Code Extraction from a Dependently Typed Language to a Stack Based Language

Louis Milliken
Supervisor(s): Jesper Cockx, Lucas Escot
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Code Extraction from a Dependently Typed Language to a Stack-Based Language

LOUIS MILLIKEN*, Delft University of Technology, The Netherlands

Additional Key Words and Phrases: Agda, Forth, Dependent Types, Code Extraction

ABSTRACT

Dependently typed languages such as Agda can provide users certain guarantees about the correctness of the code that they write, however, this comes at the cost of excess code that is not used at run time. Agda code is currently compiled to another language before it is run, there are not many target languages in popular use, so it is unclear if there are other potential target languages that could do a better job. The purpose of this paper is to investigate the efficacy of Forth as a target language for Extraction from Agda, in the hopes that Forth may prove to have the potential to be a better target language than the currently used target languages, Haskell and Javascript. Forth is a stack-based, imperative language, meaning that values are pushed and popped from a stack through the use of 'words' instead of passing and returning values with functions, as is the case in most languages. Agda is a dependently typed, purely functional language. A dependent type is a type whose definition *depends* on another value, which allows for types with very specific information, such as a 'Vec n' representing an array of length 'n'. These dependent types are used to write code with more certainty about its correctness and behaviour. Overall, Forth can be used as a target language to compile Agda code, however many of the advantages of Forth are squandered by executing code written and structured with Agda in mind, making it a rather ineffective target language when compared to other solutions such as Haskell.

1 INTRODUCTION

Dependently typed languages are a subset of programming languages whose type system allows for dependent types. These dependent types can be powerful tools for writing code that can be reasoned about and used with confidence in its correctness. That said, several drawbacks are preventing dependently typed languages from being used in mainstream production. A common strategy to alleviate these problems is to "extract" code from a dependently typed language into another language that can be compiled and run with fewer issues. Examples of dependently typed languages are Agda, Coq, and Idris.

Dependently typed languages are often compiled to different languages when they need to be run. Code written in Coq can be extracted to OCaml [Danil Annenkov 2021; Letouzey 2008] and programs written in Idris can be extracted to Scheme [Idr 2022]. Agda has support for compilation to multiple languages, namely Javascript and Haskell [Agda Team 2022a].

This project aims to investigate the efficacy of different languages to extract Agda code to, In the hopes of finding a language that can perform better than the two current languages used for extraction as previously stated, Javascript and Haskell [Agda Team 2022a].

Extraction to another language is a complicated task, and several problems need to be solved when creating an implementation. For example, if the language being extracted to uses strict evaluation, how would one go about implementing a language that uses lazy evaluation? The same can be said for extracting from a language with implicit memory management (Agda) to one with explicit memory management. Finding solutions to these problems are major hurdles for writing

Author's address: Louis Milliken, Delft University of Technology, The Netherlands, L.M.Milliken@student.tudelft.nl.

an effective code extractor, and the extent to which a given language can solve these problems will determine its suitability as a target language for extraction.

The exact question that will be answered by this research is "How effectively does Forth serve as a target language for extraction from Agda?"
While the many differences between Agda and Forth may make the creation of such an extractor difficult, Forth is an extremely lightweight and low-level language, meaning that if Agda could be extracted to Forth, the resulting code could be considerably faster and less resource-intensive than the same original code when extracted to Haskell or Javascript. Due to the limited scope of this project, it is not expected that the compiler created for this project will outperform currently used compilers, but it may indicate the potential of Forth as a target language. Even if Forth proves to be an inappropriate language for this purpose, we may still find valuable insight into what properties are valuable for a language to have when serving as a target of a dependently typed language, as well as how exactly Forth fails to provide these properties.
This question can be broken down to better identify the criteria for evaluating the efficacy of extracting Agda to Forth:

- How does the performance of evaluating complex data types differ between Agda code extracted to Forth and code compiled using Haskell?
- How does the performance of evaluating higher-order functions differ between Agda code extracted to Forth and code compiled using Haskell?
- What names/words are allowed for variables and functions in Agda, but not in Forth?

These sub-questions can be used to create easily quantifiable measures of success, which will be used to determine the efficacy of Forth as a target language for extraction from Agda.

This paper will first provide additional information by exploring the defining features of Forth and Agda, followed by a breakdown of how Agda code is compiled to Forth, and what additional Forth code is needed to support this. Then, the performance of my compiler will be evaluated, using both benchmarking and reflection on the limitations of which features can be provided in Forth. Section 6 will discuss how I ensured that my research and findings are presented responsibly. Next, Section 7 will provide my final thoughts, explaining why I ultimately don't believe that Forth is an effective target language for compilation from Agda, and finally, some discussion on related works will be had in section 8.

