BrainCoqulus: A Formally Verified Compiler of Untyped Lambda Calculus to Brainfuck

Thomas Lively
Harvard University

Victor Domene
Harvard University

Gabriel Guimaraes
Harvard University

Abstract

We investigate compilation and verification techniques for functional language compilers by developing and verifying BrainCoqulus, a compiler from the untyped call-by-value lambda calculus to brainfuck. The input language is augmented only with a simple output operation and a notion of input using encoded lists of Church numerals, and the output language is assumed to conform to an idealized but reasonable brainfuck semantics. BrainCoqulus is constructed from layers of intermediate languages including a stack machine language used to cross from the functional to imperative paradigm. We use forward simulation to prove that each translation between intermediate languages preserves program semantics, and by transitivity that the entire compilation preserves semantics.

Due to time constraints, we finished the end-to-end implementation of the compiler in Coq but we were not able to prove all of the compilation correct.

1 Introduction

Drawing on inspiration from previous work in verified compilers such as CompCert [4] that showed that it is possible to implement a provably verified compiler that works in realistic scenarios, we decided to implement an esoteric compiler that despite being verified (probably) should not be used in production environments. BrainCoqulus is a compiler from the untyped lambda calculus, a language first introduced by Alonzo Church in his "An unsolvable problem of elementary number theory" [2], to the classic esoteric language brainfuck created by Urban Müller [1]. Unlike other verfied compilers such as CompCert, however, BrainCoqulus' source language is functional, which creates a host of new verification challenges.

We chose lambda calculus and brainfuck as our source and target languages so we could focus on the challenge of functional to imperative compilation without getting bogged down by complex languages or large instruction sets, and also because it sounded fun.

We implemented the whole compiler using the Coq proof assistant and we proved parts of the compilation correct. We were not able to prove the entire compilation correct due to time constraints and we leave that as future work for ourselves unless someone else fancies the idea of working on a lambda calculus to brainfuck compiler.

The structure of this project is fairly simple. We used three intermediate languages between lambda Calculus and brainfuck forming a total of four language translations. For each of these layers we have a separate compilation routine and the final BrainCoqulus compiler is just the composition of these four routines. The five languages we used are outlined below in the order of source to target language.

- **untyped lambda calculus:** a simple, call-by-value semantics implementation of Lambda Calculus with support for input and output.
- **stack machine language (SML):** a simple stack machine, supporting operations such as `push`, `get`, `del`, `inc`, `out`. More interestingly, SML provides support for function calls with

the `call` primitive, as well as environment closures via the `pack` and `unpack` operations that create tuples on the stack.

- **jump stack machine language (JSML):** a stack machine with *jump semantics*, that is, the programs can no longer call a function and expect to return to the call site. Instead, whenever a function ends, if there is a function identifier on the top of the stack, the machine will jump to the corresponding function and continue its execution.
- **BFN:** a shallow layer on top of brainfuck that supports repeating commands *N* times. This was mostly used as an experiment in our proof and compilation strategy, but it is also useful for writing brainfuck literals in the compiler.
- **brainfuck:** the well-known esoteric language, and our final compilation target.

For each of the above languages, we implemented a reference interpreter that defines the semantics of the language in Coq. Using these interpreters we are able to run programs in any of the above languages. The correctness theorem for the compilation states that each translation between languages preserves semantics as defined by these interpreters. To preserve semantics, each translation must ensure that if the input program would terminate with a given output, then the output program is guaranteed to also terminate with the same output. This relationship is transitive, so if it can be proven for each translation between intermediate languages, it can easily be proven for the overall compilation by composing the proofs.

Throughout the project, we attempted to keep the languages as primitive and simple as possible so we would be able to prove the correctness of the compilation with the fewest number of steps.

1.1 Trusted computing base

The following are in our trusted compute base:

- The reference interpreters for lambda calculus and brainfuck, which together comprise the specification of the semantics that need to be preserved during translation.
- The lambda calculus parser. If the parser parsed all lambda calculus programs as the identity function then it would be trivial and meaningless to prove the correctness of the compiler.
- The Coq proof assistant itself and the compilation chain used to produce a final executable from the Coq project.
- The runtime used to execute the brainfuck programs produced by our compiler. If its execution semantics do not exactly match those of the compiler's reference brainfuck interpreter then the compiler's correctness guarantees do not hold.

