL4 interactive theorem prover, and alongside the implementation itself lives a proof that it is correct. This correctness proof is machine checked, so CakeML users are guaranteed that the executables generated for their programs will not contain any compiler-introduced bugs and will behave exactly as programmed. Moreover, it guarantees that extending and optimizing the compiler will not introduce any new bugs, so long as the correctness proof is updated and accepted by HOL4.

In this project I implement one such optimization, namely the Tail Recursion Modulo Cons (TMC) optimization [Risch, 1973, Friedman and Wise, 1975, Leijen and Lorenzen, 2023], and formally verify that the correctness of the compiler is preserved. The optimization automatically rewrites certain classes of functions to be more space efficient and to avoid exhausting stack space. It is applied using new constructs added to an intermediate language in a lower-level phase of the compiler and rewrites the function in a way that a user could not have done manually in CakeML source code. Not only does this optimization have tangible performance benefits for CakeML users, it is also the first time, to the best of my knowledge, that such an optimization has been implemented and proven correct in an end-to-end verified compiler.

1.1 The TMC Transformation

Here, I will describe an example of the TMC transformation in more detail. Consider the following CakeML function for appending two lists, where I use `Nil` and `Cons` instead of the standard list constructors `[]` and `::`.

```
fun append Nil             ys = ys
  | append (Cons (x, xs)) ys = Cons (x, append xs ys)
```

Figure 1.1: Unoptimized list `append` function.

This function is not tail recursive, as `Cons` is applied after the recursive call to `append`. To make it tail recursive, one can add an auxiliary continuation argument `k`:

```
fun append xs ys = append' id xs ys

and append' (k : 'a list -> 'a list) Nil            ys = k ys
  | append' (k : 'a list -> 'a list) (Cons (x, xs)) ys =
      append' (k o fn l => Cons (x, l)) xs ys
```

Figure 1.2: Continuation-passing style list `append` function.

Now, application of the constructor has been moved into the argument, and the function body is only the recursive call. In this form, the function can be optimized with tail call elimination to achieve constant space complexity on the stack. However, each recursive call allocates a new lambda function, resulting in linear space complexity on the heap.

To understand the problem, it is helpful to visualize how the CakeML compiler evaluates constructors. Each constructor allocates an immutable Block. In the case of list, each Block contains an element and the next Block. An example is shown in Figure 1.3 below:

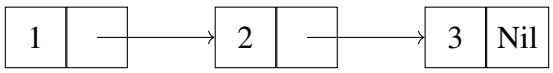

Figure 1.3: List `1::2::3::Nil` as immutable Blocks.

Such a list is constructed back-to-front. To create a Block (using `Cons`), the rotation of the list must already be defined. This constraint is the source of the inefficiency. In the unoptimized `append` (Figure 1.1), there is a `Cons` after each recursive call. With the continuation-passing style optimization (Figure 1.2), there is a `Cons` in each lambda abstraction. In both cases, we must track an unapplied constructor for each list element, all applied only once recursion bottoms out.

With the TMC optimization, we will instead build this list front-to-back, with each call only updating a pointer and allocating no memory other than the new list element itself. This can be achieved by introducing a mutable MutBlock construct, in the style of Minamide's data structures with holes [Minamide, 1998]. A MutBlock resembles a Block, but holds a pointer to the next MutBlock which can be uninitialized (the hole). Holes can be filled by modifying the pointer to point at the next MutBlock once it is allocated. This enables the construction of lists from front-to-back, where the beginning of a list is fully constructed before the rest has been filled in.

For example, consider the unfinished MutBlock `1::2::_`, which contains a hole in the second MutBlock:

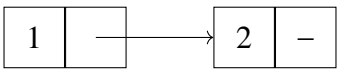

Figure 1.4: `1::2::_` as MutBlocks with holes.

The MutBlock `1::2::3::_` can be constructed by "filling" the hole: modifying it to point

to a newly allocated MutBlock containing 3. Unlike the Blocks in Figure 1.3, the new MutBlock has a hole which can be filled in later if more elements are added.

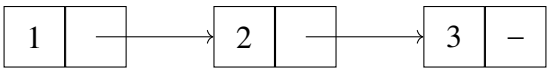

Figure 1.5: `1::2::3::_` as MutBlocks with holes.

The TMC transformation produces an `append` function that rebuilds `xs` using MutBlocks, fills the last hole with `ys` in the base case, and finalizes the result into regular Blocks.

```
fun append Nil            ys = ys
  | append (Cons (x, xs)) ys =
      let p = MutCons <x, _> in
      let _ = append' xs ys p 1 in
      FinalizeCons p

and append' Nil            ys (k : ptr) (i : int) =
      UpdateCons k i ys
  | append' (Cons (x, xs)) ys (k : ptr) (i : int) =
      let p = MutCons <x, _> in
      let _ = UpdateCons k i p in
      append' xs ys p 1
```

Figure 1.6: TMC list `append` function using MutBlocks with holes.

It also produces an auxiliary *worker* function `append'`, which takes two additional arguments and performs the core work of the recursion. The auxiliary argument `k` is a pointer to the previous MutBlock, and the argument `i` is the index of the hole in the MutBlock payload. Each iteration allocates a new MutBlock for the current element, fills the previous hole to point to the new MutBlock, and passes the new pointer and index as arguments to the recursive call.

Note that the pseudocode above relies on a few new language constructs for interacting with MutBlocks, including:

- `MutCons <`x_1`, ..., `x_{i-1}`, _, `x_{i+1}`, ..., `x_n`>` which returns a pointer to a newly allocated MutBlock with a payload that includes a hole at index `i`.
- `UpdateCons k i v` to fill the hole at index `i` of the MutBlock at address `k` with value `v`.
- `FinalizeCons p` to recursively reinterpret the mutable MutBlocks as standard Blocks.

These constructs are formalized more concretely in Chapter 2. It is worth noting that they are not implemented in the CakeML language itself, but rather in intermediate languages internal to the compiler. As mentioned earlier, the transformed function is one that a programmer cannot express in CakeML directly.

The `append` function is relatively simple as it contains a recursive call inside a single constructor in tail position. The TMC optimization I have implemented is more powerful, and can rewrite functions which contain a recursive call nested in *many* constructors. It can also optimize functions with a recursive call in another tail position that is *not* wrapped in a constructor. The transformation can be applied to functions which contain multiple recursive calls in the same constructor in tail position, but only *one* of them can be made tail recursive. More examples are provided in Chapter 5. In particular, the binary tree `insert` example in Section 5.4, illustrates that the auxiliary index argument is not always fixed, as it is in the `append` example.

1.2 Context

Abrahamsson and Myreen present a transformation for automatically introducing tail recursion in CakeML using an accumulator variable whose type matches the return type of the function [2017]. This transformation can be applied to functions where the operation in tail position is associative and has an identity, enabling accumulation. The TMC transformation I have implemented introduces tail recursion for a different class of recursive functions: namely, those which contain a constructor in tail position.

Leijen and Lorenzen have implemented the TMC optimization in the Koka programming language, a language with algebraic effect handlers [2023]. Their implementation served as an inspiration for this project, including the use of data structures with holes [Minamide, 1998]. This project, however, expands on their work by formally verifying the correctness of the transformation and by proving that the correctness of the CakeML compiler is maintained. To the best of my knowledge, this optimization has not been carried out in a verified compiler before.

Allain et al. have implemented the optimization in the OCaml compiler, and as part of their work, they formalized and verified their transformation in the Rocq interactive theorem prover [2025]. However, the OCaml compiler itself is not verified. Their verification was carried out separately from the compiler implementation, on a small, untyped calculus. In this project, I formally verify the transformation implementation within the CakeML compiler itself, and I prove that the correctness of the compiler is maintained.

1.3 Notation

Throughout this report:

- code constructors are in typewriter font: `Cons`, `MutCons`.
- semantic value constructors are in sans-serif font: Block, MutBlock.
- variables are italicized: xs, e.
- components of the semantic environment and state are given Greek letters, such as: ρ, ω, μ.
- for a list l:

  - The ith element is given as: $l[i]$.
  - Replacing the ith element with the value v is given as: $l[i \mapsto v]$.
- for a finite map μ:
  - the value at key k is given as: $\mu[k]$.
  - insertion of key k with value v is given as: $\mu\langle k \mapsto v\rangle$.
  - the set of keys in μ is given as: $dom(\mu)$.
  - the set of values in μ is given as: $range(\mu)$.
- for an expression e, the expression e^{+n} is e with every de Bruijn index shifted by n to account for new binders.

1.4 Methodology

The CakeML compiler is implemented and verified in the HOL4 interactive theorem prover [Slind and Norrish, 2008]. Alongside the implementation lives a number of correctness theorems, which are composed to prove a correctness theorem that the compiler only produces results that are permitted by the semantics of CakeML [Kumar et al., 2014]. Each component implemented during this project includes a proof that the implementation is correct and that overall correctness is maintained.

The transformation is applied in the bytecode-value intermediate language, or BVI, of the compiler. The pseudocode presented in Figure 1.6 uses three new operations: `MutCons`, `UpdateCons`, and `FinalizeCons`. Chapter 2 formalizes the extension of BVI syntax to support these new operations. Chapter 3 defines the semantics of these operations in terms of the new semantic value type MutBlock. Chapter 4 formalizes the transformation of unoptimized BVI into the optimized version, relying on the new operations. A few examples are provided in Chapter 5. Finally, Chapter 6 outlines the proof that the transformation produces a BVI expression that is semantically equivalent to the original.

1.5 Declared Use of AI

I used generative AI to assist with formatting LaTeX, to suggest conventional fonts and symbols for use in inference rules, and to discover related work (in particular, to look for other instances of TMC being carried out in a verified compiler). My supervisor helped me implement some of the more mundane proofs, and generative AI was used for some of them.

2

Extending BVI Syntax

In this chapter, I will provide an overview of the existing syntax of the bytecode-value intermediate language, or BVI, and then describe how it is extended in this project to support the TMC transformation. I will also provide some simple examples to illustrate how expressions are constructed along with a brief intuition of the semantics that will be introduced formally in Chapter 3. Finally, in Section 2.3, I provide the fully formalized BVI representation of the `append` example from Chapter 1.

2.1 BVI Expressions

Figure 2.1 outlines the syntax of BVI [Tan et al., 2019]. BVI is a low-level functional language with a code store and without closures. Its semantics, introduced in Chapter 3, include a clock to limit the number of execution steps a program can take. The `Tick` expression is used to decrement that clock. Local variables are indexed with de Bruijn indices. Each BVI function operates within only one stack frame, so new stack frame creation is allowed only using the `Call` construct. Exceptions can be raised, but there is no separate construct to handle them. Exception handling requires a fresh stack frame and is therefore performed using `Call`.

The `Call` constructor accepts four arguments: a number by which to decrement the clock in the semantics, an optional name of the code in the code store to be executed, the arguments, and an optional exception handler. If the name is `NONE`, it is instead represented by the final argument. The semantics of `Force` forces the evaluation, using the code at the provided name in the store, of the thunk in the variable with the provided de Bruijn index. The `Op` expression covers all operations supported in BVI, along with a list of arguments that the operation consumes. Most relevant here are *int_op*, *block_op*, and *mem_op*, which I will describe in the rest of this section.

2.1.1 Operations

Figure 2.1 shows the cases of *int_op*, including `Const` to express constants. The other operations, such as addition, multiplication, etc., accept their arguments as the *exp* list provided to the `Op` constructor from Figure 2.1. It is worth noting that HOL4 supports both the *int* type for integers and the *num* type for the natural numbers, defined using the Peano axioms.

```
exp  ::= Var idx
       | If exp exp exp
       | Let (exp list) exp
       | Raise exp
       | Tick exp
       | Call ticks
              (name option)
              (exp list)
              (exp option)
       | Force name idx
       | Op op (exp list)

idx     ::= num    de Bruijn index
ticks   ::= num    to decrement the clock
name    ::= num    of a function
tag     ::= num    of a constructor
el_idx  ::= num    of an element in a
                   constructor payload

op  ::= IntOp int_op
      | BlockOp block_op
      | MemOp mem_op
      | FFI mlstring
      | ...

block_op  ::= Cons tag
            | ElemAt el_idx
            | LenEq num
            | LengthBlock
            | ...

int_op  ::= Const int
          | Add
          | Sub
          | Mult
          | Div
          | Mod
          | ...

mem_op  ::= Ref
          | Update
          | El
          | ...
          | MutCons tag el_idx
          | UpdateCons
          | FinalizeCons
```

Figure 2.1: Abbreviated BVI syntax.

Operations of type *mem_op* enable allocation (`Ref`), mutation (`Update`), and lookup (`El`) of references and arrays stored in memory. These operations can mutate existing memory in the semantics, and it is here that the new constructs to interact with MutBlocks are added for this project.

Finally, operations of type *block_op*, some of which are shown in Figure 2.1, are expressions whose semantics involve construction and interaction with immutable, non-primitive data. The `Cons` constructor, in particular, represents a CakeML data constructor. Built-in constructors, such as unit, tuples, or list cons, as well as user-defined constructors are expressed this way. Each constructor is distinguished by a unique tag. For example, the expression `Op (BlockOp (Cons unit_tag)) []` evaluates to the Unit value.

As with *int_op*, the constructor arguments are passed as the final argument to the `Op` constructor. Note that, in the compiler implementation, these arguments are provided in reverse order; CakeML is evaluated right-to-left, while BVI is evaluated left-to-right. For clarity, the arguments are presented in their normal, non-reversed order in this report. The BVI expression which evaluates semantically to the list `1::2::3::Nil`, mirroring the diagram in Figure 1.3, is shown in Figure 2.2:

```
Op (BlockOp (Cons list_tag))
  [
    Op (IntOp (Const 1)) [];
    Op (BlockOp (Cons list_tag))
      [
        Op (IntOp (Const 2)) [];
        Op (BlockOp (Cons list_tag))
          [
            Op (IntOp (Const 3)) [];
            Op (BlockOp (Cons list_tag)) []
          ]
      ]
  ]
```

Figure 2.2: This `BlockOp` evaluates to `1::2::3::Nil` in the semantics.

Since operations of the form `Op (BlockOp (Cons` *tag*`))` *xs* represent constructors, it is these expressions that are targeted for optimization under the TMC transformation, which is described in more detail in Chapter 4.

2.2 New Constructs

The new constructs for MutBlocks are implemented as the new `MutCons`, `UpdateCons`, and `FinalizeCons` cases in *mem_op*.

2.2.1 MutCons

`MutCons` represents allocation of a MutBlock. It is similar to the `Cons` operation; `tag` is defined the same way, and the arguments also define the initial elements of the structure.

The only difference is that the element at provided index – the hole – can be later mutated in the semantics by the `UpdateCons` construct introduced in Section 2.2.2. For example, the construction of the singleton MutBlock `1::_` is represented using `MutCons` as below:

```
Op (MemOp (MutCons list_tag 1))
  [
    Op (IntOp (Const 1)) [];
    Op (IntOp (Const 0)) []
  ]
```

Figure 2.3: This `MutCons` evaluates to `1::_` in the semantics.

There are a few things to note in this example. First is that there is no `MutCons` expression representing an empty list. The empty list is represented here simply as the unfilled hole at index 1. In this example, and as a standard we have established in this project, the unfilled hole is given a default constant integer value of 0, expressed `Op (IntOp (Const 0)) []`. Moreover, there is no way to distinguish syntactically the difference between a filled and an unfilled hole; it is of course possible to have a filled hole with value 0. Lastly, it is possible to nest `MutCons` in a single expression, representing a MutBlock with all holes filled except for the last, as shown in the next example:

```
Op (MemOp (MutCons list_tag 1))
  [
    Op (IntOp (Const 1)) [];
    Op (MemOp (MutCons list_tag 1))
      [
        Op (IntOp (Const 2)) [];
        Op (IntOp (Const 0)) []
      ]
  ]
```

Figure 2.4: This `MutCons` evaluates to `1::2::_` in the semantics.

This example represents `1::2::_` in MutBlock form, as illustrated graphically in Figure 1.4. Here, the initial value of the outer hole is itself another MutBlock, and in this manner entire lists can be represented with a single expression containing only `MutCons`. In the TMC transformation, however, we want to construct lists (and other data structures) iteratively, by successfully filling previously constructed holes. This is achieved with the next construct: `UpdateCons`.