2 BACKGROUND INFORMATION ON FORTH

The target language of my compiler, Forth, is an eagerly evaluated, stack-oriented, imperative language. These characteristics make Forth's design a far cry from the lazily evaluated, purely functional Agda. Forth code is written by combining functions (known as 'words' in Forth) to build up complex features using several smaller ones. These words can be named anything, so long as there are no black characters such as spaces, tabs, and newline characters, guaranteeing that any Agda words will be valid in Forth.

A stack-oriented language behaves differently from others, as instead of passing values between words, values are instead pushed onto a stack, and can then be pushed off later to be used in an operation, the result of which would then be pushed back onto the stack. The stack used for pushing values in this fashion is known as the data stack and is one of two stacks used in standard implementations of Forth, along with the return stack. The return stack is primarily used when words are called to keep track of any locally bound variables and the location that the code returns

to at the end of the called word.

Another feature of Forth that differentiates itself from Agda is that it is an untyped language. all values posted to the stack are simply integer values. This means that my own implementation of a type system will be needed to reproduce Agda features such as pattern matching. This will be explained in more detail in a later section. Despite its apparent drawbacks, this lack of a type system also provides flexibility in terms of what can be pushed to the data stack. Memory addresses and execution tokens - references to words that can then be called from the data stack - are two types of value that greatly increase the range of expressions that can be written in Forth.
Pointers to memory allow for the implementation of complex data structures, and execution tokes allow for the creation of currying and higher-order functions.

These two features make Forth a very simple language to work with, despite being so fundamentally different from most, if not all, mainstream programming languages. Additionally, Forth has a dictionary used to store the definitions and values of words and variables and allows for functions to be used in a concatenative manner, in which words can be called one after the other, assuming that the appropriate values are on the stack for each call. This means that more complex words can be created by stringing several other together, to reduce repeated code and improve readability. Examples of how complex words have been created to allow for functionality like that of Agda's will be shown in Section 4.3.

Forth is a language with many implementations[1], with the most popular implementations typically adhering to the ANS Forth specifications [Knaggs 1998]. Of these many implementations, I decided to write compile with GForth [Ertl 2008] in mind. While it may not have as many specialised features as other implementations such as RVM Forth [Stoddart et al. 2010], there is a large amount of information about GForth and its implementation available online. This is an important factor for me as, until starting this project, I had no experience with Forth.

3 BACKGROUND INFORMATION ON AGDA

As mentioned briefly in both the introduction and the previous section, Agda is a purely functional, dependently typed language. This section will explain what these two properties mean, and how they affect the design of the compiler.

A pure language is a language that can be evaluated as if all expressions are mathematical functions, a function that maps each possible input to a single output. To guarantee this property, expressions cannot have any side-effects which could affect the result if that expression were to be evaluated again. Examples of side-effects include mutable variables, user input, and errors being thrown. Agda being pure means that Agda code can be reasoned about as if it were a mathematical function, which contributes to the certainty one can have about the correctness of Agda code.
Despite being a cornerstone of Agda's design, Forth does not have to be a pure language as well. This is because, by design, impure Agda cannot be written, therefore the resulting situation in which the compiler creates impure Forth code will not arise.

Agda being dependently typed is another key feature that sets it apart from most other languages. "What sets dependent type systems from others is that types can depend on terms" [Norell 2007]. The additional specificity provided by these dependent types means "that more errors can be caught

[1]https://en.wikipedia.org/wiki/Forth_(programming_language)#Implementations

at compile-time, rather than manifesting themselves only when the right circumstances arise at run-time" [Brady 2005]. Dependent Types can be defined using another value to provide additional information about a type.

```
data Vec (A : Set) : Nat → Set where
  []    : Vec A zero
  _::_  : {n : Nat} → A → Vec A n → Vec A (suc n)

tail : {A : Set}{n : Nat} → Vec A (suc n) → Vec A n
tail (_ :: ret) = ret
```

Fig. 1. Example of a dependent type, and a function that makes use of it

Figure 1 provides an example of a dependent type, 'Vec', and a function that relies on the additional information provided by the dependent type. The definition of 'Vec' *depends* on a natural number, which is used to indicate its length. This is then used to create a tail function which only takes vectors with a length greater than zero. This removes the possibility of 'tail' being called on an empty array, which would typically raise a run time error. Guarantees such as this are used to write Agda code with certainty that these types of errors will not occur. The values used to define these dependent types can be treated similarly to regular arguments of functions when compiling to Forth, there are cases in which these values can be completely ignored at compile-time, and as such can be replaced by dummy values

4 COMPILING AGDA TO FORTH

Haskell provides functionality to read in an Agda program, break it down into Agda's internal syntax, and then provide a treeless representation of that syntax. This is immensely helpful as it removes the need to manually parse the Agda code and convert it to an abstract syntax before compilation can even begin. This section will first explain some of the key forms of Agda syntax, and how they are converted to Forth, followed by an overview of some of the built-in types in Agda, and how they allow for optimisations. Finally, the run-time library created to support the newly generated Forth code will be described in detail.