2 Contributions

We have implemented a full compiler from untyped lambda calculus to brainfuck, passing through 3 intermediary languages. We have a full proof in Coq that the BFN to BF translation is correct and we have a preliminary proof that the JSML to BFN compilation is also correct. Unfortunately we have never run our compiler end to end. We are able to run each step of the compiler separately, but we get a stack overflow in Coq whenever we try to run the full compiler on the simplest identity function. We estimate that the identity function gets compiled to something on the order of 10^4 brainfuck instructions, which Coq is not able to handle in our current environment. We suspect that exporting the Coq code to OCaml might alleviate the stack overflow issue. Regardless, stack overflows in Coq are merely a quality of service issue, so they have no bearing on the correctness of our compiler.

3 Definition of correctness

BrainCoqulus' main correctness theorem is given in Figure 1. As can be inferred from the theorem, the functions `interpret_lambda` and `interpret_bf` define the semantics of the lambda calculus and brainfuck, respectively. Since both reference interpreters are implemented in Coq, both are required to provably terminate. Since it is possible to write divergent programs in both lambda cal-

```
Theorem compile_correct :
  forall (l : lambda) (input output : list nat),
    (exists fuel, interpret_lambda l input fuel = Some output) ->
    (exists fuel, interpret_bf (compile l) input fuel = Some output).
```

Figure 1: The main correctness theorem for BrainCoqulus

culus and brainfuck, we introduce a fuel argument to the interpreter functions. On each step of the interpreter, the fuel argument is decreased, and if it reaches zero the interpreter stops and the execution is considered unfinished. Given any lambda calculus or brainfuck program and an input, there exists some fuel that makes the reference interpreter finish and return the program's output if and only if the program terminates on that input. This leads to the functional correctness property given by the theorem above. Note that we make no guarantees about the behavior of the compiler's output if it is given input that would cause the input program not to terminate.

In order to modularize the compiler and the proof of correctness, we implemented interpreters defining the semantics of each of our intermediary languages as well. Proving the above correctness theorem amounts to proving similar theorems for each of the translations between intermediate languages in the compilation pipeline.

4 Reference languages

The meaning of correctness in BrainCoqulus depends on the semantics of both lambda calculus and brainfuck, so the reference interpreters for both of these languages form part of the trusted specification of the compiler. In both of these interpreters, a program's input and output are sequences of natural numbers. To simplify the correctness property of the compiler, a program's input is specified in full before execution, i.e. the program is deterministic. In addition, the program is only considered to have produced output if it terminates. The compiler makes no guarantees about the behavior of programs that do not terminate.

4.1 Lambda calculus with output

The lambda calculus used as the input language is the untyped call-by-value lambda calculus with the addition of an output operator, $\wedge$, that generally behaves identically to the identity function. The difference is that when the output operator's subterm is α-equivalent to a Church numeral representing n, n is appended to the output sequence. Lambda calculus programs are lambda terms that are applied at execution time to the term encoding the input as a list of Church numerals. [1] Since this list and number encoding is part of the semantics of the language, they cannot be determined by the user. Although there are infinite ways to encode lists and numbers in the untyped lambda calculus, the canonical encodings used in our compiler are given in Figure 2.

The implementation of the lambda calculus reference interpreter in Coq is simplified by parsing lambda calculus programs into a representation using de Bruijn indices [3]. In this representation, lambda calculus variables are no longer associated with identifiers but rather with numbers identifying how many nested lambdas ago this variable was introduced into the context. Although parsing programs into the representation puts more code into the trusted lambda calculus parser, it is worth the corresponding simplification of the trusted lambda calculus reference interpreter.