2.2.2 UpdateCons

Once allocated, the hole of a MutBlock can be mutated with the `UpdateCons` operation. `UpdateCons` itself takes no arguments, but it expects three arguments to its enclosing `Op`: the MutBlock to mutate, the index of the hole in that MutBlock, and the new value to write.

As an alternative to the example in Figure 2.4, which represents `1::2::_` using only `MutCons`, we can instead use `UpdateCons` to represent the filling of the hole in `1::_` with the singleton `2::_`. If the `MutCons` from Figure 2.3 representing `1::_` is bound with de Bruijn index `hb1`, then the variable `Var hb1` can be provided as the first argument to the `MutCons`. The following expression, when evaluated, will fill its hole to produce `1::2::_`:

```
Op (MemOp (UpdateCons))
  [
    Var hb1;
    Op (IntOp (Const 1)) [];
    Op (MemOp (MutCons list_tag 1))
      [
        Op (IntOp (Const 2)) [];
        Op (IntOp (Const 0)) []
      ]
  ]
```

Figure 2.5: This `UpdateCons` fills the hole to construct `1::2::_` in the semantics.

2.2.3 FinalizeCons

The `FinalizeCons` operation, in the semantics, recursively walks a tower of MutBlocks into a tower of Blocks. `FinalizeCons` takes no arguments, but expects a single argument to its enclosing `Op`: namely, the MutBlock being finalized. Once the MutBlock represented by Figure 2.5 is fully constructed, it can be finalized as in this example, again assuming that the top `MutCons` expression is bound at de Bruijn index `hb1`:

```
Op (MemOp (FinalizeCons)) [Var hb1]
```

Figure 2.6: This expression finalizes the MutBlock `1::2::_` into the list `1::2::Nil` in the semantics.

2.3 BVI Append

Figure 2.7 shows the full BVI syntax of the unoptimized `append` function from Figure 1.1. Variables `xs` and `ys` are represented as `Var 0` and `Var 1` respectively, and the pattern match on `xs` is an `If` statement whose condition tests whether the length of `xs` is zero. If it is not, we construct a new list using `Cons`, whose payload contains the first element in `xs` as well as the recursive call. The arguments to the recursive call are the tail of `xs` and `ys`. Here, `ts` and `l_append` are assumed to be bound to a suitable tick count and function name, respectively.

Figure 2.8 shows the BVI representation of the optimized wrapper function `append` from Figure 1.6. Here, the else branch is composed of two nested `Lets`. The first binds a `MutCons` analagous to the `Cons` from the previous example. Instead of a recursive call in

```
If
  (Op (BlockOp (LenEq 0)) [Var 0])
  (Var 1)
  (Op (BlockOp (Cons list_tag))
    [
      Op (BlockOp (ElemAt 0)) [Var 0];
      Call ts (SOME l_append)
        [
          Op (BlockOp (ElemAt 1)) [Var 0];
          Var 1
        ] NONE
    ])
```

Figure 2.7: The BVI syntax of the unoptimized `append` function.

the payload, there is a `Const 0` operation representing the unfilled hole. The second `Let` binds the call to the worker, assumed to have name `l_append'`. The first two arguments are the same as in the unoptimized version – although the de Bruijn indeces are shifted by one since they appear under a binder. The auxiliary arguments are also provided: `Var 0` referencing the `MemOp` in the binder, and the constant 1 representing the index of the hole. In the tail position of the second `Let` is the `FinalizeCons` operation, given a `Var` referencing the `MemOp` in the first binder.

Finally, Figure 2.9 shows the BVI of the worker function `append'` from Figure 1.6. The two auxiliary variables, the pointer and index of the hole, are provided as auxiliary arguments at de Bruijn indeces 2 and 3, respectively. These are provided to `UpdateCons` in the base case, along with `Var 1` – again representing `ys` – as the value to write into the hole. Like in the wrapper in Figure 2.8, the first `Let` binder is a `MutCons`. Unlike in the wrapper, there is an `UpdateCons` in the second `Let` binder, which fills the `old` hole at pointer `Var 3` and index `Var 4` (now shifted under the binder) with a pointer to the *new* `MutBlock` allocated by the `MutCons`. Finally, the recursive call appears in tail position.

```
If
  (Op (BlockOp (LenEq 0)) [Var 0])
  (Var 1)
  (Let
    [
      Op (MemOp (MutCons list_tag 1))
        [
          Op (BlockOp (ElemAt 0)) [Var 0];
          Op (IntOp (Const 0)) []
        ]
    ]
    (Let
      [
        Call ts (SOME l_append')
          [
            Op (BlockOp (ElemAt 1)) [Var 1];
            Var 2;
            Var 0;
            Op (IntOp (Const 1)) []
          ] NONE
      ]
      (Op (MemOp (FinalizeCons)) [Var 1])))
```

Figure 2.8: The BVI syntax of the optimized append wrapper function.

```
If
  (Op (BlockOp (LenEq 0)) [Var 0])
  (Op (MemOp (UpdateCons)) [Var 2; Var 3; Var 1])
  (Let
    [
      Op (MemOp (MutCons list_tag 1))
        [
          Op (BlockOp (ElemAt 0)) [Var 0];
          Op (IntOp (Const 0)) []
        ]
    ]
    (Let [Op (MemOp (UpdateCons)) [Var 3; Var 4; Var 0]]
      Call ts (SOME l_append')
        [
          Op (BlockOp (ElemAt 1)) [Var 2];
          Var 3;
          Var 1;
          Op (IntOp (Const 1)) []
        ] NONE))
```

Figure 2.9: The BVI syntax of the optimized append' worker function.

3

Extending BVI Semantics

In this chapter, I will describe the semantics of BVI and formalize the semantics of the new operations introduced in Section 2.2.

3.1 Functional Semantics

The semantics of all intermediate languages in the CakeML compiler backend, including BVI, are defined using functional big-step semantics [Owens et al., 2016]. This means that the formal semantics for each language is defined as a recursive function rather than as an inductive relation. This function is computable and resembles an interpreter for BVI, though one should note that this function is never actually computed; it exists in order to reason formally about semantic equivalence between BVI and the intermediate languages from which, and to which, it compiles. The use of functional semantics facilitates the process of writing HOL4 proofs, which use rewriting, and it enables the use of a clock in the semantic definition to reason about program termination and divergence. For this report, I have chosen to present CakeML semantics as inductive relations, but the reader should note that these rules are actually formalized as recursive functions in the compiler implementation.

3.2 Defining Semantics

Here, I present a *simplified* semantics of BVI. Expressions are evaluated in an environment ρ and a semantic state. In this report, I present the state as being comprised of a clock τ, a code store ω, and a ref store μ. If the clock reaches zero during evaluation, a timeout exception is returned and further subexpressions are not evaluated. The clock enables formal reasoning about program termination, as theorems can be universally quantified over all values of τ. Each function in a program is registered in the code store, which maps function names to function body expressions and the arity of the function. Memory is represented as a map from addresses to *ref*s. The code store is constructed during compilation. Entries are never modified during evaluation, but new ones may be added by dynamic compilation. CakeML supports dynamic compilaton [Sewell et al., 2023], and new entries can be added during evaluation.

The semantics I present here have been simplified to focus on the components most relevant to the new operations. The full semantic state definition of BVI [Kumar et al., 2014]

$$
\begin{array}{rcl}
v & ::= & \mathsf{Number}\ int \\
 & | & \mathsf{Word64}\ word64 \\
 & | & \mathsf{Block}\ num\ (v\ list) \\
 & | & \mathsf{CodePtr}\ num \\
 & | & \mathsf{RefPtr}\ num
\end{array}
\qquad
\begin{array}{rcl}
ref & ::= & \mathsf{ValueArray}\ (v\ list) \\
 & | & \mathsf{ByteArray}\ (word8\ list) \\
 & | & \mathsf{Thunk}\ thunk\_mode\ v \\
 & | & \mathsf{MutBlock}\ num\ (v\ list)\ v\ (v\ list)
\end{array}
$$

$$
thunk\_mode ::= \mathsf{Evaluated} \mid \mathsf{NotEvaluated}
$$

$$
\begin{array}{rcl}
\mathsf{True} & = & \mathsf{Block\ true\_tag\ []} \\
\mathsf{False} & = & \mathsf{Block\ false\_tag\ []} \\
\mathsf{Unit} & = & \mathsf{Block\ tuple\_tag\ []}
\end{array}
\qquad
\begin{array}{rcll}
\rho & : & v\ list & environment \\
\omega & : & num \mapsto (exp, num) & code\ store \\
\tau & : & num & clock \\
\mu & : & num \mapsto ref & ref\ store
\end{array}
$$

$$
\begin{array}{rcl}
result & ::= & \mathsf{Rval}\ (v\ list) \\
 & | & \mathsf{Rerr}\ e
\end{array}
\qquad
\begin{array}{rcl}
abort & ::= & \mathsf{Rtype\_error} \\
 & | & \mathsf{Rtimeout\_error}
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & \mathsf{Rraise}\ v \\
 & | & \mathsf{Rabort}\ abort
\end{array}
$$

Figure 3.1: Simplified semantics of BVI.

also includes a foreign function interface (FFI), a compiler oracle, and a compile function, which I have elided from this presentation for visual clarity. The compiler oracle and compile function are both used for dynamic compilation.

Blocks are the values that result from evaluation of a constructor (refer to the `BlockOp Cons` operation from Section 2.1.1). They are used to represent booleans, which are not represented as primitive *v* types. Booleans True and False are each built instead from an empty Block with a unique true_tag or false_tag, respectively. Tuples, similarly, are constructed using a Block with the tuple_tag and a payload whose size is equal to the arity of the tuple. Unit, in particular, is defined as the 0-tuple.

A CodePtr holds a name mapping to code in the code store ω and is used to evaluate `Call` expressions (Section 2.1). A RefPtr holds a reference to (possibly mutable) memory in μ as constructed by a `MemOp`, introduced in Section 2.1.1. The values in the ref store each have type *ref*. The MutBlock is new in this project and has a structure which resembles Block. Both have a *tag* field of type *num* and a payload of *v*s; however, the payload of the MutBlock is broken up into a *v list* of values to the left of the hole, the value of the hole, and another list of *v*s to the right of the hole. It is important to note that a MutBlock is allocated in the ref store, while a Block is not; this introduces complexity to the verification of semantic equivalence, which is discussed in Chapter 6.

Note: In the full semantics, RefPtr has an additional *bool* argument which indicates whether equality tests are allowed. For references to MutBlocks this field is always *F*, so I have chosen to omit this field for brevity. By design, the bit-level representation of a MutBlock is indistinguishable from that of a Block, so the equality test is prohibited.

3.3 Expression Semantics

Evaluation of BVI expressions is defined by the following relation over expression *lists*:

$$\rho, \omega, \tau_1, \mu_1 \vdash_e (exp\ list) \Downarrow result, \tau_2, \mu_2$$

This mirrors the implementation in the compiler and enables induction on `Let`, `Call`, and `Op` expressions, which each have *exp list* subexpressions. Per Figure 3.1, a list of expressions evaluates to a *result* of either an error an Rerr e, or an Rval with list of corresponding values v. If the *result* is an Rval, it is the case that the resulting *v list* is the same length as the original *exp list* – this is a theorem in CakeML that I invoke frequently! Note that, as mentioned above, the code store ω may be modified during evaluation. This fact is not relevant for the optimization or for the rules presented here, so I have omitted a final ω_2 from the judgement.

Figure 3.2 gives many of the semantic rules. Rules for error propagation are not included here, but discussed briefly in Section 3.3.1. The rules for `Thunk` have been omitted, and the rules for `Raise` (Section 3.3.1), `Call` (Section 3.3.2), and `Op` (Section 3.3.3) have been deferred to subsequent sections.

Since the semantics is defined over lists of expressions, the E-ListNil and E-ListCons rules enable the construction of rules for each expression type, which are written in terms of a singleton list containing the expression. The E-Var rule performs lookup of the value in ρ at de Bruijn index n. Environment ρ is zero-indexed, and the E-Let rule inserts the most recently bound variables at the front of ρ. `Let` expressions accept a list of binders xs, which are evaluated using the list rules. The resulting vs are prepended to the environment ρ in which to evaluate x. It is worth noting that earlier vs are not bound during evaluation of later xs. This will prove useful during the transformation in Chapter 4. The E-Tick rule dictates that the clock is to be decremented before evaluating the single subexpression of the `Tick` construct. As discussed in Section 3.3.1, decrementing a zero-valued clock results in a timeout error.

3.3.1 Error Propagation

As I have mentioned, the rules from Figure 3.2 do not account for explicit error propagation. This was a deliberate choice to focus on the most relevant inference rules, especially those introduced in Section 3.4.

An e of type Rraise results from evaluation of a `Raise` x expression, shown in Figure 3.3. It includes the v resulting from evaluation of x, which can be an error message, status code, or similar. The other errors result from evaluation of an expression where no inference rule applies. An Rtimeout_error results from inference rules, such as E-Tick or E-Call (Section 3.3.2), with a precondition on a non-zero clock that is not met.

An Rtype_error results from inference rules with other preconditions that are not met, such as the constraints on the de Bruijn index in E-Var or the boolean predicate in E-IfThen and E-IfElse. It is important to note that there is no static typing on BVI directly; the

E-ListNil

$$\frac{}{\rho, \omega, \tau, \mu \ \vdash_e \ [] \ \Downarrow \ \mathsf{Rval}\ [], \tau, \mu}$$

E-ListCons

$$\frac{\rho, \omega, \tau_1, \mu_1 \ \vdash_e \ [x_1] \ \Downarrow \ \mathsf{Rval}\ [v], \tau_2, \mu_2 \qquad \rho, \omega, \tau_2, \mu_2 \ \vdash_e \ x_2 :: xs \ \Downarrow \ \mathsf{Rval}\ vs, \tau_3, \mu_3}{\rho, \omega, \tau_1, \mu_1 \ \vdash_e \ x_1 :: x_2 :: xs \ \Downarrow \ \mathsf{Rval}\ (v :: vs), \tau_3, \mu_3}$$

E-Var

$$\frac{n < length(\rho)}{\rho, \omega, \tau, \mu \ \vdash_e \ [\texttt{Var}\ n] \ \Downarrow \ \mathsf{Rval}\ [\rho[n]], \tau, \mu}$$

E-Let

$$\frac{\rho, \omega, \tau_1, \mu_1 \ \vdash_e \ xs \ \Downarrow \ \mathsf{Rval}\ vs, \tau_2, \mu_2 \qquad vs +\!\!+ \rho, \omega, \tau_2, \mu_2 \ \vdash_e \ [x] \ \Downarrow \ \mathsf{Rval}\ [v], \tau_3, \mu_3}{\rho, \omega, \tau_1, \mu_1 \ \vdash_e \ [\texttt{Let}\ xs\ x] \ \Downarrow \ \mathsf{Rval}\ [v], \tau_3, \mu_3}$$

E-IfThen

$$\frac{\rho, \omega, \tau_1, \mu_1 \ \vdash_e \ [x_1] \ \Downarrow \ \mathsf{Rval}\ [\mathsf{True}], \tau_2, \mu_2 \qquad \rho, \omega, \tau_2, \mu_2 \ \vdash_e \ [x_2] \ \Downarrow \ \mathsf{Rval}\ [v], \tau_3, \mu_3}{\rho, \omega, \tau_1, \mu_1 \ \vdash_e \ [\texttt{If}\ x_1\ x_2\ x_3] \ \Downarrow \ \mathsf{Rval}\ [v], \tau_3, \mu_3}$$

E-IfElse

$$\frac{\rho, \omega, \tau_1, \mu_1 \ \vdash_e \ [x_1] \ \Downarrow \ \mathsf{Rval}\ [\mathsf{False}], \tau_2, \mu_2 \qquad \rho, \omega, \tau_2, \mu_2 \ \vdash_e \ [x_3] \ \Downarrow \ \mathsf{Rval}\ [v], \tau_3, \mu_3}{\rho, \omega, \tau_1, \mu_1 \ \vdash_e \ [\texttt{If}\ x_1\ x_2\ x_3] \ \Downarrow \ \mathsf{Rval}\ [v], \tau_3, \mu_3}$$

E-Tick

$$\frac{\rho, \omega, \tau_1 - 1, \mu_1 \ \vdash_e \ [x] \ \Downarrow \ \mathsf{Rval}\ [v], \tau_2, \mu_2 \qquad \tau_1 > 0}{\rho, \omega, \tau_1, \mu_1 \ \vdash_e \ [\texttt{Tick}\ x] \ \Downarrow \ \mathsf{Rval}\ [v], \tau_2, \mu_2}$$

Figure 3.2: Semantics of some CakeML *exp*s.