4.1 Agda's Internal Syntax

Agda's Internal syntax provides many definitions, each corresponding to a certain type of operation or declaration. To be able to compile any valid Agda code into Forth code, every definition in the Agda syntax requires a corresponding definition in terms of Forth code. For the sake of brevity, not all of these definitions will be detailed in this section.

- **Constructors**
  In Agda, constructors are used to defining data types, as well as provide methods for creating those data types. An example of such a constructor can be seen in Figure 2 Since Forth is an untyped language, a rudimentary type system needed to be created to accommodate these constructors.

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

Fig. 2. Definition and constructor of the List types [] and _::_ in Agda

```
variable type[] 1 cells allot 1 type[] !
variable XT[]
:noname   type[] here 1 fillArray here 1 cells allot  ; XT[] !
:noname XT[] @ makeTHUNK ; is []
:noname ." [] " ; type[] 1 cells + !

variable type_::_ 1 cells allot 3 type_::_ !
variable XT_::_
:noname   type_::_ here 3 fillArray here 3 cells allot  ; XT_::_ !
:noname XT_::_ @ makeTHUNK ; is _::_
:noname ." _::_ " ; type_::_ 1 cells + !
```

Fig. 3. Definition and constructor of the List types [] and _::_ in Forth

Figure 3 shows how the same List type defined in Agda in Figure 2 is defined in Forth. A type is defined by creating a variable in the Forth environment. The address of this variable can then be treated as an identifier corresponding to the constructor of that type. Similarly to other languages with manual memory management such as C, memory addresses can be treated as arrays by reserving the subsequent locations in memory. This is used in the constructors to represent an object as an array, with the first element of the array containing a pointer corresponding to its type, and each element after containing an attribute of the object. For example, the first line of the definition of '[]' allots one block of memory to make it an array of length 1 as, other than its type, this object has no attributes. On the other hand, the constructor for _::_ creates an array of length three because this type also has a head and tail attribute. It is worth noting that it is also possible to represent data types in Forth such that objects themselves, rather than pointers, can be pushed to the stack [Wavrik 1990]. Such a method would provide a more efficient implementation, however, would lead to considerably more code, and a much more complicated solution to automatically generating constructors that would be beyond the scope of this project.

- **Functions**
  For a given Agda function, the corresponding Forth definition is made up of two parts. The reason for this is to allow for the creation of thunks, the particulars of which will be explained in the next subsection. The first part is a list of nameless functions, with each function representing a lambda expression in Agda. Typically these lambda expressions either take a single argument and return the rest of the function, or they are explicitly defined lambda expressions to be used in the function itself. returns a pointer - or execution token - of the actual function. The second part is a word that takes the elements of this list and passes each word to the next, resulting in a single, fully formed word. While Forth is a stack-based language, there is still the option of binding values on the stack to local variables. This is a neat way of allowing a word to take an 'argument', by taking a value off of the stack and immediately assigning it to a local variable which can then be reused during the execution of the word. Another solution to taking arguments would be leaving them on the stack and retrieving them as needed by keeping track of their position, unfortunately, the implementation of such a feature in the compiler is beyond the scope of this project, as it would take too much time to implement.
  Figure 4 Gives a simple example of a function in Agda; the 'not' function takes a single

```
not : Bool → Bool
not true  = false
not false = true
```

Fig. 4. Definition of the not function in Agda

```
defer not
...
ENDFLAG  :noname { a  }
a if 0 else -1 then  ;
create XTnot fillHere
:noname XTnot foldThunks ; is not
```

Fig. 5. Definition of the not word in Agda

Boolean value and returns the negation of that argument. Figure 5 Shows how this function is translated into Forth. First, the word 'not' is declared at the top of the file, which allows other words to reference it before the actual definition of 'not' has been provided. Then, the actual functionality of the 'not' function is stored in a variable called 'XTnot', which is retrieved and stored as a thunk when the 'not' word is called. Again, the details of thunks will be described in the next subsection. While the difference in size between the Agda and Forth definition of 'not' is fairly small, given that, more complicated expressions that are made up of multiple lambda expressions - either by taking more arguments or creating lambda expressions within the function - become much larger and, unfortunately, harder to read and understand:

```
ite : {A : Set} → Bool → A → A → A
ite true x y = x
ite false x y = y
```