4.2 brainfuck

There are many possible semantics for brainfuck [1], but the core language is well established. Like a Turing machine, a brainfuck program has a tape and a single pointer into the tape. All of brainfuck's

[1]It would be possible and in some ways more intuitive to have lambda calculus programs evaluate to the encoded list of Church numerals corresponding to their output, so we may change to this model in the future. This would allow us to remove the non-standard output operator.

$$
\begin{aligned}
\text{ZERO} &\triangleq \lambda f.\lambda x.x \\
\text{SUCC} &\triangleq \lambda n.\lambda f.\lambda x.f\ (n\ f\ x) \\
\text{TRUE} &\triangleq \lambda x.\lambda y.x \\
\text{FALSE} &\triangleq \lambda x.\lambda y.y \\
\text{NIL} &\triangleq \lambda f.\text{TRUE}\ (\lambda x.x) \\
\text{CONS} &\triangleq \lambda a.\lambda l.\lambda f.\text{FALSE}\ (\lambda f.f\ a\ l) \\
\text{ISEMPTY} &\triangleq \lambda l.l\ \text{TRUE} \\
\text{HEAD} &\triangleq \lambda l.l\ \text{FALSE TRUE} \\
\text{TAIL} &\triangleq \lambda l.l\ \text{FALSE FALSE}
\end{aligned}
$$

Figure 2: canonical lambda calculus encodings

eight operations involve this pointer and are given in Table 1. The part of the semantics where implementations vary is how large the tape is, what the wrapping behavior is at the tape boundaries if they exist, and similarly what values cells on the tape can hold and how they wrap when incremented or decremented.

Since there is so much variability anyway, and to reduce the the impedance mismatch between the input and output languages as much as possible, we use an idealized version of the Brainfuck semantics. While most Brainfuck interpreters use fixed-size cells and many use fixed-size arrays, our reference Brainfuck interpreter uses an infinite array of natural numbers, or in other words it represents memory as a function $\mathbb{N} \to \mathbb{N}$. Moving left off the end of the tape and decrementing a 0 valued cell are both no-ops in our semantics. Another choice we made for the brainfuck semantics is to make the read input operator (,) write a zero to the current cell when there is no input available. This means that the input sequence is not allowed to contain zeros so that the end of input can be detected and acted upon.

Not only is the execution model of brainfuck completely different from that of lambda calculus, the way it handles input is also different. In the lambda calculus semantics input is materialized all at once as an encoded list of numbers at the beginning of execution, but in brainfuck the input is part of the execution state and is made available to the program on demand at any point in the execution. This means that at some point during the compilation we need to inject startup code that reads all of the available input and encodes it on the tape in such a way that the compiled program cannot tell it wasn't there immediately at the beginning of execution. This code is injected in the form of a runtime library written in stack machine language (SML), the first intermediate language in BrainCoqulus.

5 Intermediate languages

In this section, we describe in detail the semantics of all intermediate languages used in the compilation process.

5.1 SML

We use a stack machine to bridge the gap between functional and imperative code, and we refer to our stack machine language as SML, not to be confused with Standard ML of New Jersey. Each unique lambda subterm within a program is associated a *function identifier* that can be used to locate the SML translation of its body in a function table. Function IDs also allow functions to be represented by simple natural numbers on the stack. Although each lambda subterm only introduces a single new argument that can be used in its body, a nested lambda subterm has access to a number of arguments equal to its nesting depth. We need a way to package a function and its environment into a single item on the stack, since such closures can be passed around as arguments to other functions or closures. To accomplish this packaging we introduce a notion of tuples into SML.

A `Stack` is defined as essentially a list of items, where each item can be either a single natural number, or a tuple. Unfortunately `Stack` had to be defined as an inductive type instead of as a list because defining it as a list made it impossible to prove that the SML step function terminated, as described below. The SML specification allows the following commands: `push n`, which pushes a natural number to the top of the stack; `get n`, which copies the n-th item in the stack to the top of the stack; `del n`, which removes the n-th item in the stack; `pack n`, which packages the top n items on the

$>$	Move pointer one space right
$<$	Move pointer one space left
$+$	Increment value at pointer
$-$	Decrement value at pointer
,	Set value at pointer to next input value
.	Output value at pointer
[	If value at pointer is 0, skip to instruction after matching]
]	If value at pointer is nonzero, loop back to matching [

Table 1: brainfuck operators

stack into a tuple; `unpack`, which expands the tuple at the top of the stack into its individual entries or does nothing if the top of the stack is a number; `cond_get n k`, which is like `get n` if the top of the stack is a 0 and like `get k` otherwise; `call`, which calls the function corresponding to the function identifier on the top of the stack; `inc` and `dec`, which increase/decrease the number on the top of the stack by 1; `out`, which outputs a natural number on the top of the stack; and `read`, which takes an input from the environment and puts it on the top of the stack. Just as in our brainfuck semantics, `read` pushes a 0 onto the stack if there is no more input.