E-RAISE

$$\rho, \omega, \tau, \mu \vdash_e [\texttt{Raise}\ x] \Downarrow \mathsf{Rerr}\ (\mathsf{Rraise}\ x), \tau, \mu$$

Figure 3.3: Semantics of `Raise`.

E-CALL

$$\frac{\rho, \omega\langle l \mapsto (n, e)\rangle, \tau_1, \mu_1 \vdash_e xs \Downarrow \mathsf{Rval}\ vs, \tau_2, \mu_2 \qquad length(xs) = n \qquad vs, \omega\langle l \mapsto (n, e)\rangle, \tau_2 - (\tau_{call} + 1), \mu_2 \vdash_e [e] \Downarrow \mathsf{Rval}\ [v], \tau_3, \mu_3 \qquad \tau_2 > \tau_{call}}{\rho, \omega\langle l \mapsto (n, e)\rangle, \tau_1, \mu_1 \vdash_e [\texttt{Call}\ \tau_{call}\ (\texttt{SOME}\ l)\ xs\ \texttt{NONE}] \Downarrow \mathsf{Rval}\ [v], \tau_3, \mu_3}$$

E-CALLHANDLE

$$\frac{\rho, \omega\langle l \mapsto (n, e)\rangle, \tau_1, \mu_1 \vdash_e xs \Downarrow \mathsf{Rval}\ vs, \tau_2, \mu_2 \qquad length(xs) = n \qquad vs, \omega\langle l \mapsto (n, e)\rangle, \tau_2 - (\tau_{call} + 1), \mu_2 \vdash_e [e] \Downarrow \mathsf{Rerr}\ (\mathsf{Rraise}\ err), \tau_3, \mu_3 \qquad err :: vs, \omega\langle l \mapsto (n, e)\rangle, \tau_3, \mu_3 \vdash_e [x] \Downarrow r, \tau_4, \mu_4 \qquad \tau_2 > \tau_{call}}{\rho, \omega\langle l \mapsto (n, e)\rangle, \tau_1, \mu_1 \vdash_e [\texttt{Call}\ \tau_{call}\ (\texttt{SOME}\ l)\ xs\ (\texttt{SOME}\ x)] \Downarrow r, \tau_4, \mu_4}$$

Figure 3.4: Semantics of CakeML `Call` *exp*s.

evaluation function mentioned in Section 3.1, which resembles an interpreter, checks each of these preconditions and yields an appropriate error if necessary. Type safety is instead ensured during verification. The correctness theorem for BVI asserts that the evaluation of a BVI program will not produce a type error, provided that it is compiled from a program which itself does not evaluate to a type error.

3.3.2 Function Call Semantics

The `Call` expression is equipped with three inference rules, excluding those for type and timeout errors. E-CALL is applied to function calls that do not have an error handler and that include a name l to a CakeML function in the code store. For such a `Call`, the function body is retrieved from ω, the arguments evaluated in the current environment ρ, and the resulting values used as the environment in which to evaluate the body. Evaluation of the function body decrements the clock by $\tau_{call} + 1$: the number of clock ticks the function call consumes, and an additional tick for the invocation of the call itself.

The E-CALLHANDLE rule is applied to functions that include a name l to a CakeML function in the code store and that also have an error handler. The semantics are similar to those of E-CALL, but if evaluation of the function body produces an error, the handler is executed in an environment constructed from prepending the error to ρ. There is, of course, a rule for such a `Call` with a handler in which the call does not fail, but I have omitted it for brevity. The semantics of such a call are the same as for E-CALL. `Call` expressions whose optional name field is `NONE` represent computed jumps, the rule for which is omitted here.

$$\text{E-OP}\quad \frac{\rho, \omega, \tau_1, \mu_1 \vdash_e xs \Downarrow \mathsf{Rval}\ vs, \tau_2, \mu_2 \qquad vs, \mu_2 \vdash_{op} op \Downarrow v, \mu_3}{\rho, \omega, \tau_1, \mu_1 \vdash_e [\mathtt{Op}\ op\ xs] \Downarrow \mathsf{Rval}\ [v], \tau_2, \mu_3}$$

Figure 3.5: Semantics of CakeML `Op` *exp*s.

The E-CALL rule is most relevant to this project, as it is **only** these calls which are eligible for the TMC transformation. The E-CALLHANDLE rule passes control flow back to the executing environment for error handling, so it is not possible to make such functions tail recursive using TMC alone.

3.3.3 Op Semantics

The rule E-OP is given in Figure 3.5 and defines `Op` semantics as first the evaluation of the `Op` arguments *xs* in ρ and then evaluation of the *op* according to the relation $\vdash_{op}$ in an environment consisting of the resulting *vs*.

The relation $\vdash_{op}$ is defined over the *op*s presented in Chapter 2, of which there are many. Here, I present only the rules most relevant to the project: those that are used in the transformation and those that provide a useful comparison from the new *op* rules introduced in Section 3.4. As shown in Figure 3.5, the environment should be understood as the result of evaluating the `Op` arguments. These rules define the contract for which arguments are to be provided to each `Op`, and evaluation of an `Op` whose arguments do match this contract results in a type error. In these rules, I omit the *result*, and use the judgement

$$\rho, \mu_1 \vdash_{op} op \Downarrow v, \mu_2$$

Note that other operations not listed here may produce non-type errors and may interact with other components of the state besides the ref store μ. This notation was chosen to only include components of the state relevant to the operations in this section.

The OP-INTOP-CONST rule defines evaluation of constant integers; exactly zero `Op` arguments are expected here. The OP-BLOCKOP-CONS rule defines evaluation of a constructor, which returns a Block constructed from the evaluated `Op` arguments and the indentifying *tag*. OP-MEMOP-REF allocates a ValueArray at the next available address in the *ref* store and returns a RefPtr to that address. Once allocated, elements of the ValueArray can be retrieved and modified using the OP-MEMOP-EL and OP-MEMOP-UPDATE rules, respectively.

OP-BLOCKOP-CONS and OP-INTOP-CONST are directly relevant to the TMC transformation introduced in Chapter 4, which is applied to a `BlockOp Cons` in tail position and uses an `IntOp (Const 0)` to represent unfilled holes. The `MemOp` rules presented here illustrate existing semantics that interact with the *ref* store μ. The semantics of the new `MemOp`s look similar and are discussed next.

$$\frac{}{[], \mu \vdash_{op} \texttt{IntOp}\ (\texttt{Const}\ i) \Downarrow \mathsf{Number}\ i, \mu}\ \textsc{Op-IntOp-Const}$$

$$\frac{}{vs, \mu \vdash_{op} \texttt{BlockOp}\ (\texttt{Cons}\ tag) \Downarrow \mathsf{Block}\ tag\ vs, \mu}\ \textsc{Op-BlockOp-Cons}$$

$$\frac{ptr = min\{x | x \notin dom(\mu)\}}{vs, \mu \vdash_{op} \texttt{MemOp Ref} \Downarrow \mathsf{RefPtr}\ ptr, \mu\langle ptr \mapsto \mathsf{ValueArray}\ vs\rangle}\ \textsc{Op-MemOp-Ref}$$

$$\frac{0 \leq i < length(vs) \qquad a = \mathsf{ValueArray}\ vs}{[\mathsf{RefPtr}\ ptr, \mathsf{Number}\ i], \mu\langle ptr \mapsto a\rangle \vdash_{op} \texttt{MemOp El} \Downarrow vs[i], \mu\langle ptr \mapsto a\rangle}\ \textsc{Op-MemOp-El}$$

$$\frac{0 \leq i < length(vs) \qquad a_1 = \mathsf{ValueArray}\ vs \qquad a_2 = \mathsf{ValueArray}\ vs[i \mapsto v]}{[\mathsf{RefPtr}\ ptr, \mathsf{Number}\ i, v], \mu\langle ptr \mapsto a_1\rangle \vdash_{op} \texttt{MemOp Update} \Downarrow \mathsf{Unit}, \mu\langle ptr \mapsto a_2\rangle}\ \textsc{Op-MemOp-Update}$$

Figure 3.6: Semantics of select CakeML *op*s.

3.4 Extending Semantics

The semantics of the new `MemOp`s for interacting with MutBlocks are defined in Figure 3.7. OP-MEMOP-MUTCONS splits up the MutBlock payload, provided in its environment, such that the hole is the value at index i. The new MutBlock is allocated at the next available address, and a RefPtr to that address is returned. Note that there is no restriction on the value of the hole, though by convention we represent unfilled holes with the Number 0.

The OP-MEMOP-UPDATECONS rule defines the semantics of "filling" the hole with a new value. It expects this value and the MutBlock pointer in its environment, along with the the index of the hole to fill.

Semantically, `FinalizeCons` converts a MutBlock into a regular Block. The rules are defined inductively over a potentially nested MutBlock resulting from evaluation of the `Op` argument. In the base cases, OP-MEMOP-FINALIZECONS-VAL and OP-MEMOP-FINALIZECONS-REF, finalization simply returns that value. If the value is a RefPtr to something other than a MutBlock, the RefPtr is returned but not dereferenced. The inductive case, OP-MEMOP-FINALIZECONS-HOLEBLOCK, is applied when the `Op` argument evaluates to a RefPtr to a MutBlock. This MutBlock is converted to a related Block with the same tag and the result of recursively finalizing the hole. This relation is defined formally in Section 6.3.1. The hole is recursively finalized with *ptr* removed from the ref store; this guarantees termination – which I have proved formally – by eliminating the possibility of circular references. Note, however, that *ptr* is not removed from the resulting ref store.

OP-MEMOP-MUTCONS

$$\frac{i = length(l) \qquad ptr = min\{x \mid x \notin dom(\mu)\} \qquad m = \mathsf{MutBlock}\ tag\ l\ v\ r}{l{+}{+}[v]{+}{+}r, \mu\ \vdash_{op}\ \texttt{MemOp}\ (\texttt{MutCons}\ tag\ i)\ \Downarrow\ \mathsf{RefPtr}\ ptr, \mu\langle ptr \mapsto m\rangle}$$

OP-MEMOP-UPDATECONS

$$\frac{i = length(l) \qquad m_1 = \mathsf{MutBlock}\ tag\ l\ v_1\ r \qquad m_2 = \mathsf{MutBlock}\ tag\ l\ v_2\ r}{[\mathsf{RefPtr}\ ptr, \mathsf{Number}\ i, v_2], \mu\langle ptr \mapsto m_1\rangle\ \vdash_{op}\ \texttt{MemOp UpdateCons}\ \Downarrow\ \mathsf{Unit}, \mu\langle ptr \mapsto m_2\rangle}$$

OP-MEMOP-FINALIZECONS-VAL

$$\frac{v \neq \mathsf{RefPtr}\ ptr}{[v], \mu\ \vdash_{op}\ \texttt{MemOp FinalizeCons}\ \Downarrow\ v, \mu}$$

OP-MEMOP-FINALIZECONS-REF

$$\frac{p = \mathsf{RefPtr}\ ptr \qquad a \neq \mathsf{MutBlock}\ tag\ l\ v\ r}{[p], \mu\langle ptr \mapsto a\rangle\ \vdash_{op}\ \texttt{MemOp FinalizeCons}\ \Downarrow\ p, \mu\langle ptr \mapsto a\rangle}$$

OP-MEMOP-FINALIZECONS-MUTBLOCK

$$\frac{p = \mathsf{RefPtr}\ ptr \qquad m = \mathsf{MutBlock}\ tag\ l\ v_1\ r \qquad [v_1], \mu\ \vdash_{op}\ \texttt{MemOp FinalizeCons}\ \Downarrow\ v_2, \mu \qquad b = \mathsf{Block}\ tag\ (l{+}{+}[v_2]{+}{+}r)}{[p], \mu\langle ptr \mapsto m\rangle\ \vdash_{op}\ \texttt{MemOp FinalizeCons}\ \Downarrow\ b, \mu\langle ptr \mapsto m\rangle}$$

Figure 3.7: Semantics of the new CakeML *mem_op*s.

4

Formalizing the Transformation

For a function f that is eligible for the TMC optimization, the transformation produces a wrapper function f_{wrap} and a worker function f_{work}, which correspond to `append` and `append'` from Figure 1.6, respectively. The exact eligibility criteria are outlined in Section 4.1. The function f_{wrap} has the same interface as f. The function f_{work} takes two auxiliary arguments: a pointer to an existing MutBlock and the index of its hole. It returns Unit, and instead writes its result into the hole specified by its arguments.

The compilation process is outlined in Section 4.2, which illustrates how f_{wrap} and f_{work} are constructed from f. During this process, f_{wrap} replaces f, while f_{work} is inserted with a fresh name. The untransformed function f does not appear in the compiled program. The process relies on two transformations, $\overset{wrap}{\rightsquigarrow}$ and $\overset{work}{\rightsquigarrow}$, which are formalized mathematically in subsequent sections.

Specification of f_{work}

The function f_{work} does most of the heavy lifting for the optimization. It accepts two auxiliary arguments: a pointer to an existing MutBlock and the index of the hole therein. Instead of returning a value, f_{work} writes its result to the hole specified by evaluation of its arguments. At each stage of the recursion in f, where the recursive call is nested in one or more constructors, it:

1. uses `MutCons` to allocate equivalent MutBlocks for these constructors, with the MutBlock of each nested constructor filling the hole of its parent,
2. uses `UpdateCons` to write the (top-most) newly allocated MutBlock into the hole specified by its arguments,
3. finishes with a tail recursive call, including as arguments the pointer and index of the (bottom-most) newly allocated MutBlock.

The function f_{work} returns Unit. It allocates MutBlocks in the ref store, each of which is reachable from its parent. The full data structure is walked during finalization by f_{wrap}, so there is no need to return intermediate RefPtrs. In the base case, instead of returning the value yielded by evaluation of the expression in tail position, this value is written to the final hole. The rules for constructing f_{work} from f are presented in Section 4.6.

Specification of f_{wrap}

The function f_{wrap} is a wrapper around f_{work}. It is the optimized equivalent of f and has the same interface so that it can replace f in the code store. It accepts no auxiliary arguments, and returns the same Block as f. At the first stage of the recursion in f, where the recursive call is nested in one or more constructors, it:

1. uses `MutCons` to allocate equivalent MutBlocks for these constructors, with the MutBlock of each nested constructor filling the hole of its parent,
2. calls f_{work} instead of f, including as arguments the pointer and index of the (bottom-most) newly allocated MutBlock,
3. returns the result of applying `FinalizeCons` to the top-most MutBlock, which has been fully filled by the call to f_{work}.

The function f_{wrap} is not directly recursive, nor does it make a tail call. It simply initializes the first set of MutBlocks, delegates all recursion to f_{work}, and waits for control flow to return before finalizing. However, a mutual recursion between f_{wrap} and f_{work} can occur if f has multiple recursive calls within the same constructor in tail position; only one can be made tail recursive. An example of this is shown in Section 5.3. The rules for constructing f_{wrap} from f are presented in Section 4.7.

4.1 TMC Eligibility

The unoptimized `append` function from Figure 1.1 is eligible for the TMC transformation because it contains a constructor operation in tail position with a recursive call as an element of the payload. However, the transformation can also be applied to functions with more complicated data structures in tail position, with a recursive call nested arbitrarily deep inside multiple constructor invocations. An example, `duplicate`, is shown in Section 5.1.

Now, I will formalize precisely which functions are eligible for the transformation.

Definition 1. A function f is **eligible** for TMC transformation if its body contains a subexpression e in tail position such that:

1. e is of the form `Op (BlockOp (Cons` tag`))` es.
2. es contains an element e_{hole} which is either a recursive invocation of f without a handler, or is itself an expression that meets these criteria.
3. Every element in es appearing to the right of e_{hole} is guaranteed not to fail and not to be effectful (change state) during evaluation.

The third criterion is a significant but necessary restriction on the operations that may appear to the right of the recursive call. Recall from rule E-ListCons in Figure 3.2 that evaluation order is left to right. The optimized function must, when evaluated, exhibit the same semantic behavior as the unoptimized version. In the unoptimized version, these expressions are evaluated *after* the recursive call. In the optimized version, these expres-

sions are evaluated *before* the recursive call, which is, by design, the final subexpression to be evaluated. If there is a subexpression to the right of e_{hole} whose semantics allow for state change, then the transformed e_{hole} may be evaluated in a different state than the e_{hole} in the original. Likewise, if there is a subexpression to the right of e_{hole} whose semantics allow for failure, then the transformed e_{hole} may not be evaluated at all – which changes semantic behavior if e_{hole} can itself fail or exhibit effects.