Fig. 6. Definition of the ite (if, then else) function in Agda

```
defer ite
...
ENDFLAG  :noname { d c b a  }
b if c else d then  ;
:noname { c b a  e }
c  b a  e  pass  pass  pass  ;
:noname { b a  f }
b  a  f  pass  pass  ;
:noname { a  g }
a  g  pass  ;

create XTite fillHere
:noname XTite foldThunks ; is ite
```

Fig. 7. Definition of the ite word in Agda

Figure 7 shows the translation of the 'ite' function, Figure 6. Despite only taking two more arguments, the size of the resulting word is much greater than that of 'not'. You can see that for each additional argument, another word is defined. Starting from the bottom and going up, each word takes the next word as its first (leftmost) argument, then all of the previous word's arguments and finally one additional argument.
While many implementations of Forth would face this same issue, there are some, such as RVM Forth that supports *nested* lambda functions[Stoddart and Lynas 2006]. Such a feature would greatly reduce the amount of code generated, as functions consisting of multiple lambda expressions, such as 'ite', could still be written as a single Forth word.

- **Function application**
  Agda allows for functions to take arguments in both an in-order and a post-order fashion, however, this ordering is disregarded once the code has been converted to the abstract syntax. arguments are passed to words in Forth by pushing values to the stack before calling the word in question, the resulting function application must structure the application in reverse polish notation, where arguments are given first, followed by the word that the arguments are being passed to. Agda is an implicitly curried language, meaning that providing a function an argument just returns a new function that takes one less argument until all arguments have been passed. This is not the case in Forth, and as such the 'pass' word is used to manually create a curried version of a given function, with one argument being passed at a time. Since only one argument can be passed at once, the order in which the arguments are written in Forth is the opposite of how they would be written in Agda, as shown in the figure below.

AGDA:
```
foo x y
```

FORTH:
```
y x foo pass pass
```

Fig. 8. Function application in Agda and Forth

Figure 8 Shows how function application works in the two languages, with the 'pass' word serving to create a curried version of 'foo'. I created this word by making use of the 'curry' word defined in Rosetta Code [2]. It takes a thunk and an argument as a function, and returns a new thunk containing an execution token with the argument applied. It is important to note that every time an argument is passed, a new dictionary entry is created to store that word. This means that over time, as more and more new words are created, the dictionary may fill up, causing a dictionary overflow. This will be discussed further in 5.

While there are other definitions in Agda's syntax, these make up a large portion of the definitions that are converted to Forth in a noteworthy manner.

4.2 Built-Ins and Optimisations

Agda supports "built-in' definitions of several types, meaning that, while a definition of a type and the functions that make use of it can be specified in code, another, more efficient implementation can be used at run time.

[2]https://rosettacode.org/wiki/Currying#Forth

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
{-# BUILTIN NATURAL Nat #-}
```

Fig. 9. Definition of the Nat type, with the built-in flag

Figure 9 Shows an implementation of the 'Nat' type in Agda, along with a line indicating that the built-in definition for 'Nat' can also be used. This means that the definition of Nat being made up of 'zero' and 'suc' can still be used in other functions, however, at run time, natural numbers will be represented as actual natural numbers, rather than the recursively defined objects as defined in the code. Agda provides built-ins for several other types, such as Booleans, Integers and Strings [Agda Team 2022b].

4.3 The Forth Run-time Library

As explained in Section 2, Forth is a fundamentally different language to Agda, and as such does not support many features of Agda such as laziness, partial application and pattern matching. That said, Forth words can be created to provide an environment in which these concepts can be used. This is through the creation of a type system, along with a pattern matching method, as well as the creation of the thunk and result classes, which can be used alongside the type system to provide lazy evaluation and partial application.

- **The Obj= Word and Pattern Matching**
  Functions in Agda often rely on pattern matching to allow for functions with specific outputs for different input patterns. As such, a method to compare an object with a given pattern, and return any bound variables in the case of a successful match is required. Making use of the constructors described in Section 4.1, an 'Obj=' method was created to iterate through two given objects at once to ensure that they are both of the same type, and then recursively call itself for each of the two objects' attributes. This word is rather long and for the sake of brevity, will not be included in this report.
  When pattern matching in Agda, the provided pattern can consist of an incomplete object, with missing attributes being represented as a variable, which will then have the argument's corresponding attribute being bound to that variable for use within the function. Such functionality is achieved in Forth through the use of a 'wildcard' object, to fill in any gaps in the provided pattern.