The semantics of `call` are that the item at the top of the stack is recursively unpacked until it is a number instead of a tuple. Then that number is taken to be a function ID and consumed as the interpreter jumps into the corresponding function body. This means that when a tuple comprised of arguments followed by a function ID is at the top of the stack, the `call` operation will expand that tuple and at the beginning of the function body the arguments will all be available at known stack indices. This is exactly how closures are implemented in SML. The recursive unpacking step is why the stack must be an inductive type, since it can add an arbitrary number of items to the stack, and it is not possible to create measure function to prove this operation terminates if the stack is a list.

5.2 JSML

Given the specification of SML, we realized that implementing the semantics of `call` directly in Brainfuck would be a very difficult task, since there is no way in brainfuck to record the location of and return to the call site of a function. We considered solving this problem by translating the lambda calculus to continuation passing style before lowering it to SML, but that did not work because that translation introduces a new function application that would need to be able to return to its call site. So instead we created a new intermediate language, the jump stack machine language (JSML).

JSML is essentially the same as SML: it supports all of its primitives, with the exception of `call`. A JSML program will execute all commands similarly to a SML program, but whenever a function finishes its execution, if there is a function identifier on the top of the stack, the JSML runtime will *implicitly jump* to the corresponding function and continue its execution. As we will later see, this can be implemented in Brainfuck in a straightforward manner: the execution becomes essentially a while loop with a switch statement, jumping into the correct functions and repeating the process until termination, which is signaled by the function identifier 0.

5.3 BFN

BFN is a shallow layer on top of brainfuck that allows typical brainfuck commands to be repeated a fixed number of times. Thus, instead of writing the cumbersome $>>>>>>>> +++++ <<<<<<<<$, we could write something similar to $(8 >)(5+)(8 <)$. While we could have completed the compiler without this intermediate step, it did serve as an effective proving ground for gaining insight into how the other intermediate languages could be implemented and proven correct.

Perhaps unsurprisingly, writing code in BFN was much more pleasant than directly dealing with brainfuck. In many cases the number of repeated commands was dependent on the BFN implementation of the JSML stack. As we iterated on that implementation, we were able to update all of the BFN stack widgets by simply modifying constants such as `KELL_SIZE` instead of manually changing the number of repeated commands.

6 Compiler overview

Having discussed the semantics of each language used in the BrainCoqulus compilation process, we can now discuss the translations between those languages.

6.1 Lambda calculus to SML

The translation from lambda calculus to SML is relatively straightforward. At the beginning of a function body the stack is known to contain all the variables in that functions environment, which is equal to its nesting depth. Therefore the stack depth of variables can be computed from their de Bruijn indices to translate them into `get` operations. The translation of function applications of e_1 to e_2 first translates e_2 to get its value on the stack, then translates e_1 to get its value, which is a closure or function, on the stack, then appends a `call` operator to perform the function call. The translation of a lambda term appends the translation of its body to the function table then pushes its function ID onto the stack. Finally, the translation of the lambda calculus output operator assumes the the top of the stack is corresponds to a Church numeral $\bar{n}$ and applies that Church numeral to the special function *inc* and 0 and then applies the `out` operator. `inc` is a function that consists only of the `inc` operator, and is part of the SML runtime library so it is always available to the translation. The result of this lowering is that the 0 gets incremented n times then appended to the output stream, as desired.

Note that already in the first intermediate language we have switched from the lambda calculus notion of input and output to that of brainfuck. This combined with the fact that SML is the highest level of our intermediate languages make it the best place to inject startup code to read and encode the program input. The SML runtime library contains the SML translations of the canonical lambda calculus functions, which it calls to properly encode the input. While there is still input remaining, it reads one input value, converts it into a church numeral using the translations of `ZERO` and `SUCC`, then appends that church numeral to an encoded list by calling the SML translation of `CONS`. Once it has finished encoding the input, the runtime calls the translation of the lambda calculus program to execute the program.