On the other hand, it is worth mentioning the restrictions that are *not* imposed by these criteria. The function f is still eligible for optimization if a recursive call appears elsewhere in a different tail position (such as a different branch of an `If` expression) *not* wrapped in a constructor. In this case, the worker transformation replaces that call with a call to f_{work} which passes along its auxiliary arguments unchanged, and the wrapper transformation replaces that call with a call to f_{wrap}. This is shown in the `filter` example in Section 5.2.

Moreover, es may contain a recursive call somewhere to the *left* of e_{hole}. In this case, both the worker and wrapper transformations will replace this call with a call to f_{wrap} after lifting it out of tail position into a `Let` binder, as discussed in Section 4.3. This is shown in the example in Section 5.3 which mirrors binary trees. Generally, recursive calls in non-tail position are permitted and replaced with a call to f_{wrap} in both the wrapper and the worker. *As such, if multiple elements of es meet the inductive criteria, then only the rightmost is chosen as e_{hole}.*

4.2 Compiling Programs

The TMC optimization is implemented as a new phase of the compiler which compiles BVI to optimized BVI. Programs are compiled by iterating through the code store ω (Section 3.2) and constructing a new code store ω' such that each eligible f is *replaced* by f_{wrap}, with its corresponding f_{work} inserted at a fresh address.

Section 4.7 defines a transformation $\overset{wrap}{\rightsquigarrow}$ for constructing the body of f_{wrap}. This also serves as the eligibility check; $\overset{wrap}{\rightsquigarrow}$ succeeds if and only if f is eligible for the TMC optimization. Section 4.6 defines a transformation function $\overset{work}{\rightsquigarrow}$ for constructing the body of f_{work}, parameterized by the expressions ptr and idx. For top-level expressions, ptr and idx are the auxiliary arguments. This transformation always succeeds, but is only applied if f is deemed eligible by $\overset{wrap}{\rightsquigarrow}$.

The compilation process is as follows. For each entry $\langle l \mapsto (e, arity)\rangle \in \omega$:

- If there exists an e_{wrap} such that $e \overset{wrap}{\rightsquigarrow} e_{wrap}$,
  1. The entry $\langle l \mapsto (e_{wrap}, arity)\rangle$ is inserted into ω'.
  2. The entry $\langle l_{work} \mapsto (e_{work}, arity + 2)\rangle$ is inserted into ω', where
     (a) l_{work} is the next available address in ω'.
     (b) $\texttt{Var}\ arity, \texttt{Var}\ (arity + 1) \vdash e \overset{work}{\rightsquigarrow} e_{work}$

- Otherwise, $\langle l \mapsto (e, arity) \rangle$ is inserted into ω'.

It is important that f_{wrap} replaces f at the same address. This ensures that any call to f anywhere in the source program is replaced with a call to f_{wrap} in the compiled program. This includes recursive calls within f itself that do not appear in tail position; these are replaced with a call to f_{wrap}.

Note: the addresses l and l_{work} are known during compilation and transformation. They will appear bound in inference rules in this chapter.

The transformations $\overset{wrap}{\rightsquigarrow}$ and $\overset{work}{\rightsquigarrow}$ are both defined inductively over expressions. For the base cases, I will define three auxiliary transformations, $\overset{cb}{\rightsquigarrow}$ (Section 4.3), $\overset{opt}{\rightsquigarrow}$ (Section 4.4), and $\overset{fill}{\rightsquigarrow}$ (Section 4.5), which are used to fill holes. I present these transformations using inference rules for the sake of presentation. As with the semantics in Chapter 3, each transformation is implemented in HOL4 as a function, with an *option* return type if the transformation is not total. The transformations $\overset{wrap}{\rightsquigarrow}$ and $\overset{work}{\rightsquigarrow}$ can be compared to the "direct-style" and "destination-passing-style" transformations presented in the Rocq formalization used for the OCaml compiler implementation of TMC [Allain et al., 2025].

4.3 Lifting Op Arguments

The most meaningful rewrite occurs in the base case of the induction where a recursive call is nested inside the arguments of a constructor. To facilitate verification in Chapter 6, transformation of these subexpressions, for both the worker and the wrapper, is carried out in two stages.

The first stage is defined in this section and entails lifting and flattening `Op` arguments and `Call` arguments into a let binder. This transformation is given as $\overset{cb}{\rightsquigarrow}$, and its rules are defined in Figure 4.1. By design, it applies to a subexpression of the form e from the criteria outlined in Section 4.1, as well as to recursive calls which may inhabit e_{hole}. These eligibility criteria are baked into the preconditions, such that this transformation serves as the eligibility check for expressions in tail position.

The purity check in T-CB-CONS also enforces that cb_{hole} contains the rightmost recursive call, as function calls are not guaranteed to be pure and cannot appear in es_{right}. This guarantees that $\overset{cb}{\rightsquigarrow}$ is deterministic. Any other recursive calls are lifted and replaced with a call to f_{wrap}, as they no longer appear in tail position. It is worth mentioning that the TMC compiler phase is applied after small functions have been inlined, so simple, pure functions are not automatically excluded from es_{right}.

Here, cb stands for **call block**, the name I have given the CakeML data structure representing the transformed `Op` or `Call` inside the `Let` binder.

Definition 2. e has a **call block** cb if there exists bs such that

$$e \overset{cb}{\rightsquigarrow} \texttt{Let}\ bs\ cb$$

$$\frac{d = \texttt{SOME}\ l \qquad vs = [\texttt{Var}\ 0, \ldots, \texttt{Var}\ (length(es) - 1)]}{\texttt{Call}\ \tau\ d\ es\ \texttt{NONE} \overset{cb}{\rightsquigarrow} \texttt{Let}\ es\ (\texttt{Call}\ \tau\ d\ vs\ \texttt{NONE})}\ \text{T-CB-CALL}$$

$$\frac{\begin{array}{ll} e_{hole} \overset{cb}{\rightsquigarrow} \texttt{Let}\ bs_{hole}\ cb_{hole} & bs = es_{left} +\!\!+ bs_{hole} +\!\!+ es_{right} \\ vs_{left} = [\texttt{Var}\ 0, \ldots, \texttt{Var}\ (length(es_{left} - 1))] & es_{right} \text{ are pure} \\ vs_{right} = [\texttt{Var}\ (length(bs) - length(es_{right})), \ldots, \texttt{Var}\ (length(bs) - 1)] & \end{array}}{\begin{array}{l} \texttt{Op}\ (\texttt{BlockOp}\ (\texttt{Cons}\ tag))\ (es_{left} +\!\!+ [e_{hole}] +\!\!+ es_{right}) \\ \overset{cb}{\rightsquigarrow} \\ \texttt{Let}\ bs\ (\texttt{Op}\ (\texttt{BlockOp}\ (\texttt{Cons}\ tag))\ (vs_{left} +\!\!+ [cb_{hole}] +\!\!+ vs_{right})) \end{array}}\ \text{T-CB-CONS}$$

Figure 4.1: Rules for the transformation $\overset{cb}{\rightsquigarrow}$.

If we view the original `Cons` constructor as a tree of expressions with some path down to the recursive call, a path that contains only constructors along the way, the transformation removes other possible paths by replacing them with terminal `Var`s so that we can reason inductively over the path later. Additionally, the lifting of more complex expressions out of the constructor tree leaves the recursive call as the *only* effectful subexpression. The transformation does not perform any optimization yet. Section 4.4 defines the second phase transformation $\overset{opt}{\rightsquigarrow}$, which is applied *only* to the call block itself.

If the transformation succeeds, then

1. e meets the eligibility requirements from Section 4.1. T-CB-CALL ensures the call is recursive, and T-CB-CONS enforces the purity check as a precondition.
2. cb is either a recursive call whose arguments are variables, or a constructor invocation where exactly one argument is a call block and the others are variables.
3. each variable nested within cb is indexed to point to its original expression in bs under the `Let` binder.
4. `Let` $bs\ cb$ is semantically equivalent to e (Theorem 2).

4.4 Rewriting Call Blocks

The first phase of the transformation constructs bs and cb for an eligible subexpression e such that $e \overset{cb}{\rightsquigarrow} \texttt{Let}\ bs\ cb$. The second stage is defined in this section as:

$$ptr, idx \vdash cb \overset{opt}{\rightsquigarrow} opt$$

The transformation $\overset{opt}{\rightsquigarrow}$ rewrites call blocks into an expressions that, when evaluated, fills

T-OPT-CALL

$$ptr, idx \vdash \texttt{Call}\ \tau\ (\texttt{SOME}\ l)\ vs\ \texttt{NONE} \overset{opt}{\rightsquigarrow} \texttt{Call}\ \tau\ (\texttt{SOME}\ l_{work})\ (vs{+\!\!+}[ptr, idx])\ \texttt{NONE}$$

T-OPT-CONS

$$\frac{\begin{array}{c} idx' = \texttt{Op}\ (\texttt{IntOp}\ (\texttt{Const}\ (length(vs_{left})))\ [] \\ \texttt{Var}\ 1, idx' \vdash cb^{+2}_{hole} \overset{opt}{\rightsquigarrow} opt_{hole} \qquad hole = \texttt{Op}\ (\texttt{IntOp}\ (\texttt{Const}\ 0))\ [] \end{array}}{\begin{array}{ll} ptr, idx \vdash & \texttt{Op}\ (\texttt{BlockOp}\ (\texttt{Cons}\ tag))\ (vs_{left}{+\!\!+}[cb_{hole}]{+\!\!+}vs_{right}) \\ \overset{opt}{\rightsquigarrow} & \\ & \texttt{Let}\ [\texttt{Op}\ (\texttt{MemOp}\ (\texttt{MutCons}\ tag\ idx'))\ (vs_{left}{+\!\!+}[hole]{+\!\!+}vs_{right})] \\ & \texttt{Let}\ [\texttt{Op}\ (\texttt{MemOp}\ \texttt{UpdateCons})\ [ptr, idx, \texttt{Var}\ 0]] \\ & opt_{hole} \end{array}}$$

Figure 4.2: Rules for the transformation $\overset{opt}{\rightsquigarrow}$.

a pre-existing hole. It *only* applies to the call block cb, such that $\overset{opt}{\rightsquigarrow}$ can be composed with $\overset{cb}{\rightsquigarrow}$ to produce an expression e' of the form `Let` $bs\ opt$. The optimized expression e' has a similar structure to e:

1. if e is a recursive call to f, then e' is a recursive call to f_{work}.
2. if e is a series of nested `Cons` constructors ending with a recursive call to f, then e' consists of a series of alternating `MutCons` constructors and `UpdateCons` operations ending with a recursive call to f_{work}.

The expression e' will replace e in the transformation that produces f_{work}. As such, this transformation is parameterized over two arguments, ptr and idx, expressions which evaluate to the RefPtr of the most recently allocated MutBlock and the index of its hole. The function f_{work} is responsible for filling holes, and these parameters point to the next hole to be filled.

The rules for $\overset{opt}{\rightsquigarrow}$ are given in Figure 4.2. Note that I use the syntax e^{+n} to indicate that every de Bruijn index in e should be incremented by n to account for new `Let` binders.

These rules do not have many restrictive preconditions and can be applied to any cb produced by the $\overset{opt}{\rightsquigarrow}$ transformation; the rule T-OPT-CALL applies to a cb produced by T-CB-CALL, and T-OPT-CONS applies to a cb produced by T-CB-CONS. Since the preconditions in the rules for $\overset{opt}{\rightsquigarrow}$ are restrictive enough to determine TMC eligibility, we can assume here, compositionally, that those preconditions have been met.

The rule T-OPT-CALL constructs a recursive `Call` to f_{work}, using the location l_{work} (assumed to bound, as discussed in Section 4.2). It appends as arguments the ptr and $index$ of the hole to be filled by the recursive call. The rule T-OPT-CONS constructs a nested `Let` containing, in evaluation order,

1. a `MutCons` to allocate a new MutBlock matching the shape and *tag* of the original `Cons`, with `Const 0` as the hole.
2. an `UpdateCons` to fill the previous hole, located by evaluation of *ptr* and idx, with the RefPtr yielded by the evaluation of the the `MutCons` (indexed at 0 in the `Let`).
3. an expression resulting from inductive application of $\overset{opt}{\rightsquigarrow}$ on cb_{hole} (with indices shifted by 2 to account for the new binders), with a new *ptr* expression `Var 1` and a new idx expression `Const` ($length(vs_{left})$), representing the index of the hole of the `MutCons` allocated in 1.

4.5 Filling the Hole

The transformations $\overset{cb}{\rightsquigarrow}$ and $\overset{opt}{\rightsquigarrow}$ can be composed to rewrite an eligible expression in tail position when constructing f_{work}. However, every subexpression e in tail position that is *not* eligible is still subject to a rewrite. Such an expression serves as the base case of the recursion in f. As described at the beginning of Chapter 4, the final hole is to be filled with the value yielded by evaluation of e. This rewrite is achieved by the transformation $\overset{fill}{\rightsquigarrow}$ defined by the single inference rule T-FILL, which is also parameterized by the expressions *ptr* and idx.

$$\text{T-FILL}\quad \frac{}{ptr, idx \vdash e \overset{fill}{\rightsquigarrow} \texttt{Op}\ (\texttt{MemOp}\ \texttt{UpdateCons})\ [ptr, idx, e]}$$

Figure 4.3: The one and only inference rule for $\overset{fill}{\rightsquigarrow}$.

Recalling the semantics from Section 3.4, evaluation of this `UpdateCons` will evaluate *ptr* and *idx* to locate the hole, evaluate e to obtain its value, and write that value into the hole. This transformed expression returns Unit instead of that value, which matches the return type of f_{work}.

T-FILL is even less restrictive than the rules for $\overset{opt}{\rightsquigarrow}$, as it can be applied to *any* expression. However, contrasted with $\overset{opt}{\rightsquigarrow}$, which is only ever applied to a call block produced by $\overset{cb}{\rightsquigarrow}$, the transformation $\overset{fill}{\rightsquigarrow}$ is always applied to subexpressions in tail position for which $\overset{cb}{\rightsquigarrow}$ does not produce anything.

4.6 Constructing the Worker

The worker function f_{work} from Definition **??** is defined inductively by the rules given in Figure 4.4. Again, it is parameterized by *ptr* and idx, which, at the top level of the induc-

tion, are the auxiliary variables provided to f_{work}. In the inductive cases, each expression in tail position is rewritten, shifting *ptr* and idx to account for new binders as necessary. For the base cases, the expression is rewritten to a new expression which, semantically, fills the hole located by evaluating *ptr* and idx. The rules that define $\overset{opt}{\rightsquigarrow}$ and $\overset{fill}{\rightsquigarrow}$ provide three ways to "fill the hole":

1. T-OPT-CONS: to rewrite eligible constructor invocations, which contain a recursive call nested in its arguments.
2. T-OPT-CALL: to rewrite recursive calls that appear in tail position *not* wrapped in a constructor.
3. T-FILL: to rewrite any other expression.

Recall from Section 4.4 that T-OPT-CONS and T-OPT-CALL, the rules for $\overset{opt}{\rightsquigarrow}$, must be applied to a call block produced by $\overset{cb}{\rightsquigarrow}$. Also recall from Section 4.5 that T-FILL, the rule for $\overset{fill}{\rightsquigarrow}$, is designed to be applied to those expressions which *cannot* be transformed by $\overset{cb}{\rightsquigarrow}$. The $\overset{work}{\rightsquigarrow}$ rule for each base case is decided accordingly.

1. For `Ops` and `Calls` that can be rewritten with $\overset{cb}{\rightsquigarrow}$, the rule T-WORK-CB applies. It composes $\overset{opt}{\rightsquigarrow}$ with the result of applying $\overset{cb}{\rightsquigarrow}$, after shifting *ptr* and idx to account for the new binders.
2. For `Ops` and `Calls` that *cannot* be rewritten with $\overset{cb}{\rightsquigarrow}$, the rules T-WORK-OP and T-WORK-CALL apply, respectively. These rules invoke $\overset{fill}{\rightsquigarrow}$.
3. Each other base case has a corresponding rule, which invokes $\overset{fill}{\rightsquigarrow}$.