```
...
dup @ WILDCARD = if
    drop
    wildcards pointer @ cells + !
    pointer @ 1 + pointer !
    -1
...
```

Fig. 10. Identifying wildcards in the 'Obj=' word in Forth

Figure 10 shows the portion of the 'Obj=' function that identifies and handles these wildcard objects. Simply put, if the type of the pattern being compared is 'WILDCARD', then the

value of the object that is being compared is added to the 'wildcards' array. The values in this array are then pushed to the stack and bound to local variables if the whole pattern is matched, allowing them to be reused later in the word.
With this 'Obj=' word, Agda's style of pattern matching can easily be achieved through the use of nested 'if, then, else' statements to iterate through each possible pattern and execute the correct code.

- **Thunks and Results** Thunks are a data structure used to enable lazy evaluation in eagerly evaluated languages. Typically, they are represented as an object with two attributes: a flag to indicate whether the thunk has been evaluated, and either an expression waiting to be evaluated or a value, depending on the value of the flag. For the sake of readability, my implementation splits this object into two, each only containing either an expression or value and with their type replacing the flag.

```
variable THUNK 1 cells allot 2 thunk !
: makeTHUNK ( xt -- [THUNK xt] )
    thunk here 2 fillArray
    here
    2 cells allot
;
:noname ." T: " ; THUNK 1 cells + !
variable RESULT 1 cells allot 2 thunk !
: makeRESULT ( val -- [RESULT val] )
    result here 2 fillArray
    here
    2 cells allot
;
:noname ." R: " ; RESULT 1 cells + !
```

Fig. 11. Definition of the THUNK and RESULT types in Forth

Figure 11 Shows the definition of the two types, 'thunk', which is used to store expressions, and 'result' which is used to store the result of evaluating said expressions. Since the types of objects in this type system are stored in the same way as variables, it is a simple matter of replacing the first attribute of a thunk, a pointer to the 'thunk' variable, with the pointer to the 'result' variable to indicate that a thunk has been evaluated. These definitions provide a simple way of delaying evaluation by storing the execution tokens of words which, when given as an argument to the 'pass' word as described in Section 4.1, can easily take additional arguments before they are evaluated.

My run-time library contains two different methods of evaluating these thunks, depending on the situation.
The first, and simpler of the two, 'dethunk' takes an object and, if it is a 'thunk', executes the execution token stored within and returns the result. If the argument is not a thunk, then it is immediately returned instead. While it would be fair to assume that this method would return the result of fully evaluating the expression represented by the thunk, the token executed is still executed lazily, meaning that the object returned could still contain

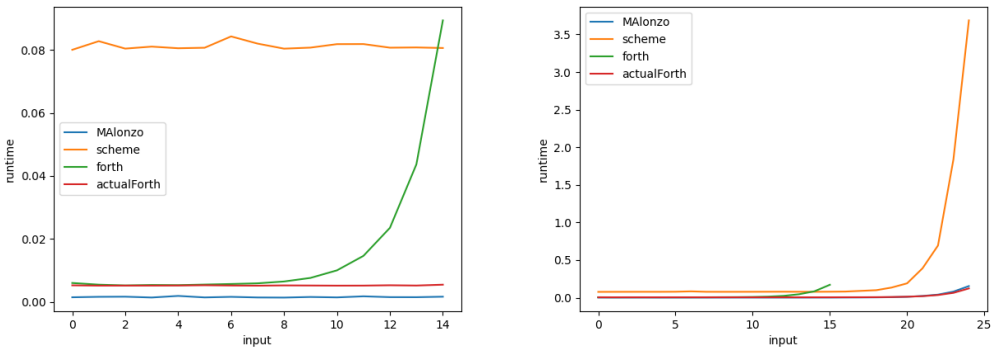
other thunks as attributes. This could be equated to a single act of innermost evaluation; 'dethunk' only really evaluates a single part of the whole expression.
To fully evaluate, or 'force' an expression, the word 'deepdethunk' was created. This word first calls 'dethunk' on the argument to ensure that it is working with an object that is not a thunk. It then recursively calls itself on all attributes of the objects, to fully evaluate the expression and return an object that does not contain any thunks.

5 PERFORMANCE EVALUATION

This section will present the results of benchmarking Agda code when compiled to Forth, as well as other target languages, Haskell, Scheme, HVM and LLVM. These results will be discussed to determine Forth's performance in comparison to other solutions. The limitations that have been found when working with Forth will also be discussed, outlining the problems that I have encountered while developing the compiler to Forth, as well as the runtime library used to support the execution of the Forth code.

5.1 Bench Marking

While not all functions can be compiled to Forth and run, it is still valuable to measure Forth's performance when running the functions it can run. The following section will compare the compiled Forth code to three other subjects: Haskell, Scheme and a version of the target Agda code handwritten in Forth by myself. I chose to include handwritten Forth code to identify the difference between the performance of my compiler with an 'optimal' solution. The handwritten Forth code will have many benefits over the compiled code; it will be eagerly evaluated, removing the need for thunks, and it will have tail call optimisation, meaning that it can handle a recursive function of arbitrary size. To improve the reliability of my results, the run times displayed for each of the inputs are the average of ten separate runs.