6.2 SML to JSML

All commands are compiled from SML to their exact JSML counterpart, with the noticeable exception of the `call` primitive. This makes most of the translation trivial, but the scheme for allowing call semantics without a return address or return instruction requires some ingenuity.

```
f:                  f0:
    push g              push g
    call                push f1
    push 5              get 1
    out                 del 2
    del 0           g:
g:                      push 7
    push 7              out
    out                 del 0
    del 0           f1:
                        push 5
                        out
                        del 0
```

Figure 3: SML code and the corresponding JSML translation

For concreteness, consider the SML program and its translation into JSML in Figure 3. Effectively, the compiler iterates over all functions f in the original SML program (the main function as well as every function in the function table) and splits them into two functions: a "pre-call" (f_0) and a "post-call" (f_1). Since other functions in the SML program can call f and will do so by its identifier, we maintain the mapping of function calls by assigning the same function identifier of f to f_0. Notice that if g had a `call` statement within it, it would also be split into two functions, but the "pre-call" would still correspond to the g function identifier.

```
bfn_right offset (bfn_loop (bfn_dec 1
  (bfn_right move (bfn_inc 1 (bfn_left
   move bfn_end)))) (bfn_left offset
   bfn_end))
```

Figure 4: brainfuck snippet that copies a cell.

Finally, we inject a small snippet of JSML code into f_0 that pushes the function identifier of f_1 as the second item on the stack (using `get` and `del`). When the JSML runtime reaches the end of the execution of f_0, there will be a function identifier to g, the function that was originally called by f. After that function finishes its execution, the next value on the stack will be the function identifier of f_1, and the runtime will jump to it. This emulates the behavior of `call` semantics within JSML.

Notice that it is absolutely *essential* for correctness of JSML translation that the functions called, in this case g, do not leave any remaining state on the stack. This is guaranteed by the translation from Lambda Calculus to SML: any program that we will be dealing with is guaranteed to clean up after its arguments, removing them from the stack when appropriate.

6.3 JSML to BFN

Arguably, this was the most bizarre part of our project. We spent hours debugging brainfuck code that implemented a very specific stack machine, defined by the JSML semantics.

6.3.1 Writing brainfuck code in Coq

In this part of the project, we actually had to write brainfuck code simulating the stack machine supported by JSML. Initially, we were writing brainfuck code directly in Coq: we wrote code as an AST, which would be executed via our BFN interpreter in Coq itself. Debugging brainfuck in these conditions was difficult to say the least. We found ourselves staring at snippets of code with hundreds of brainfuck instructions and ASTs like the one in Figure 4

Thus, to actually write code in a friendly environment, we wrote another brainfuck interpreter in Python and wrote all of our brainfuck code in that environment as Python strings. Once we were reasonably certain of the correctness of our brainfuck code, we wrote a Python routine that compiled Python strings into Coq AST definitions of brainfuck code to automate the process of getting the brainfuck programs back into Coq.

6.3.2 JSML Stack as a brainfuck Tape

The primary design decision when compiling JSML to BFN is deciding how to represent the JSML stack in a brainfuck tape. The basic data unit is what we call a "kell": a group of contiguous brainfuck cells. Initially, a kell consisted of only two cells, corresponding to a *tag* and a *value*. In that context, we would represent the stack containing $42,(21,84)$ as follows: `(0,_)(1,42)(0,_)(2,21)(1,_)(2,84)(0,_)`. A kell with a tag of 0 is always a separator between two *stack items*, which can be either a single natural number or a tuple. The value of a separator kell is unused (represented as an underscore). The separator kells are essential to the stack semantics for two reasons: i) they allow us to seek to previous stack items in a straightforward manner by simply looking at the tags until we reach a zero; ii) they allow us to implement tuples.