The rules for $\overset{work}{\rightsquigarrow}$ are shown in Figure 4.4.

The transformation $\overset{work}{\rightsquigarrow}$ is a function capable of rewriting any expression. Of course, not every expression *should* be rewritten; not all functions are eligible for TMC transformation. Recall from Section 4.2 that $\overset{work}{\rightsquigarrow}$ is only applied to those function bodies which are first rewritten by $\overset{wrap}{\rightsquigarrow}$. That transformation is introduced next, and is not a function; it only constructs a wrapper function f_{wrap} for functions f which are eligible.

T-WORK-CB applies to eligible constructors in tail position, which can be converted to a call block. Moreover, it applies to recursive calls in tail position *not* wrapped in a constructor. As discussed in Section 4.1, such a call is replaced with a call to f_{work}, passing along *ptr* and idx and not performing any allocation or hole filling; this is exactly the call produced by $\overset{opt}{\rightsquigarrow}$ (T-OPT-CALL from Figure 4.2).

T-WORK-IF

$$\frac{ptr, idx \vdash e_2 \overset{work}{\rightsquigarrow} e_2' \qquad ptr, idx \vdash e_3 \overset{work}{\rightsquigarrow} e_3'}{ptr, idx \vdash \texttt{If}\ e_1\ e_2\ e_3 \overset{work}{\rightsquigarrow} \texttt{If}\ e_1\ e_2'\ e_3'}$$

T-WORK-LET

$$\frac{ptr^{+length(es)}, idx^{+length(es)} \vdash e \overset{work}{\rightsquigarrow} e'}{ptr, idx \vdash \texttt{Let}\ es\ e \overset{work}{\rightsquigarrow} \texttt{Let}\ es\ e'}$$

T-WORK-TICK

$$\frac{ptr, idx \vdash e \overset{work}{\rightsquigarrow} e'}{ptr, idx \vdash \texttt{Tick}\ e \overset{work}{\rightsquigarrow} \texttt{Tick}\ e'}$$

T-WORK-VAR

$$\frac{ptr, idx \vdash \texttt{Var}\ n \overset{fill}{\rightsquigarrow} e'}{ptr, idx \vdash \texttt{Var}\ n \overset{work}{\rightsquigarrow} e'}$$

T-WORK-RAISE

$$\frac{ptr, idx \vdash \texttt{Raise}\ e \overset{fill}{\rightsquigarrow} e'}{ptr, idx \vdash \texttt{Raise}\ e \overset{work}{\rightsquigarrow} e'}$$

T-WORK-FORCE

$$\frac{ptr, idx \vdash \texttt{Force}\ d\ t \overset{fill}{\rightsquigarrow} e'}{ptr, idx \vdash \texttt{Force}\ d\ t \overset{work}{\rightsquigarrow} e'}$$

T-WORK-CALL

$$\frac{\texttt{Call}\ \tau\ d\ es\ h \overset{cb}{\not\rightsquigarrow} \texttt{Let}\ bs\ cb \qquad ptr, idx \vdash \texttt{Call}\ \tau\ d\ es\ h \overset{fill}{\rightsquigarrow} e'}{ptr, idx \vdash \texttt{Call}\ \tau\ d\ es\ h \overset{work}{\rightsquigarrow} e'}$$

T-WORK-OP

$$\frac{\texttt{Op}\ op\ es \overset{cb}{\not\rightsquigarrow} \texttt{Let}\ bs\ cb \qquad ptr, idx \vdash \texttt{Op}\ op\ es \overset{fill}{\rightsquigarrow} e'}{ptr, idx \vdash \texttt{Op}\ op\ es \overset{work}{\rightsquigarrow} e'}$$

T-WORK-CB

$$\frac{e \overset{cb}{\rightsquigarrow} \texttt{Let}\ bs\ cb \qquad ptr^{+length(bs)}, idx^{+length(bs)} \vdash cb \overset{opt}{\rightsquigarrow} opt}{ptr, idx \vdash e \overset{work}{\rightsquigarrow} \texttt{Let}\ bs\ opt}$$

Figure 4.4: Rules for the worker transformation $\overset{work}{\rightsquigarrow}$.

T-WRAP-IF-THEN

$$\frac{e_2 \overset{wrap}{\rightsquigarrow} e_2' \qquad e_3 \overset{wrap}{\not\rightsquigarrow} e_3'}{\texttt{If}\ e_1\ e_2\ e_3 \overset{wrap}{\rightsquigarrow} \texttt{If}\ e_1\ e_2'\ e_3}$$

T-WRAP-IF-ELSE

$$\frac{e_2 \overset{wrap}{\not\rightsquigarrow} e_2' \qquad e_3 \overset{wrap}{\rightsquigarrow} e_3'}{\texttt{If}\ e_1\ e_2\ e_3 \overset{wrap}{\rightsquigarrow} \texttt{If}\ e_1\ e_2\ e_3'}$$

T-WRAP-IF-BOTH

$$\frac{e_2 \overset{wrap}{\rightsquigarrow} e_2' \qquad e_3 \overset{wrap}{\rightsquigarrow} e_3'}{\texttt{If}\ e_1\ e_2\ e_3 \overset{wrap}{\rightsquigarrow} \texttt{If}\ e_1\ e_2'\ e_3'}$$

T-WRAP-LET

$$\frac{e \overset{wrap}{\rightsquigarrow} e'}{ptr, idx \vdash \texttt{Let}\ es\ e \overset{wrap}{\rightsquigarrow} \texttt{Let}\ es\ e'}$$

T-WRAP-TICK

$$\frac{e \overset{wrap}{\rightsquigarrow} e'}{ptr, idx \vdash \texttt{Tick}\ e \overset{wrap}{\rightsquigarrow} \texttt{Tick}\ e'}$$

T-WRAP-CONS

$$\frac{\begin{array}{c} op \overset{cb}{\rightsquigarrow} \texttt{Let}\ bs\ (\texttt{Op}\ (\texttt{BlockOp}\ (\texttt{Cons}\ tag))\ (vs_{left} +\!\!+ [cb_{hole}] +\!\!+ vs_{right})) \\ \texttt{Var}\ 0, idx \vdash cb_{hole}^{+1} \overset{opt}{\rightsquigarrow} opt \\ idx = \texttt{Op}\ (\texttt{IntOp}\ (\texttt{Const}\ (length(vs_{left})))\ [] \qquad hole = \texttt{Op}\ (\texttt{IntOp}\ (\texttt{Const}\ 0))\ [] \end{array}}{\begin{array}{ll} op \overset{wrap}{\rightsquigarrow} & \texttt{Let}\ bs \\ & \texttt{Let}\ [\texttt{Op}\ (\texttt{MemOp}\ (\texttt{MutCons}\ tag\ idx))\ (vs_{left} +\!\!+ [hole] +\!\!+ vs_{right})] \\ & \texttt{Let}\ [opt] \\ & \texttt{Op}\ (\texttt{MemOp}\ \texttt{FinalizeCons})\ [\texttt{Var}\ 1] \end{array}}$$

Figure 4.5: Rules for the wrapper transformation $\overset{wrap}{\rightsquigarrow}$.

4.7 Constructing the Wrapper

The wrapper function f_{wrap} from Definition **??** is constructed by the $\overset{wrap}{\rightsquigarrow}$ transformation. This transformation visits every tail position in the body of f searching for a constructor invocation e which can be written with $\overset{cb}{\rightsquigarrow}$, and therefore meets the eligibility requirements from Section 4.1. If such an e is found, it is rewritten by the rule T-WRAP-CONS and bubbles up through the inductive cases, indicating that f is eligible for the optimization. Unlike $\overset{work}{\rightsquigarrow}$, the transformation $\overset{wrap}{\rightsquigarrow}$ does not always produce a rewritten expression. An expression is produced *only* if an eligible constructor is found. As such, there are no rules for other base cases.

If such a constructor is found, T-WRAP-CONS rewrites it in accordance with Definition **??** of f_{wrap}. This rewrite produces a nested `Let` with

1. The expressions *bs* which have been lifted by $\overset{cb}{\rightsquigarrow}$, under which the transformed call block *opt* must be evaluated.
2. a `MutCons` to allocate a new MutBlock matching the shape and *tag* of the original

`Cons`, with `Const 0` as the hole.

3. an expression resulting from inductive application of $\overset{opt}{\rightsquigarrow}$ on cb_{hole} (with indices shifted by 1 to account for the new binders), with a *ptr* expression `Var 0` and an *idx* expression `Const` $length(vs_{left})$, representing the hole of the `MutCons` allocation in 1.

4. a `FinalizeCons` operation to rebuild the MutBlock as a regular Block before returning.

This is very similar to the rule T-OPT-CONS of the transformation $\overset{opt}{\rightsquigarrow}$, except that

1. it does not contain an `UpdateCons` subexpression. There is no hole to fill in the wrapper; evaluation of this expression allocates the *first* MutBlock which is filled recursively by $\overset{opt}{\rightsquigarrow}$.

2. the `FinalizeCons` operation is applied to maintain that f_{wrap} returns the same value – a Block – that f would have returned.

Note that the *entire* expression *opt* built inductively by $\overset{opt}{\rightsquigarrow}$ is bound in a single `Let` binder. It is possible that *opt* itself is a deeply nested `Let` expression, but its placement in this binder ensures that the `FinalizeCons` expression can always refer to the top-most `MutCons` using the expression `Var 1`. For some time, this transformation was implemented not by delegating to $\overset{opt}{\rightsquigarrow}$, but by recursing over the call block, with the `FinalizeCons` expression as the base case. This implementation pushed the `FinalizeCons` to the bottom of a deeply nested `Let`, which meant that the de Bruijn index of the top-most `MutCons` was shifted within each `Let` and had to be tracked. To deal with this, I designed a higher order function that recursively allocated the call block while tracking the relevant indices for both the top and bottom MutBlocks, applying the function argument to those indices in the base case. This was elegant in the code and frustrating in the verification. Pushing all allocation into a single `Let` binder proved much simpler.

Note also that this *opt* expression is never referenced again after being let bound. We only care about its *effect* – that it fills the hole of the top-most `MutCons` – and not its return value, which just Unit. This mirrors the `append` function from Figure 1.6, where the result of the recursive call is bound to the wildcard pattern _.

The rules for `If` are interesting as well. There are three, as contrasted with the transformation $\overset{work}{\rightsquigarrow}$, which only has one: T-WORK-IF. The key difference is that $\overset{work}{\rightsquigarrow}$ always performs a rewrite; if one of its branches is not an eligible subexpression, it must still be rewritten in order to fill the hole. In $\overset{wrap}{\rightsquigarrow}$, we want to rewrite *only* a branch which is eligible, and therefore we must track whether each branch is actually rewritten.

Lastly, there is no rule that applies to recursive calls in tail position but *not* wrapped in a constructor. The transformation $\overset{work}{\rightsquigarrow}$ optimizes these calls, but these calls are *already* tail recursive and do not warrant a rewrite in f_{wrap}, which has the same return type. Moreover, the absence of a constructor means there is nothing to allocate, and therefore nothing to finalize, so a transformation mirroring T-WRAP-CONS is unnecessary.

5

Examples

Section 1.1 introduces the TMC optimization of a relatively simple `append` function. In this chapter, I provide three new examples to illustrate the capabilities and limitations of the TMC optimization I have implemented. Since the new operations are implemented in BVI and not CakeML itself, I will again use a bit of invented syntax to represent them:

- `MutCons <x`$_1$`, ..., x`$_{i-1}$`, _, x`$_{i+1}$`, ..., x`$_n$`>` which returns a pointer to a newly allocated MutBlock with a payload that includes a hole at index `i`.
- `UpdateCons k i v` to fill the hole at index `i` of the MutBlock at address `k` with value `v`.
- `FinalizeCons p` to recursively reinterpret the mutable MutBlocks as standard Blocks.

I will also use the standard `list` constructors `::` and `[]` instead of `Cons` and `Nil`.

5.1 Duplicate

The first example illustrates how the TMC optimization rewrites functions with nested constructors. The `duplicate` function applies to a list and returns a new list, where each element now appears twice:

```
fun duplicate []      = []
  | duplicate (x::xs) = x::x::duplicate xs
```

Figure 5.1: Unoptimized `duplicate` function.

In each inductive case of the recursion, the recursive call is inside two `::` constructors. In the optimized worker function `duplicate'`, each inductive case of the recursion:

1. allocates a new MutBlock for the first constructor with pointer `p0`.
2. writes `p0` into the previous hole.
3. allocates a new MutBlock for the second constructor with pointer `p1`.
4. writes `p1` into the hole at `p0`.
5. recurses, providing `p1` and the index 1 as arguments.

```
fun duplicate []      = []
  | duplicate (x::xs) =
      let v0 = x in
      let v1 = x in
      let v2 = xs in
      let p0 = MutCons <v0, _> in
      let _  =
        let p1 = MutCons <v1, _> in
        let _  = UpdateCons p0 1 p1 in
        duplicate' v2 p1 1
      FinalizeCons p0

and duplicate' []      (k : ptr) (i : int) =
      UpdateCons k i []
  | duplicate' (x::xs) (k : ptr) (i : int) =
      let v0 = x in
      let v1 = x in
      let v2 = xs in
      let p0 = MutCons <v0, _> in
      let _  = UpdateCons k i p0 in
      let p1 = MutCons <v1, _> in
      let _  = UpdateCons p0 1 p1 in
      duplicate' v2 p1 1
```

Figure 5.2: TMC optimized `duplicate` function.

Similarly, the optimized wrapper version of `duplicate` performs two allocations, one for each constructor. Note that the second `MutCons` allocation into variable `p1`, the `UpdateCons` which writes to the hole at `p0`, and the call to `duplicate'` are *all* nested in a single expression assigned to the wildcard `_`. This is discussed in Section 4.7 and simplifies construction of the final `FinalizeCons` subexpression; from its perspective, `p0` *always* has de Bruijn index 1, regardless of the number of nested constructors.

Note also that each `x` and `xs` from the original `x::x::duplicate xs` subexpression are lifted into binders `v0`, `v1`, and `v2`. This is an artifact of the call block construction described in Section 4.3, which lifts constructor and call arguments, even if they are already variables. This has no impact on performance, as these extra let-expressions are optimized away in a later phase of the compiler.

5.2 Filter

The second example, `filter` illustrates how the optimization is applied to functions that have an expression in tail position *not* wrapped in a constructor. The `filter` function accepts a list and a predicate function `f`, recurses over the list, and returns a new list containing only those elements `x` of the original for which `f x` is true. In the first branch of the `if` expression, we have a TMC eligible subexpression `x::filter f xs`, with a recursive call to filter inside the `::` constructor. This subjects `filter` to optimization.

```
fun filter (f : 'a -> bool) []      = []
  | filter (f : 'a -> bool) (x::xs) =
      if f x then
        x::filter f xs
      else
        filter f xs
```

Figure 5.3: Unoptimized `filter` function.

The unoptimized `filter` function *also* contains a recursive call `filter f xs` in the second branch of the `if` expression. This alone does not subject `filter` to TMC optimization, but since it is there, it is rewritten in `filter'` to pass along its arguments `k` and `i`. It performs no allocation, so there is no reason for these arguments to change.

In the optimized `filter` function, the second branch is not rewritten; it performs no allocation, so it cannot provide a `k` or `i` to `filter'`. This imposes no cost – the expression is already a tail call and does not leave open a stack frame.

```
fun filter (f : 'a -> bool) []      = []
  | filter (f : 'a -> bool) (x::xs) =
      if f x then
        let v0 = x in
        let v1 = f in
        let v2 = xs in
        let p  = MutCons <v0, _> in
        let _  = filter' v1 v2 p 1
        FinalizeCons p
      else
        let v0 = f in
        let v1 = xs in
        filter v0 v1

and filter' (f : 'a -> bool) []      (k : ptr) (i : int) =
      UpdateCons k i []
  | filter' (f : 'a -> bool) (x::xs) (k : ptr) (i : int) =
      if f x then
        let v0 = x in
        let v1 = f in
        let v2 = xs in
        let p  = MutCons <v0, _> in
        let _  = UpdateCons k i p in
        filter' v1 v2 p 1
      else
        let v0 = f in
        let v1 = xs in
        filter' v0 v1 k i
```

Figure 5.4: TMC optimized `filter` function.