The first test is the consume function, which simply takes a number and recursively decreases that number until it reaches zero. The expected complexity of this function is linear, however, the X-axis for the following graphs are on a logarithmic scale for the sake of readability, so at x = 3, the input value to consume is 8. While this function may seem trivial, simply taking a large number and always returning zero, in the end, it does serve a useful purpose by testing each of the versions' ability to handle simple recursive functions with very large depths.



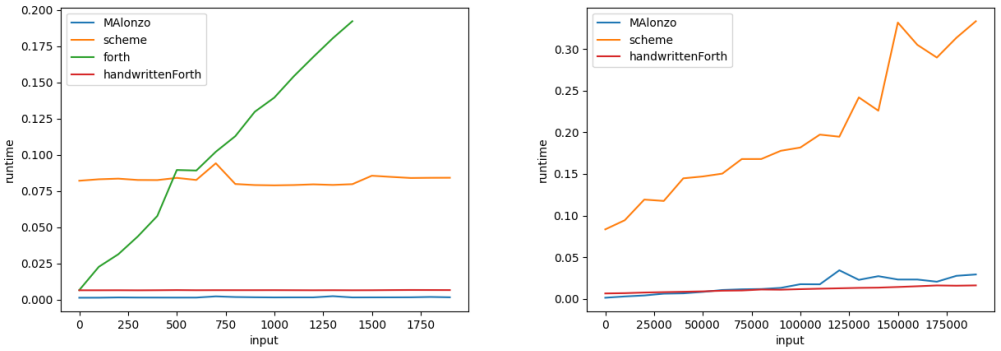
(a) Consume for inputs between 0 and 15

(b) Consume for inputs between 0 and 25

Fig. 12. Run times of the consume function

Figure 12 Shows the run times of the four different versions of the consume function for inputs between 0 and 25, with Figure 12a being between 0 and 15, and Figure 12b being between 0 and 25. 12a stops at 15 since, past that point, the compiled Forth version reaches a dictionary overflow, causing it to crash. In the case of all languages, the run time seems to not change for most of the inputs. An explanation for this is that the overhead time for the languages to start is much greater than the time it takes to actually evaluate the function, making the change in time unnoticeable. Two differences between the languages can be observed: the first being that the overhead of Forth and Haskell is considerably lower than that of Scheme, and the second being that time taken to evaluate consume becomes noticeable much earlier in compiled Forth, with Forth's run time starting to visibly increase at an input of 13, whereas Schemes run time doesn't visibly change until an input of 18. and Haskell and handwritten Forth don't show an increase until 23. It is worth noting that an input of 18 is 2^5 times larger than an input of 13. The graphs clearly show that both the Forth and Scheme compilers are not able to handle nearly as many levels of recursion as Haskell or the handwritten Forth solution. Even without being able to see the run time for larger inputs in compiled Forth, the difference in run time between it and the other can easily be predicted given how early its runtime starts to increase compared to the other versions.

The second function being compared is quick sort, which typically has a time complexity of $nlog(n)$. This is to test how effectively each implementation handles increasingly more complex, data types, in this case, an array, which is defined recursively in Agda. For Forth to be able to run this function, the version of quick sort used here makes use of two other comparison functions in place of the lambda expressions typically used for filtering the input list.



(a) Quick sort for inputs between 0 and 2000

(b) Quick sort for inputs between 0 and 200000

Fig. 13. Run times of the quick sort function

Similarly to the run times shown for consume, the compiled Forth code performs the worst of the four versions, both in runtime and maximum input. Figure 13a run times for the quick sort code for inputs up to 2000, the compiled Forth code is only able to sort a list of length 1300 before encountering a dictionary overflow, whereas Figure 13b shows that the other versions are not only able to run quick sort for an input list with length 200000, but they can all perform the task in less time than it took the compiled Forth code to sort a list of length 1300, other than Scheme, who's run time is shown to increase at a much greater rate than the other two versions still running at the higher input lengths. These two shortcomings, low maximum input and poor performance,

are both indicative of the compiled Forth's representation and handling of large data structures being very inefficient. On the other hand, the handwritten Forth implementation of quick sort can maintain a performance similar to Haskell's and even starts to surpass it for input lengths of 80000 and more. The handwritten implementation takes a list defined as a series of adjacent memory addresses making up each index, instead of the recursively defined definition used by the other three versions here. Figure 13 clearly shows that this is the more efficient way of handling such data structures in Forth.

5.2 Limitations

Agda and Forth are wildly different languages, and it seems unavoidable that these differences lead to limitations when executing Forth code structured with Agda in mind. The greatest limitation I have found is Forth's inability to implement tail call optimisation. Tail call optimisation is a way of reducing the number of nested function calls in situations where the last action made in a function is making a call to another, potentially itself. This optimisation is achieved by replacing this final function call with a 'jump' to the other function, removing the need to push a return address to the return stack, as there is no need to return to the function making the call afterwards. It is important to note that typical Forth words can be optimised for recursive tail calls through the use of the 'begin, again' control structure.