An item that is a single natural number n will be, then, of the form `(0,_)(1,n)`. Within a single tuple, the separator between subitems in the tuple is of the form `(1,_)`, and the kell of a natural number n in the tuple is `(2,n)`. With nested tuples, we simply increase the tag values, so that within k nested tuples the outermost tags are $k+1$ and the separators are k. Therefore, the stack containing only the tuple $(42,(21,84))$ is simply

```
(0,_)(2,42)(1,_)(3,21)(2,_)(3,84)(0,_)
```

As we started implementing the JSML semantics in brainfuck, we realized that we would need a notion of registers, or temporary space to perform computation. An `if-else` statement, for instance, can only be executed in brainfuck with additional scratch space available. Furthermore, to implement operations such as `get n` and `del n`, we needed to keep coming back and forth between positions on the brainfuck tape, and we did not have a way to know where to go back to. To address these two issues, we increased the size of a kell to four brainfuck cells. Thus, the final design of a kell includes a tag, idenitifying items and implementing tuples; a value, corresponding to the actual number on the stack; a mark, corresponding to a temporary value used to seek between kells; and a scratch space, used primarily for copies and `if-else` statements.

6.3.3 JSML Operations

Pushing an item n onto the stack involves adding two contiguous kells, containing $(0,0,1,0)$ and $(1,n,1,0)$ respectively. As explained in the previous section, the first kell, with tag 0, indicates the beggining of an item: it is the separator between items. The second kell, with tag 1, indicates an item at depth one, or equivalently a natural number. Note that both of them are initialized with empty scratch space and 1 for the marked cell. In brainfuck, it is much easier to seek to the first zero cell than it is to seek to the first non-zero cell; therefore, we call kells with a 1 in the third cell unmarked, and kells with a 0 in the third cell marked.

In order to implement `get n` and `del n`, we first identify which item needs to be copied or deleted, respectively. Recall that an item in SML is either a natural number or a tuple of items. We are able to seek to items on the tape by looking for kells with tag value 0, which always signify the beginning of an item on the stack. Once we find the desired item, we use the mark cell to mark it as places we would like to seek back to. Since we are deleting or copying kells on the tape and these operations are destructive in brainfuck, we cannot rely on the tag value for seeking. This is when the mark cells and scratch spaces come into play. We keep moving back and forth using the marked cells until the entire item has been deleted or copied to the top of the stack.

A few of the operations, such as `inc`, `out`, `read` and `dec` are trivial implementations in brainfuck. It suffices to seek to the value cell within the kell at the top of the stack and perform the corresponding brainfuck operation. For `pack n`, the stack was designed so as to make this operation as simple as possible: it entails simply increasing the tags of the previous n items on the stack.

Finally, for `unpack` and `cond_get`, we needed some notion of control flow. For `cond_get n k`, this is obvious: if the value at the top of the stack is 0, we want to perform `get n`, and we want `get k` otherwise. For `unpack`, we seek to the last item on the stack, and we want to decrement the tag values only if the current tag value is greater than 1 (namely, if we have a tuple). The actual implementations of these operations is trivial once we have an `if-else` construct. We implemented these semantics: the `if-else(nonzero, zero)` block copies the kell's value to the scratch space, and if it is 0, it executes `zero`; it performs `nonzero` otherwise. The construct expects `nonzero` and `zero` to move the brainfuck tape pointer to the top of the stack at the end of their execution. Finally, the value on the scratch spaces after an `if-else` execution is undefined, and therefore it must be zeroed out before being used.

6.3.4 Implementing Jump Semantics

The compiler takes in a main JSML function, as well as a function table that will be used throughout the program execution. In brainfuck, there is no way to directly force a jump into code section; the code pointer is followed almost entirely in a linear manner, *with the exception of the loop construct*. This is the only way to simulate function calling in brainfuck.

```
Function switch fn_table :=
  match fn_table with
  | [] => stack_top
  | hd :: tl =>
    if_else_val (bfn_dec 1 (switch tl))
      (bfn_left 1 stack_top & del 0 &
      bfn_of_jsmp hd & stack_top)
  end.
```

Figure 5: switch statement in brainfuck implementing function calls

After executing the main function, the compiled brainfuck code enters a while loop whose body is effectively a switch statement (Figure 5). To find the appropriate function to execute, the body of the switch statement works as follows. Suppose the function identifier at the top of the stack in *n*. In the switch statement, if the top of the stack is non-zero, then decrease it and fallthrough to the next if statement; if it is 0, then delete the function identifier and jump into a particular function (which correspond to the function identifier *n* in the original JSML function table). The while loop will terminate when the function identifier at the top of the stack is 0, which signifies the end of the program.