5.3 Mirroring a Binary Tree

The third example shows how the transformation is applied to a function with a constructor in tail position that contains two recursive calls. Provided the following definition of a binary tree, the function `mirror` produces the mirror image of this tree:

```
datatype 'a btree = Leaf
                  | Node of 'a * 'a btree * 'a btree

fun mirror Leaf                = Leaf
  | mirror (Node x left right) =
      Node x (mirror right) (mirror left)
```

Figure 5.5: Unoptimized `mirror` function.

There are two calls to `mirror` inside the `Node` constructor. Unlike in the `filter` example, both recursive calls are executed in each recursive invocation, and only one can be transformed into a tail call. As outlined in Section 4.3, the rightmost call, `mirror left`, is selected to be transformed to use the worker function `mirror'`.

```
fun mirror Leaf                = Leaf
  | mirror (Node x left right) =
      let v0 = x in
      let v1 = mirror right in
      let v2 = left in
      let p  = MutCons <v0, v1, _> in
      let _  = mirror' v2 p 2 in
      FinalizeCons p

and mirror' Leaf                (k : ptr) (i : int) =
      UpdateCons k i Leaf
  | mirror' (Node x left right) (k : ptr) (i : int) =
      let v0 = x in
      let v1 = mirror right in
      let v2 = left in
      let p  = MutCons <v0, v1, _> in
      let _  = UpdateCons k i p in
      mirror' v2 p 2
```

Figure 5.6: TMC optimized `mirror` function.

The first invocation – `mirror right` – is lifted out of the `MutCons` as outlined in Section 4.3 and is left otherwise unchanged, such that it calls the optimized wrapper function `mirror`. What results is a mutual recursion between `mirror` and `mirror'`.

5.4 Inserting Into a Binary Tree

In the previous examples, the index of the hole in each `MutCons` is fixed. The final example illustrates that this is not always the case. The `insert` function inserts an element into an ordered binary tree (ignoring duplicates), using the same `btree` definition as above:

```
fun insert x  Leaf                  = Node x Leaf Leaf
  | insert x1 (Node x2 left right) =
      if x1 < x2 then
        Node x2 (insert x1 left) right
      else if x1 > x2 then
        Node x2 left (insert x1 right)
      else
        Node x2 left right
```

Figure 5.7: Unoptimized `insert` function.

Here, the position of the recursive call in the constructor arguments varies depending on whether `x1` is less than or greater than `x2`. In the optimized version shown on the next page, the position of the hole varies, and the auxiliary index argument becomes necessary in order to modify the correct element in the constructor.

Note also that this example illustrates more clearly how the transformation is applied to a function body with multiple tail positions. Each branch of the nested `if` statement is rewritten separately, each with its own lifted binders, `MutCons`, `UpdateCons`, and (in the wrapper function `insert`) `FinalizeCons`.

```
fun insert x  Leaf                  = Node x Leaf Leaf
  | insert x1 (Node x2 left right) =
      if x1 < x2 then
        let v0 = x2 in
        let v1 = x1 in
        let v2 = left in
        let v3 = right in
        let p  = MutCons <v0, _, v3> in
        let _  = insert' v1 v2 p 1 in
        FinalizeCons p
      else if x1 > x2 then
        let v0 = x2 in
        let v1 = left in
        let v2 = x1 in
        let v3 = right in
        let p  = MutCons <v0, v1, _> in
        let _  = insert' v2 v3 p 2 in
        FinalizeCons p
      else
        Node x2 left right

and insert' x  Leaf                 (k : ptr) (i : int) =
      UpdateCons k i (Node x Leaf Leaf)
  | insert' x1 (Node x2 left right) (k : ptr) (i : int) =
      if x1 < x2 then
        let v0 = x2 in
        let v1 = x1 in
        let v2 = left in
        let v3 = right in
        let p  = MutCons <v0, _, v3> in
        let _  = UpdateCons k i p in
        insert' v1 v2 p 1
      else if x1 > x2 then
        let v0 = x2 in
        let v1 = left in
        let v2 = x1 in
        let v3 = right in
        let p  = MutCons <v0, v1, _> in
        let _  = UpdateCons k i p in
        insert' v2 v3 p 2
      else
        UpdateCons k i (Node x2 left right)
```

Figure 5.8: TMC optimized `insert` function.

6

Verification

This chapter discusses the verification, using HOL4, of the TMC transformation presented in Chapter 4.

6.1 HOL4

CakeML is both implemented and verified in HOL4, as is this optimization. HOL4 [Slind and Norrish, 2008] is an interactive theorem prover that provides a single environment for writing and verifying software. The development process entails:

- writing *definitions*, which can be functions or propositions in higher-order logic,
- writing and proving *theorems* about those definitions.

Proving of theorems is interactive. One proves a theorem by first starting a *goal stack*. Each goal contains an (initially empty) list of assumptions. Tactics can be issued to:

- strip antecedents from implications in the goal and add them to the list of assumptions,
- simplify and rewrite expressions, in both the assumptions and goal statement,
- expand definitions,
- instantiate theorems, including by
  - matching theorem assumptions with existing assumptions, so that the theorem conclusion can be deduced,
  - matching the theorem conclusion with the goal, replacing the goal with the theorem assumptions,
- case split on terms, including but not limited to terms in the assumptions or goal statement. If multiple cases are possible, a new goal is opened for each.

This style of proof is *goal-directed*; rather than the *forward-style* of building upon previously-proved theorems to construct the goal, the goal is progressively reduced until it is simply True. A goal is closed when it is fully reduced to True. Once all goals are closed for the theorem, it is considered proved. The sequence of tactics issued are recorded in the source code, such that the proof is retained and checked each time the theory is built. When the

proof is checked, a forward-style proof is automatically constructed.

6.2 Program Semantics

Chapter 3 on semantics discusses the semantics of BVI expressions. It does not, however, discuss the semantics of BVI *programs*. The semantics of CakeML, and of each phase in the compiler, are defined in terms of:

1. Whether the program fails, terminates successfully, or diverges.
2. The I/O events that are recorded during program evaluation.

For a given phase of the compiler – including the TMC optimization phase – we can define semantic equivalence between source and target programs in terms of whether the two programs always exhibit the same externally-visible behavior (the I/O events) when run. Here, we do not need to enforce that each function in the code store ω has the exact same semantic behavior before and after the transformation. In fact, as is the case with TMC, they likely will not. Rather, we care only about the behavior that a user can observe.

Of course, programs involve the invocation of functions, so we do need a way to relate the semantics of optimized and non-optimized functions that is strong enough to ensure that externally-observed behavior is unchanged. Section 6.4 lays out a theorem which defines this relation for expressions, which applies to all function bodies. The proof of this theorem involves stepping through the execution of both the unoptimized expression and the optimized expression to ensure that the relation is maintained.

6.3 Relating Semantics

How, exactly, do we "relate" semantics? One might be tempted, especially in a compiler phase like TMC, which rewrites expressions into the *same language*, to say that a transformation $e \rightsquigarrow e'$ is correct if we can evaluate e and e' in the same initial states and environments and get the same results:

$$\rho, \omega, \tau, \mu \vdash_e [e] \Downarrow r, \tau', \mu' \implies \rho, \omega, \tau, \mu \vdash_e [e'] \Downarrow r, \tau', \mu'$$

This assertion, however, is too strong and not usually correct.

Recall from Figure 3.1 that Blocks are values that result from evaluating `Cons` constructors, while MutBlocks are refs that are allocated during the evaluation of `MutCons` constructors. For a TMC eligible function, evaluation of the optimized function does not have the same semantics as evaluation of the original. The optimized function allocates MutBlocks that the original does not.

Instead, we need to relax these equalities into *relations* which capture how we expect states and environments to differ between the source and target, both before and after evaluation. This gives us a correctness statement along the lines of:

If, for some suitable relations $\overset{\rho}{\sim}$, $\overset{\omega}{\sim}$, $\overset{\tau}{\sim}$, and $\overset{\mu}{\sim}$, we have that $\rho_s \overset{\rho}{\sim} \rho_t$, $\omega_s \overset{\omega}{\sim} \omega_t$, $\tau_s \overset{\tau}{\sim} \tau_t$,

$\mu_s \overset{\mu}{\sim} \mu_t$ and :

$$\rho_s, \omega_s, \tau_s, \mu_s \vdash_e [e] \Downarrow r_s, \tau'_t, \mu'_s$$

then there exist r_t, τ'_t, μ'_t such that $r_s \overset{res}{\sim} r_t$, $\tau'_s \overset{\tau}{\sim} \tau'_t$, $\mu'_s \overset{\mu}{\sim} \mu'_t$ and:

$$\rho_t, \omega_t, \tau_t, \mu_t \vdash_e [e'] \Downarrow r_t, \tau'_t, \mu'_t$$

This is less concise, but it allows us to reason about, for example, MutBlock allocations that exist in μ_t but not μ_s. The first step, in this section, is actually defining the relations we need. The statement of Theorem 1 in the next section is long, but it captures this idea.

If we step through evaluation of an unoptimized and optimized program simultaneously, as one does while writing proofs in HOL4, how do their respective states relate? In particular, how do the source and target ref stores, μ_s and μ_t, relate? The transformation does not remove any expressions that perform allocation, so every reference in μ_s should also be in μ_t. However, μ_t has entries for the allocated MutBlocks that are not in μ_s. An example snapshot of two such ref stores at simultaneous points of execution is shown in Figure 6.1.

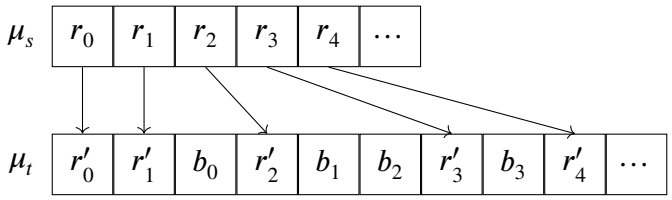

Figure 6.1: μ_s and μ_t at simultaneous points of execution.

Each r_i is some reference in the range of μ_s, and therefore has a related reference r'_i in μ_t. However, it may not appear at the *same index* in μ_t. Also contained in μ_t are several extra entries of the form b_i, representing the MutBlocks that have been allocated in the optimized code. These extra allocations *shift* the indices at which each r'_i is allocated.

To reason about the semantic equivalence between expressions evaluated in μ_s and optimized expressions evaluated in μ_t, we must formalize a relation between these two ref stores. The key ingredient is already illustrated in Figure 6.1: the arrows from each r_i in μ_s to the related r'_i in μ_t.

To formalize this, we define a map F whose domain is μ_s and whose range is a subset of the domain of μ_t. For each r_i, the map F has an entry mapping its index in μ_s to the index of r'_i in μ_t. Those addresses in the domain of μ_t which are *not* in the range of F hold the MutBlocks.

Note that the mapping F is similar, in form and in purpose, to the heap bijections used in the Rocq verification of TMC for use in the OCaml compiler [Allain et al., 2025].

Why are the r'_is in μ_t primed? One might expect them to be the same, but this is not always the case. If r_i is, for example, a ValueArray, then it may contain a value of the form RefPtr *ptr*. This *ptr* is itself an index in μ_s, so the related entry r'_i must actually be RefPtr $F[ptr]$, so that it references the appropriate index in μ_t. Clearly, we need some abstractions to manage this complexity.

6.3.1 Relating Values

Indeed, before we can relate u_s to u_t, we need to relate the values yielded by evaluation of source and transformed expressions. This relation is defined as $\overset{v}{\sim}$ and is parameterized by the ref store address map F. The V-REL-REFPTR rule captures the intuition described in the previous section, that F maps addresses μ_s to addresses in μ_t. Other primitive values relate if they are equal.

$$\frac{}{F \vdash \mathsf{Number}\ i \overset{v}{\sim} \mathsf{Number}\ i}\ \text{V-REL-NUM} \qquad \frac{}{F \vdash \mathsf{Word64}\ w \overset{v}{\sim} \mathsf{Word64}\ w}\ \text{V-REL-WORD}$$

$$\frac{}{F \vdash \mathsf{CodePtr}\ ptr \overset{v}{\sim} \mathsf{CodePtr}\ ptr}\ \text{V-REL-CODEPTR} \qquad \frac{xs \text{ and } ys \text{ relate pairwise by } \overset{v}{\sim} \text{ under } F}{F \vdash \mathsf{Block}\ tag\ xs \overset{v}{\sim} \mathsf{Block}\ tag\ ys}\ \text{V-REL-BLOCK}$$

$$\frac{}{F\langle ptr_s \mapsto ptr_t\rangle \vdash \mathsf{RefPtr}\ ptr_s \overset{v}{\sim} \mathsf{RefPtr}\ ptr_t}\ \text{V-REL-REFPTR}$$

Figure 6.2: Rules for the semantic value relation $\overset{v}{\sim}$.

Additionally, recall from Section 3.4 that the semantics of `FinalizeCons` recursively constructs a tower of Blocks from a tower of related MutBlocks pointed to by its argument. To verify that the `FinalizeCons` expression produced by the $\overset{wrap}{\rightsquigarrow}$ transformation evaluates to the correct Block, we need to formalize what it means for Blocks to relate to MutBlock pointers. As shown in Figure 6.3, mirroring the semantics of `FinalizeCons`, the relation $\overset{mb}{\sim}$ holds if two values relate under $\overset{v}{\sim}$ *or* if one is a Block and the other points to a MutBlock with arguments related under $\overset{v}{\sim}$ and a hole related under $mb_r el$. In MB-REL-BLOCK, there is an additional constraint that $ptr \notin range(F)$. This ensures that b is one of the MutBlocks allocated by the transformation – that it is one b_is in Figure 6.1.

$$\frac{F, \mu \vdash v_1 \overset{v}{\sim} v_2}{F, \mu \vdash v_1 \overset{mb}{\sim} v_2}\ \text{MB-REL-VAL}$$

$$\frac{ptr \notin range(F) \qquad m = \mathsf{MutBlock}\ tag\ l_2\ v_2\ r_2 \qquad l_1 \text{ and } l_2 \text{ relate pairwise by } \overset{v}{\sim} \qquad F, \mu \vdash v_1 \overset{mb}{\sim} v_2 \qquad r_1 \text{ and } r_2 \text{ relate pairwise by } \overset{v}{\sim}}{F, \mu\langle ptr \mapsto m\rangle \vdash \mathsf{Block}\ tag\ (l_1 {+\!\!+} [v_1] {+\!\!+} r_1) \overset{mb}{\sim} \mathsf{RefPtr}\ ptr}\ \text{MB-REL-BLOCK}$$

Figure 6.3: Rules for the relation $\overset{mb}{\sim}$.

6.3.2 Relating Results

The relation $\overset{v}{\sim}$ is valuable not only for relating states; in Theorem 1, we will assert that the result of evaluating f always relates to the result of evaluating f_{wrap}. However, $\overset{v}{\sim}$ alone is not sufficient; these functions may evaluate to an error. In this case, we will assert that the errors should be the same. We need a way not just to relate values, but also to relate *results*:

$$\frac{\textsc{Res-Rel-Val}\quad vs_1 \text{ and } vs_2 \text{ relate pairwise by } \overset{v}{\sim} \text{ under } F}{F \vdash \mathsf{Rval}\ vs_1 \overset{res}{\sim} \mathsf{Rval}\ vs_2} \qquad \frac{\textsc{Res-Rel-Err}}{F \vdash \mathsf{Rerr}\ e \overset{res}{\sim} \mathsf{Rerr}\ e}$$

Figure 6.4: Rules for the semantic result relation $\overset{res}{\sim}$.

6.3.3 Relating References

Now that we can relate values using $\overset{v}{\sim}$, we can relate each r_i from Figure 6.1 to the r'_i it is mapped to under F. This is formalized by the relation $F \vdash r_i \overset{ref}{\sim} r'_i$, which requires all values that appear in r_i and r'_i to relate under $\overset{v}{\sim}$.

$$\frac{\textsc{Ref-Rel-ByteArray}}{F \vdash \mathsf{ByteArray}\ bs \overset{ref}{\sim} \mathsf{ByteArray}\ bs} \qquad \frac{\textsc{Ref-Rel-ValueArray}\quad xs \text{ and } ys \text{ relate pairwise by } \overset{v}{\sim} \text{ under } F}{F \vdash \mathsf{ValueArray}\ xs \overset{ref}{\sim} \mathsf{ValueArray}\ ys}$$

$$\frac{\textsc{Ref-Rel-Thunk}\quad F \vdash x \overset{v}{\sim} y}{F \vdash \mathsf{Thunk}\ tm\ x \overset{ref}{\sim} \mathsf{Thunk}\ tm\ y}$$

$$\frac{\textsc{Ref-Rel-MutBlock}\quad l_s \text{ and } l_t \text{ relate pairwise by } \overset{v}{\sim} \text{ under } F \qquad F \vdash x \overset{v}{\sim} y \qquad r_s \text{ and } r_t \text{ relate pairwise by } \overset{v}{\sim} \text{ under } F}{F \vdash \mathsf{MutBlock}\ l_s\ x\ r_s \overset{ref}{\sim} \mathsf{MutBlock}\ l_t\ y\ r_t}$$

Figure 6.5: Rules for the semantic reference relation $\overset{ref}{\sim}$.