```
: FOO   ( n - 0 )
    dup 0 = if
        EXIT
    else
        1 -
        recurse  ( the word used to make a recursive call in Forth )
    then
;
: FOO   ( n - 0 )
    BEGIN
    dup 0 = if
        EXIT
    then
    1 -
    AGAIN ( jump back to the BEGIN word )
;
```

Fig. 14. The function 'foo' with (top) and without (bottom) recursive tail call optimisation

Figure 14 Shows two different versions of a recursive function, with the second making use of tail call optimisation to avoid making recursive calls and filling up the return stack. Unfortunately, Agda code - even Agda code that is originally tail-recursive - will not have this structure which allows for optimisation. This is due to the implementation of lazy evaluation and the 'dethunk' word; even if an Agda function is tail-recursive, the resulting Forth word would be evaluated with *mutual* recursion between the given function and the 'dethunk' function, meaning that at no point is a recursive call made. This serves to severely limit the size of problems that can be solved, as inputs to recursive functions that are too large will lead to return stack overflows.

Another limitation of Forth - or at least most standard implementations of Forth - as a language is its inability to support nested lambda expressions. This limitation and its effects were briefly discussed in Section 4.1, though to summarise what was said there, my solution to emulate the nested lambda expressions that make up many of Agda's functions, the compiled functions needed to be represented in the form of several different 'words' which are then passed to each other to form a single function. This results in even relatively simple Agda functions' Forth equivalents being very long and hard to read. With this lack of readability came a much more difficult debugging process, which greatly slowed down development progress. As such, not all Agda functions, when compiled to Forth, will be able to run correctly, or give correct results when they are run. While Forth's lack of support for nested lambda functions is a limitation of Forth, resulting in longer, harder-to-read code, it is arguable whether the resulting incompleteness of my compiler is a limitation of Forth as a language, or just a symptom of my unfamiliarity with writing and working with Forth.

6 RESPONSIBLE RESEARCH

As with any research, this report and investigation must be carried out responsibly. Since no parties are affected by the results of this work, or by the process through which the results were obtained, there is little room to conduct this research *irresponsibly*. That said, I could not have accomplished what I have in this report without the abundance of resources online regarding Forth, Agda and Dependently typed languages in general. Therefore, I should ensure that my findings and the work done for this report are also available online, for others to access if they need information about compiling dependently typed languages, specifically for Forth. To achieve this, my report, the poster used in my final presentation and the code base containing both the compiler and run time library will be available on public repositories.
The repository for my compiler and run time library can be found here: https://github.com/LMMilliken/agda2Forth, and the repository for the benchmarking can be found here: https://gitlab.ewi.tudelft.nl/jcockx/agda-extractors

7 CONCLUSION

Despite their differences, Forth has proven capable of supporting many features of Agda such as lazy evaluation, partial application, and a type system that enables pattern matching. That said, there is a difference between supporting these features and supporting them *well*. Many of the solutions to the problems posed by compilation from Agda are at the cost of the strengths of Forth; binding to local variables to allow words to take arguments sacrifices the low memory cost that comes with working with values on the stack, and while thunks can emulate laziness and partial application, they also prevent the tail call optimisation, limiting the size of inputs to recursive functions. On the other hand, an implementation with tail call optimisation could be written, with the caveat that the compiled Agda code would be evaluated eagerly, and without support for partial application. It is possible to write Agda code that does not require laziness or partial application, meaning that there would be a use case for this implementation with tail call recursion, but ultimately there are other target languages that can achieve both.

The notion that other target languages can achieve what Forth can more effectively is what leads us to the answer to the initial question, "How effectively does Forth serve as a target language for extraction from Agda?". While Forth can be used as a target language, its clear limitations when compared to other potential languages such as Haskell prevent it from being regarded as an 'effective' target for compilation. Even though there are several ways to improve the efficiency of the compiler and run-time library written for this project such as a more efficient method of generating constructors as described in Section 4.1, adding more builtin optimisations, or even

writing the compiler with a different implementation of Forth in mind, ultimately I believe that such improvements would not be beneficial enough to outweigh the limitations of Forth as a target language for compilation from Agda.