6.4 BFN to Brainfuck

The implementation of this step was trivial: we simply compiled every command like `(8 >)` into 8 repeated > symbols. However, proving the correctness of this translation was not so trivial, as discussed in the next section.

7 Proof Strategy

Our original proof strategy was to simply prove each translation correct however seemed best. But when we struggled to prove the trivial BFN to BF translation correct, it became clear that we needed a more deliberate approach. Since the semantics of each of our languages was already represented with interpreters that repeated step functions from execution state to execution state, the obvious proof method was forward simulation.

7.1 Forward Simulation

To prove translations correct with forward simulation, we first need to find and define a relationship between execution states of the source and target languages that holds for the initial states of both languages when given the same input. When the source language program takes a step from a state where the relation holds, it must be proven that the target language program that was the result of translating the source program takes some number of steps to make the relation hold again. Finding such a relation is enough to prove that no matter how many steps the source language program takes, its translation takes enough steps to maintain the relation. If the relation enforces that the state of the outputs in the source and target programs are the same, then the correctness proof for the translation follows trivially.

The relationship between BFN execution states and their corresponding BF execution states is that they are the exact same except all of the BFNs in the execution state are translated to BFs. Since one step in BFN corresponds to one step in BF, this was easy to prove and from that relation it was easy to prove that the output was always the same. However, this was the only translation we have been able to prove so far. We believe that the forward simulation structure will work for proving the other translations, but finding the relations between the execution states of the other languages is much more difficult.

7.2 Pre-Conditions and Post-Conditions

Each compilation step expects the previous layer to output programs with a given format. This could be formulated in terms of pre-conditions and post-conditions. For instance, the JSML layer expects that a program compiled from lambda calculus to SML be such that it does not leave extraneous arguments on the stack, i.e., it cleans up after its arguments after the execution. This is, indeed, enforced by the SML compilation step, and it is much of the reason why we can implement jump semantics the way we did in Section 6.

Another example of this type of requirement is the fact that JSML to brainfuck compilation expects that the function with identifier equal to 0 indicates the termination of the program, instead of being possibly called during code execution. This allows brainfuck to determine termination.

In BrainCoqulus, the approach taken was to prove lemmas about the output of each intermediate layer. It is possible that introducing some sort of ghost state or assertions within the program itself would have been easier to manage in terms of proofs. In particular, it seems like something that an automated prover such as Dafny could verify, reducing the verification burden when compared to what must be done in Coq.

8 Conclusion and Future Work

Writing a verified compiler for an esoteric language has proved to be quite challenging and fun. The largest challenges have been: i) the bridge between the functional and imperative worlds in the lambda calculus to SML layer, ii) in particular implementing a runtime in SML that converts the input list into a list encoding of Church numerals, iii) the implementation of a stack machine in brainfuck for the SML to BFN layer, and iv) proving any of the layers correct using Coq.

For future work, we leave the obvious task of finishing the verification of all parts of our compilation pipeline. It is also possible to make the compiler aware of more types of data structures and control flow idioms to further optimize the output. Furthermore, future work can also include optimization passes on brainfuck itself, to simplify the code generated from lambda calculus. Optimization passes to decrease the size of the generated brainfuck code are crucial for our compiler to be usable in more realistic scenarios As noted before, the current version of BrainCoqulus generates over 10,000 brainfuck instructions for the simplest of lambda terms.

References

[1] brainfuck. `https://esolangs.org/wiki/Brainfuck`. Accessed: 2017-5-8.

[2] CHURCH, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*, 2 (Apr. 1936), 345–363.

[3] DE BRUIJN, N. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings) 75*, 5 (1972), 381 – 392.

[4] LEROY, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (July 2009), 107–115.