6.3.4 Relating State

We can now formalize Figure 6.1 in mathematical terms, describing precisely how we expect μ_s to relate to μ_t. This is given as the relation $\overset{\mu}{\sim}$, which is parameterized by the

map F and enforces that every address in u_s is mapped to an address in u_t, and that the references at those addresses relate under $\overset{ref}{\sim}$.

$$\textsc{RefStore-Rel}\quad \frac{dom(F) = dom(\mu_s) \qquad \forall i \in dom(\mu_s).\ F \vdash \mu_s[i] \overset{ref}{\sim} \mu_t[F[i]]}{F \vdash \mu_s \overset{\mu}{\sim} \mu_t}$$

Figure 6.6: Rule for the semantic ref store relation $\overset{\mu}{\sim}$.

The preconditions of RefStore-Rel capture most of the specifications of F outlined above in Section 6.3, except that there is no restriction on those addresses in the domain of μ_t and not in the range of F. I will discuss these in Section 6.3.5.

6.3.5 Relating State Change

Figure 6.1 is a *snapshot* of μ_s and μ_t at simultaneous points of execution. This valuable for relating the initial states of the source and target programs, before using HOL4 to step through the evaluation of an expression in each. In fact, the relations I have described so far appear as assumptions in Theorem 1, which defines how semantic behavior relates for expressions evaluated in the source and target. But to reason about how the state *changes*, and to make assertions about the *final* states after simulating execution, we need more than just a snapshot.

Consider an expression e performs allocation and that is to be evaluated in both related ref stores μ_s and μ_t, such that

1. $F \vdash \mu_s \overset{\mu}{\sim} \mu_t$
2. ρ_s and ρ_t relate pairwise by $\overset{v}{\sim}$ under F.
3. $\rho_s, \omega, \tau, \mu_s \vdash_e [e] \Downarrow r_s, \tau', \mu_s'$
4. $\rho_t, \omega, \tau, \mu_t \vdash_e [e] \Downarrow r_t, \tau', \mu_t'$

Certainly, μ_s' and μ_t' should be related as well; evaluation of e allocates the same number of fresh addresses in each, and we should be able to construct a mapping between them. But we cannot say $F \vdash \mu_s' \overset{\mu}{\sim} \mu_t'$, as F does not include entries for these new addresses. Instead, there must be a supermap F' of F, which holds all of the old entries from F as well as new entries mapping the fresh addresses in μ_s' to their counterparts in μ_t'. But for F' to be a sensible supermap, we must assert that those counterparts are actually fresh:

Definition 3. A supermap F' of F **only maps to fresh references** not in μ_t iff:

$$\forall i \in range(F') \setminus range(F).\ i \notin dom(\mu_t)$$

In other words, addresses that are newly mapped-to in F' are also new in μ_t'. This definition prohibits the backwards arrow depicted in Figure 6.7:

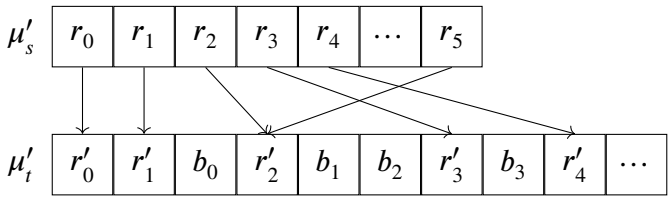

Figure 6.7: μ'_s and μ'_t at simultaneous points of execution, with a non-fresh map entry.

Note that this is not a theorem, and there is nothing yet to prove. It is a definition – one that I have constructed in HOL4 – that stipulates what it means for F' to be a suitable supermap. This will appear in the conclusions of Theorem 1, in which we reason about evaluating two *different* expressions, one optimized and one not, and how we expect the resulting ref stores to compare.

First, we must say something about the b_is, the MutBlocks, which each contain a hole. If, as above, we are considering the simultaneous evaluation of the *same* expression e in both the source and target states, we should not expect any b_i to change. Evaluation of e in μ_s can only modify addresses in μ_s, and so evaluation of e in μ'_s should only modify addresses in the range of F. This leads to another definition:

Definition 4. The **holes are unchanged** from μ_t to μ'_t under F iff:

$$\forall i \notin range(F).\ \mu_t[i] = \mu'_t[i]$$

This definition will also appear in the conclusions Theorem 1, when we reason about evaluation of optimized expressions and which holes are and are allowed to change.

6.4 Theorem of Related Expression Semantics

This section introduces Theorem 1, which states how the semantics of an *expression* evaluated in the source environment and state relate to the semantics of the TMC-transformed expression evaluated in a related target environment and state. There are three conclusions to this theorem:

1. a conclusion about the semantics of the untransformed expression,
2. a conclusion about the semantics of the expression after being transformed with $\overset{wrap}{\rightsquigarrow}$,
3. a conclusion about the semantics of the expression after being transformed with $\overset{work}{\rightsquigarrow}$.

Conclusion 2 is the key to verifying the TMC transformation of an eligible function f, which is replaced in the target code store with the result of applying $\overset{wrap}{\rightsquigarrow}$ to its body. Since a wrapper function f_{wrap} is rewritten to call its corresponding worker function f_{work}, Conclusion 3 is needed in order to reason about the semantics of that worker. The proof of this theorem is inductive, as discussed in Section 6.5, and many of the subexpressions that we induct on do not appear in tail position. For these subexpressions, Conclusion 1 is a necessary (and the only relevant) conclusion of the inductive hypothesis. Moreover, it

validates the behavior of ineligible functions, which are re-inserted into the target code store unchanged.

Theorem 1. Assume that we evaluate xs in the source environment and state, and have a related target environment and state. More precisely, that

1. $\rho_s, \omega_s, \tau, \mu_s \vdash_e xs \Downarrow r, \tau', \mu_s'$
2. ω_t has been initialized from ω_s by the TMC compilation stage, as outlined in Section 4.2. In particular, that for each eligible function f at address l in ω_s, f_{wrap} has been inserted into ω_t at position l, and f_{work} has been inserted into ω_t at a fresh address l_{work}. Ineligible functions appear in ω_t at the same address.
3. $F \vdash \mu_s \overset{\mu}{\sim} \mu_t$ – that is, the initial ref stores μ_s and μ_t resemble Figure 6.1 with arrows represented by F.
4. The environments ρ_s and ρ_t relate pairwise by $\overset{v}{\sim}$ under F.
5. $r \neq$ Rerr (Rabort Rtype_error)

Then we can conclude three things:

1. **Evaluation of the original has related semantics.**
   There exist a new map F', a result r', and a final ref store μ_t' such that:
   (a) $\rho_t, \omega_t, \tau, \mu_t \vdash_e xs \Downarrow r', \tau', \mu_t'$
   (b) $F' \vdash \mu_s' \overset{\mu}{\sim} \mu_t'$ – that is, the final ref stores μ_s' and μ_t' resemble Figure 6.1 with arrows represented by F'.
   (c) $F \sqsubseteq F'$ – where F' only maps to fresh references not in μ_t, per Definition 3.
   (d) $F' \vdash r \overset{res}{\sim} r'$ – that is, both evaluations succeed with related values or fail with the same error.
   (e) The holes are unchanged from μ_t to μ_t'.
2. **Evaluation of the wrapper has related semantics.**
   If $xs = [x]$ and $x \overset{wrap}{\rightsquigarrow} x_{wrap}$, then there exist F_{wrap}, a result r_{wrap}, and a final ref store μ_{wrap} such that:
   (a) $\rho_t, \omega_t, \tau, \mu_t \vdash_e [x_{wrap}] \Downarrow r_{wrap}, \tau', \mu_{wrap}$
   (b) $F_{wrap} \vdash \mu_s' \overset{\mu}{\sim} \mu_{wrap}$
   (c) $F \sqsubseteq F_{wrap}$ – where F_{wrap} only maps to fresh references not in μ_t.
   (d) $F_{wrap} \vdash r \overset{res}{\sim} r_{wrap}$
3. **Evaluation of the worker has related semantics.**
   If:
   (a) $xs = [x]$

(b) $\texttt{Var}\ (length(\rho_t)), \texttt{Var}\ (length(\rho_t) + 1) \vdash x \overset{work}{\rightsquigarrow} x_{work}$

(c) ptr actually points to a MutBlock with index idx in μ_t,

then there exist F_{work}, r_{work}, and μ_{work} such that:

(a) $\rho_t{+}{+}[\text{RefPtr } ptr, \text{Number } idx], \omega_t, \tau, \mu_t \vdash_e [x_{work}] \Downarrow r_{work}, \tau', \mu_{work}$

(b) $F_{work} \vdash \mu'_s \overset{\mu}{\sim} \mu_{work}$

(c) $F \sqsubseteq F_{work}$ – where F_{work} only maps to fresh references not in μ_t.

(d) r is not an error and $r_{work} = \text{Unit}$, or both failed with related errors such that $r \overset{res}{\sim} r_{work}$.

(e) If r is not an error, such that $r = \text{Rval}\ [v]$ for some v, then the hole at $\mu_{work}[ptr]$ has been filled with some v_{hole} such that the MutBlock relation holds: $F_{work}, \mu_{work} \vdash v \overset{mb}{\sim} v_{hole}$.

6.5 Proof Sketch

The proof of Theorem 1 is a proof by induction on the semantics, such that the inductive hypothesis can be applied to a judgement of the form $\rho_i, \omega_i, \tau_i, \mu_i \vdash_e xs_i$ in either of two cases:

1. $\tau_i < \tau$, or
2. $\tau_i \leq \tau$ and the expression size of xs_i is less than that of xs.

The second case is used when invoking the inductive hypothesis on a smaller subexpression, such as the branches of an `If` statement. The first case is used when invoking the inductive hypothesis on the expression of a function body found in the code store ω_s. This is useful when we encounter a recursive `Call`, where the expression in the code store may be larger than the subexpression we are currently cased on, but we know (from the semantics of `Call` in Section 3.3.2), that the clock decrements by at least 1 when the call is made.

The induction begins by breaking xs into the following cases, which are organized by how they are rewritten by the transformation function $\overset{work}{\rightsquigarrow}$.

1. Non-singleton cases, Section 6.5.1:
   (a) CASE-EMPTY: $xs = []$
   (b) CASE-CONS: $xs = x_1 :: x_2 :: xs'$
2. Inductive cases with a tail position, Section 6.5.2:
   (a) CASE-LET: $xs = [\texttt{Let}\ xs'\ x]$
   (b) CASE-IF: $xs = [\texttt{If}\ x_1\ x_2\ x_3]$

(c) CASE-TICK: $xs = [\texttt{Tick}\ x']$

3. Cases which fill the hole, Section 6.5.3:

   (a) CASE-VAR: $xs = [\texttt{Var}\ n]$

   (b) CASE-RAISE: $xs = [\texttt{Raise}\ x]$

   (c) CASE-FORCE: $xs = [\texttt{Force}\ d\ t]$

   (d) CASE-CALL: $xs = [\texttt{Call}\ t\ d\ xs'\ h]$ and $\texttt{Call}\ t\ d\ xs'\ h \overset{cb}{\not\rightsquigarrow} \texttt{Let}\ bs\ cb$

   (e) CASE-OP: $xs = [\texttt{Op}\ op\ xs']$ and $\texttt{Op}\ op\ xs' \overset{cb}{\not\rightsquigarrow} \texttt{Let}\ bs\ cb$

4. Cases subject to optimization, Section 6.6:

   (a) CASE-CALL-CB: $xs = [\texttt{Call}\ t\ d\ xs'\ h]$ and $\texttt{Call}\ t\ d\ xs'\ h \overset{cb}{\rightsquigarrow} \texttt{Let}\ bs\ cb$

   (b) CASE-OP-CB: $xs = [\texttt{Op}\ op\ xs']$ and $\texttt{Op}\ op\ xs' \overset{cb}{\rightsquigarrow} \texttt{Let}\ bs\ cb$

6.5.1 Non-Singletons

Conclusions 2 and 3 of Theorem 1 both apply only when xs is a singleton; the transformations $\overset{work}{\rightsquigarrow}$ and $\overset{wrap}{\rightsquigarrow}$ do not apply to expression lists. For CASE-EMPTY and CASE-CONS, these conclusions are void true. However, we still need to prove Conclusion 1, that evaluation of these untransformed expression lists exhibit the semantics we expect. This is necessary for use in inductive hypotheses. As we will see in CASE-LET in Section 6.5.2, we must apply the inductive hypothesis to the binders before moving on to verify the tail position. In that case, Conclusions 2 and 3 are meaningless.

The proof of Conclusion 1 for CASE-EMPTY is trivial, and can be performed in one line by simplifying with definitions. For CASE-CONS, the inductive hypothesis is invoked twice, once on x_1 and once on $x_2 :: xs'$, and the results composed together. This composition will be discussed in more detail in subsequent sections.

6.5.2 Inductive Cases with a Tail Position

CASE-LET, CASE-IF, and CASE-TICK each have a subexpression in tail position which may be eligible for optimization. Moreover, both $\overset{wrap}{\rightsquigarrow}$ and $\overset{work}{\rightsquigarrow}$ have inference rules that apply to these cases, shown in Figure 4.5 and Figure 4.4, respectively. Therefore, we must prove each of the three conclusions in Theorem 1 for these cases. I will give a brief sketch of my proof for CASE-LET; the other cases are very similar. Here, I will discuss the proof in terms of how one interacts with HOL4, but I will not do this in the rest of the chapter.

For CASE-LET, where $xs = [\texttt{Let}\ xs'\ x]$:

I begin by expanding the definition of the semantics for `Let` – implemented as a recursive function (Section 3.1) – in both the assumption and the goal (refer to Section 6.1 where I discuss verification in HOL4). I apply the inductive hypothesis to $\rho_s, \omega_s, \tau, \mu_s \vdash_e xs'$.

This instantiates the existentially quantified variables from Conclusion 1 of Theorem 1: F_{xs}, r_{xs}, and μ_{xs}. As mentioned in Section 6.5.1, the other conclusions are not relevant here; xs is not in tail position. The propositions from that conclusion are added as assumptions. I case on whether evaluation of r_{xs} is an error, and for the case where it is, I verify that errors propagate appropriately. In the case where it is not, and $r_{xs} = \mathsf{Rval}\ vs$, I invoke the inductive hypothesis again, this time on $vs{+\!\!+}\rho_s, \omega_s, \tau_{xs}, \mu_{xs} \vdash_e x$. The conclusions of the first invocation are used to instantiate the assumptions of the second invocation; in particular, F_{xs} is used as the initial mapping. The second invocation instantiates the existentially quantified variables from all Conclusions, as x is in tail position, including three new maps *each of which is a supermap of* F_{xs}. These variables are used to instantiate the existentially quantified variables I must produce during this proof. For the remaining goals, I invoke a number of lemmas I have written about the relations and definitions from Section 6.3 being transitive and/or preserved under supermaps.