8 RELATED WORK

While much more feature-rich than Forth, there is another stack-based language currently being used as the target of a dependently typed language: Michelson[Michelson Team 2002], which is primarily used on the Tezos blockchain[3]. This language is "stack-based, with high-level data types and primitives, and strict static type checking"[Michelson Team 2002], and is the target language of compilation for the dependently typed language, Juvix[4]. While Michelson's additional features allow it to avoid some of the challenges I faced when compiling to Forth, there are still some similarities between the challenges faced, as well as the some of the solutions reached, when comparing compilation from Agda to Forth, and Juvix to Michelson [Metastate Team 2020].
The first of these is both Forth and Michelson's approach to functions and their application. As mentioned in Section 5.2, Forth is unable to support nested lambda functions, and as such the Agda2Forth compiler breaks down functions with multiple arguments into multiple single argument words. The Juvix to Michelson compiler also endeavours to have functions only take a single argument at a time, however for a different reason; Michelson functions only take a single function with the workaround for such a limitation being very expensive. To avoid such costs, two different solutions are proposed: "when we come across an exact application: we just inline it immediately . If we come across a partially applied function, we name the arguments on the stack for future reference, and push a virtual closure on the stack that can be later fully applied."[Metastate Team 2020]. While the second solution is very similar to my own 'pass' method for partial application, their method of inlining functions that have been fully applied is a very interesting one, however, it would not be as simple for me to implement, certain functions that contain lambda functions would still be impossible for me to inline, as they would still need to be defined elsewhere and passed into the function. That said, this solution would be possible in other cases if I were able to identify whether a function contains lambda expressions and compile-time, and treat their application differently.
Another problem faced when compiling to a stack-based language is how to handle variables that are now stored on a stack, rather than as a named variable. section 4.1 explains how my implementation binds arguments that have been pushed to the stack as local variables, however, the Juvix to Michelson compiler instead leaves arguments on the stack, and keeps track of their location to reuse with the 'dug' and 'dup' commands, which retrieve values from the stack and push a copy of the top item of the stack respectively. This method results in much less memory usage and eliminates the need to look up the value of a variable each time it is used, resulting in a faster and more efficient implementation than mine. I would have liked to implement such a solution, it would have taken too much time for the scope of this project.

While the comparison between my compiler and the Juvix to Michelson compiler may make Michelson seem like a more attractive choice for a stack-based target language than Agda, it comes with some severe drawbacks as well. Michelson does not support custom data types, meaning that code needs to be written with Michelson in mind to easily accommodate the data types used. This is a huge limiting factor, and severely limits the variety of uses that Michelson could have as a target language for Agda.

[3]https://tezos.com/
[4]https://juvix.org/

REFERENCES

2022. Chez Scheme code generator. *Chez Scheme Code Generator - Idris2 0.0 documentation.* https://idris2.readthedocs.io/en/latest/backends/chez.html

Agda Team. 2022a. Agda GHC Backend. *Compilers - Agda 2.6.2.1 documentation.* https://agda.readthedocs.io/en/v2.6.2.1/tools/compilers.html#ghc-backend

Agda Team. 2022b. Agda GHC Backend. *Compilers - Agda 2.6.2.1 documentation.* https://agda.readthedocs.io/en/v2.6.2.1/language/built-ins.html

Edwin Brady. 2005. *Practical Implementation of a Dependently Typed Functional Programming Language.* Ph. D. Dissertation. Durham University.

Bas Spitters Danil Annenkov, Mikkel Milo. 2021. Code Extraction from Coq to ML-like languages. In *ML-family workshop.*

Anton Ertl. 2008. GForth Manual. https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/index.html#Top

Peter J Knaggs. 1998. ANS Forth: Standardising. (1998).

Pierre Letouzey. 2008. 'Extraction in Coq: An Overview.' In Logic and Theory of Algorithms. In *Lecture Notes in Computer Science*, Vol. 5028. 359–369.

Metastate Team. 2020. Compiling Juvix to Michelson | PLT Type Theory RD at METASTATE. https://research.metastate.dev/juvix-compiling-juvix-to-michelson/

Michelson Team. 2002. Michelson: the language of Smart Contracts in Tezos. https://tezos.gitlab.io/active/michelson.html

Ulf Norell. 2007. Towards a practical programming language based on dependent type theory. (2007), 14.

Bill Stoddart and Robert Lynas. 2006. Adding Lambda Expressions to Forth. (2006).

Bill Stoddart, Robert Lynas, and Frank Zeyda. 2010. A virtual machine for supporting reversible probabilistic guarded command languages. *Electronic Notes in Theoretical Computer Science* 253, 6 (2010), 33–56.

John J Wavrik. 1990. Handling multiple data types in Forth. *Journal of Forth Application and Research* 6, 1 (1990), 65–76.