6.5.3 Cases That Fill the Hole

The cases in this category do not have a subexpression in tail position that is eligible for optimization, nor are they eligible for optimization themselves. CASE-VAR, CASE-RAISE, and CASE-FORCE do not have subexpressions in tail position. CASE-CALL and CASE-OP are the cases where the `Call` or `Op` has already been deemed ineligible by the fact that *no call block could be constructed* (Section 4.3). Referring to Figure 4.5, there are no rules for $\overset{wrap}{\rightsquigarrow}$ that rewrite expressions of this form, as they cannot be optimized. Moreover, referring to Figure 4.4, these are exactly the expressions which the transformation $\overset{work}{\rightsquigarrow}$ rewrites using the auxiliary transformation $\overset{fill}{\rightsquigarrow}$. In other words, these are the expressions that actually *fill the hole.*

In these cases, Conclusion 1 of Theorem 1 is proved by a similar process as CASE-LET described in Section 6.5.2, although invocation of the inductive hypothesis varies by expression type. Notably, in CASE-CALL, we induct first on the arguments, and then on the function body expression found in the respective ω_s. For these cases, Conclusion 2 is void true, as $\overset{wrap}{\rightsquigarrow}$ does not produce anything. Conclusion 3 is proved by first unpacking the definition of $\overset{fill}{\rightsquigarrow}$, which is written in terms of a single `UpdateCons` expression (Figure 4.3), and then rewriting with the semantic definition to simulate the new state after the hole is filled and to instantiate the return value as Unit. At this point, the remaining conclusions can be proved by their definitions. I have a handy lemma for this, which I reuse for each case, that assumes all of the relevant original assumptions as well as the propositions from Conclusion 1 (which has been proven at this point), and verifies the $\overset{fill}{\rightsquigarrow}$ logic for any generic expression.

6.6 Verifying the Optimization

The last two cases, CASE-CALL-CB and CASE-OP-CB, are where the core of the transformation is verified. In these cases, xs can be of the form [`Call` $t\ d\ xs'\ h$] or of the form [`Op` (`BlockOp`(`Cons` tag)) xs'], and will henceforth be referred to as $[x]$. In both of these

cases, $x \overset{cb}{\rightsquigarrow} \texttt{Let}\ bs\ cb$.

The rule T-WORK-CB from Figure 4.4 specifies that the worker transformation is constructed by composing $\overset{cb}{\rightsquigarrow}$ and $\overset{opt}{\rightsquigarrow}$. The verification follows a similar structure, by composing two proofs, one for each stage of the transformation. The first is Theorem 2, presented in Section 6.6.1, which states that $\texttt{Let}\ bs\ cb$ is semantically equivalent to x. This allows us to strip away the binders and reason only about the call block, whose only effectful operation is the recursive call. The second is Theorem 3, which very closely resembles the main Theorem 1, except that it applies *only* to call blocks. The inductive structure of the call block, combined with the predictability of its effects, makes the proof of Theorem 3 much simpler than if we were to verify a deeply nested, untransformed `Op` in one go. I know this because I have tried – my original design did not lift anything into binders, and it became quickly difficult to reason about failing subexpressions.

Moreover, because call blocks can represent both `Op`s and `Call`s, this verification structure allows us to solve two cases, CASE-CALL-CB and CASE-OP-CB, using the same reusable theorems. This provides the machinery to optimize functions like `filter` (Section 5.2) almost for free. The process for composing the theorems is outlined in Section 6.6.3.

6.6.1 Verifying Call Blocks

Theorem 2 below states that the expression produced by the transformation $\overset{cb}{\rightsquigarrow}$ is semantically equivalent to the original.

Theorem 2. If $x \overset{cb}{\rightsquigarrow} x'$ and $\rho, \omega, \tau, \mu \vdash_e [x] \Downarrow r, \tau', \mu'$,
then $\rho, \omega, \tau, \mu \vdash_e [x'] \Downarrow r, \tau', \mu'$

The proof rests on the fact that x' must have the form $\texttt{Let}\ bs\ cb$, and is carried out by induction on cb. In the base case, both x and cb are `Call`s. After simplifying with the semantics of `Let`, we have assumptions that

$$\rho, \omega, \tau, \mu \vdash_e [\texttt{Call}\ t\ xs\ h] \Downarrow r, \tau', \mu'$$

$$\rho, \omega, \tau, \mu \vdash_e xs \Downarrow vs, \tau_{xs}, \mu_{xs}$$

and our goal is to prove:

$$vs{+\!+}\rho, \omega, \tau_{xs}, \mu_{xs} \vdash_e [\texttt{Call}\ t\ ns\ h] \Downarrow r, \tau', \mu'$$

where, by Figure 4.1, $ns = [\texttt{Var}\ 0, \ldots, \texttt{Var}\ (length(xs) - 1)]$. This aligns exactly with the semantics of `Call`, which specify that the arguments are to be evaluated first. One can see that evaluating ns in $vs{+\!+}\rho$ will yield exactly vs – the result of evaluating xs.

In fact, one can see from Figure 4.1 that the call block has been carefully designed precisely to enable this semantic equivalence. The inductive case applies similar reasoning, but to the `Op` arguments instead of the `Call` arguments. Extra care is required to account for the shifting of indices, and I have a handful of lemmas to facilitate this. It is also in the inductive case where the purity check of Figure 4.1 comes into play. Any expressions to the right of the hole, which are evaluated *after* the recursive call in the original, are now

evaluated before it. If any of these expression can fail with a different error than the call, or change the state in which the call evaluates, the semantics have changed and this theorem cannot be proven. I have another lemma which states that evaluation of pure expressions never fails and never changes state.

6.6.2 Verifying the Optimization

Theorem 3. I do not provide the full statement of this theorem. This statement is nearly identical to Theorem 1, except that

1. xs is assumed to be a call block [cb]. That is, there exists an x such that

$$x \overset{cb}{\rightsquigarrow} \texttt{Let}\ bs\ cb$$

2. the inductive hypothesis of Theorem 1 is provided as an additional assumption.

As we will see in Section 6.6.3, this theorem will be instantiated in an initial environment consisting of the values obtained by evaluating bs.

Unlike Theorem 1, which inducted over the clock and subexpressions, this theorem inducts over the call block cb.

In the inductive case, where cb has the form `Op` (`BlockOp`(`Cons` tag)) xs', we can easily evaluate the variables to the left and right of the hole. These are variable lookups that, by the initial assumption 5 (borrowed from Theorem 1) that evaluation in the source does not produce a type error, are guaranteed to succeed. They are guaranteed to succeed in the target as well; the environments are pairwise related by assumption 4. The inductive hypothesis gives us everything we need for Conclusion 1. For Conclusion 3, evaluation of the expression transformed by $\overset{work}{\rightsquigarrow}$, we step through the semantics of the `MutCons` and `UpdateCons` of $\overset{opt}{\rightsquigarrow}$ and verify that each memory allocation and update performs as expected, before invoking the inductive hypothesis for the worker. For Conclusion 2, evaluation of the expression transformed by $\overset{wrap}{\rightsquigarrow}$, we rely on Conclusion 3e of the inductive hypothesis and induct on the relation $\overset{mb}{\sim}$ to verify the `FinalizeCons` operation.

In the base case, where cb has the form `Call` t xs h, the process for evaluating the arguments is similar; these are again variables to values known to be in the environment. We then case on whether the clock has run out, and prove that a Timeout exception results in both the source and target if it has (the clocks are assumed to be the same). Once we reach the code lookup, we invoke the inductive hypothesis – not of this theorem, but that of Theorem 1, which has been provided as an assumption – on the function body found in the respective code stores. This invocation gives us the information we need, the conclusions of Theorem 1, to proceed with the proofs of Conclusions 1 and 3, just as in the inductive case. Per Figure 4.5, there is no way to rewrite a `Call` using $\overset{wrap}{\rightsquigarrow}$, so Conclusion 2 is void true.

6.6.3 Composing the Proofs

Now, I will describe how these theorems are composed together to prove Theorem 1, for both CASE-CALL-CB and CASE-OP-CB.

First, Theorem 2 is invoked on $[x]$ in both the source and the target – here, x is *either* the `Call` or `Op` expression – so that we can instead reason about `Let` *bs cb*. From here, we proceed exactly as described in Section 6.5.2 for CASE-LET, starting with invocation of the inductive hypothesis on *bs*. If evaluation of *bs* produces an error, we just have some book-keeping to ensure that the conclusions are met, just as in other cases. If respective evaluation of *bs* in the source and target produces two list of related values, these values serve as the initial environments for evaluation of the call block, according to the semantics of `Let`. In fact, all of the initial assumptions of Theorem 3 follow directly from the inductive hypothesis invocation, so we invoke it now. Its conclusions finish the proof.

6.7 Lifting the Correctness to Observable I/O Behavior

As mentioned in Section 3.2, there is an additional component of the state – the FFI – that does not appear in the judgement rule for the semantics I have defined. It was omitted because it is not modified in any of the inference rules relevant to the transformation, and because the full statement of Theorem 1 assumes the source and target FFIs to be equal before evaluation, and asserts them to be equal after evaluation. It is this data structure that records the I/O events mentioned in Section 6.3, and it is modified by `FFI` operations (Figure 2.1).

Theorem 4 states that the TMC compilation stage produces a program with equivalent semantics.

Theorem 4. If the following assumptions hold:

1. ω_t has been initialized from ω_s by the TMC compilation stage, as outlined in Section 4.2.
2. `Call 0 (SOME` l_{main}`) [] NONE` – representing a call to main – is evaluated in the empty environment and some initial state with the source code store ω_s and an empty ref store.
3. The call to main in the source from Assumption 2 did not fail.
4. `Call 0 (SOME` l_{main}`) [] NONE` is evaluated in the empty environment and *the same* initial state, except with the target code store ω_t.

Then:

1. Both evaluations succeed, or both evaluations diverge.
2. Both of the final FFIs have recorded the same I/O events.

The proof involves applying Theorem 1 to assumptions 1 and 2, leveraging the fact that the initial environments and initial ref stores are both empty and therefore relate under the empty map F. From Conclusion 1 of Theorem 1, we get that the two results of evaluating

main are related by $\overset{res}{\sim}$: in particular, that the result of evaluating main in the target *also* is not an error. We also get that equality of the clock and FFI is preserved after the evaluation. The former ensures that if the source evaluation diverges, then so does the target. The latter ensures that both have recorded the same I/O events.

7

Conclusion and Related Work

In this chapter, I outline the related work and conclude the report with metrics (in lines of code) of the size of the implementation and verification.

7.1 Related Work

Abrahamsson and Myreen present a transformation for automatically introducing tail recursion in CakeML to functions with a binary operation in tail position that is associative and has an identity [2017]. Leijen and Lorenzen have implemented the TMC optimization in the Koka programming language [2023]. This project expands on their work by formally verifying the semantic correctness of the transformation.

As in the Koka implementation, my transformation uses data structures with holes [Minamide, 1998] – namely, the MutBlocks presented in Chapter 3 – which allow for efficient mutation of inductive datatypes. A purely functional, though less efficient, approach to this problem is to use zippers, presented in Huet's functional pearl [1995]. Abbott et al. generalize zippers to any inductive type constructor using a notion of a derivative of data structures [2005].

This project is not the first time that the TMC transformation has been verified. Allain et al. have implemented the optimization in the OCaml compiler, and as a part of that effort, they formally verified their TMC transformation in the Rocq [2025] interactive theorem prover. Their formalization defines a direct-style transformation and a destination-passing-style transformation, similar to my $\overset{wrap}{\rightsquigarrow}$ and $\overset{work}{\rightsquigarrow}$, respectively. It also defines a heap bijection, similar to the mapping F that I define in Section 6.3, to reason about additional allocations performed when evaluating optimized programs. However, the OCaml compiler itself is not end-to-end verified; their verification is separate from the compiler implementation. The CakeML compiler is end-to-end verified, and I have verified that the overall correctness of the compiler is preserved.

The TMC optimization is well suited for functional programming languages that, like CakeML, support algebraic datatypes and represent lists as inductive data structures. Other verified compiler optimizations for functional languages include the inliner for PureCake [Kanabar et al., 2024], which expands definitions of functions at the call site and applies loop specialization to recursive calls with known arguments. CertiCoq, a verified compiler for the Rocq interactive theorem prover, also has an optimizing stage for inlining, as

well as for uncurrying and closure conversion [Anand et al., 2017].

CompCert [Leroy, 2009] is a verified C compiler that has a number of optimization stages, including register allocation, instruction selection, constant propagation, and common subexpression elimination. The TMC optimization is not as relevant to a C compiler, as inductively defined data types are typically implemented using pointers, and recursive allocation can easily be made tail recursive in the source code.

7.2 Discussion

The design of the TMC transformation underwent several iterations. The first did not perform any lifting of expressions into binders using the $\overset{cb}{\rightsquigarrow}$ transformation, as described in Section 4.3. This implementation was simpler, but verification proved to be difficult; if evaluation of a constructor argument failed, it was difficult to reason about which failed and how the error propagated. Lifting into binders allowed for a single instantiation of the inductive hypothesis on the flattened list of binders, as described in Section 6.6.3. It also made very simple the verification in Section 6.6.1 of the call block itself, as the only effectful operation was the recursive call.

Secondly, when constructing the wrapper (using rule T-Wrap-Cons from Figure 4.5), the result of applying $\overset{opt}{\rightsquigarrow}$ is bound in a single `Let` statement around the `Finalize` operation, even though that result may be a deeply nested `Let` expression. This can be seen in the `duplicate` example in Figure 5.2. Previous iterations nested the `Finalize` under those binders, which shifted the de Bruijn index of the `Finalize` argument and prevented $\overset{opt}{\rightsquigarrow}$ from being reused for both the worker and the wrapper transformations. Once I had the insight that the `Finalize` could be pulled out, verification of this transformation became much simpler.

7.3 Future Work

As mentioned in Section 4.1, the purity constraint on expressions which appear to the right of the hole is rather restrictive. One could investigate ways to relax this restriction, for example by permitting an effectful operation if its effect interacts only with references that are guaranteed to be unaffected by the recursive call. Additionally, one could investigate if an expression can be permitted to the right of the hole even if its semantics allow for failure, provided this expression is the only source of error in the function. This could circumvent the issue described in Section 4.1 where semantic behavior of the transformed function differs from that of the original if the recursive call and expression to its right fail with different errors.

Project in Numbers. The implementation of the TMC transformation in the CakeML compiler is approximately **320 lines of code.** The verification of that transformation is approximately **5350 lines of code.**

Bibliography

Michael Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. for data: Differentiating data structures. *Fundam. Inform.*, 65:1–28, 01 2005.

Oskar Abrahamsson and Magnus O. Myreen. Automatically introducing tail recursion in CakeML. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming (TFP)*, volume 10788 of *Lecture Notes in Computer Science*, pages 118–134. Springer, 2017. doi: 10.1007/978-3-319-89719-6_7. URL `https://link.springer.com/chapter/10.1007/978-3-319-89719-6_7`.

Clément Allain, Frédéric Bour, Basile Clément, François Pottier, and Gabriel Scherer. Tail modulo cons, ocaml, and relational separation logic. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi: 10.1145/3704915. URL `https://doi.org/10.1145/3704915`.

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for coq. In *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

Daniel P. Friedman and David S. Wise. Unwinding stylized recursions into iterations. *Comput. Sci. Dep., Indiana University, Bloomington, IN, Tech. Rep*, 19, 1975.

Gérard Huet. Functional p earl the zipper. *Journal of Functional Programming - JFP*, 01 1995.

Hrutvik Kanabar, Kacper Korban, and Magnus O. Myreen. Verified inlining and specialisation for PureCake. In Stephanie Weirich, editor, *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer, 2024. URL `esop24-inlining.pdf`.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, January 2014. doi: 10.1145/2535838.2535841. URL `https://cakeml.org/popl14.pdf`.

Daan Leijen and Anton Lorenzen. Tail recursion modulo context: An equational approach. *Proc. ACM Program. Lang.*, 7(POPL):1152–1181, 2023. doi: 10.1145/3571233. URL `https://doi.org/10.1145/3571233`.

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115,

July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

Yasuhiko Minamide. A functional representation of data structures with a hole. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, page 75–84, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919793. doi: 10.1145/268946.268953. URL https://doi.org/10.1145/268946.268953.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In Peter Thiemann, editor, *European Symposium on Programming (ESOP)*, volume 9632 of *Lecture Notes in Computer Science*, pages 589–615. Springer, April 2016. doi: 10.1007/978-3-662-49498-1_23. URL https://cakeml.org/esop16.pdf.

Tore Risch. *REMREC - A program for Automatic Recursion Removal*. Uppsala University (Datalogilaboratoriet), 1973. URL http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-18538.

Thomas Sewell, Magnus O. Myreen, Yong Kiam Tan, Ramana Kumar, Alexander Mihajlovic, Oskar Abrahamsson, and Scott Owens. Cakes that bake cakes: Dynamic computation in CakeML. In *Programming Language Design and Implementation (PLDI)*. ACM, 2023. URL pldi23-eval.pdf.

Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019. doi: 10.1017/S0956796818000229. URL jfp19.pdf.