Code extraction from Agda to HVM

Matteo Meluzzi
Supervisors: Jesper Cockxs, Lucas Escot
EEMCS, Delft University of Technology, The Netherlands

June 17, 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 Agda to HVM

MATTEO MELUZZI*, Delft University of Technology, The Netherlands
JESPER COCKX, supervisor
LUCAS ESCOT, supervisor

Dependently typed languages such as Agda have the potential to revolutionize the way we write software because they allow the programmer to catch more bugs at compile time than classical languages. Nonetheless, dependently typed languages are hardly used in practice. One of the reasons is the lack of mature compilers for them.

This paper describes the implementation of a new Agda compiler that targets the Higher-Order Virtual Machine (HVM). Firstly we outline the theoretical benefits of using an optimal functional language such as HVM. Secondly, we present the problems we faced and the solutions we devised in the implementation of the agda2hvm compiler. Lastly, we compare our implementation to the current best Agda compilers by running benchmarks and analyzing both time and space performance. We obtained results ranging from our compiler being exponentially faster than the state-of-the-art to being exponentially slower.

Additional Key Words and Phrases: Dependent Types, Compiler, Code Extraction, Agda, Interaction Nets

1 INTRODUCTION

In the XXI century, many aspects of our lives depend on software. Making sure that it is correct and bug-free is thus a crucial problem. In recent years computer scientists have developed a new family of programming languages called dependently typed programming languages which have the potential to be part of the solution to this problem. Some languages that are in this family are for example Agda [Norell 2007], Coq [Barras et al. 1997], Idris [Brady 2013], Cayenne [Augustsson 1998], and Twelf [Pfenning and Schürmann 1999]. These languages allow programmers to mathematically prove properties about code using induction and assign additional constraints to the type system to catch more errors at compile time. Nonetheless, according to the research presented in [Hausmann et al. 2015] most of these systems focus on theorem proving rather than efficient execution and very few have a mature compiler. As a result, these languages are hardly used in practice.

This paper focuses on the implementation of a compiler for the Agda language using the HVM language [Taelin 2022] as a core language. At the time of writing there are four main implementations of compilers for Agda:

Agda to Haskell (MAlonzo) [Benke 2007] uses unsafe typecasts to solve the issues which arise from the fact that Agda's type system is more powerful than Haskell's. It relies on GHC to perform code optimizations.

Agda to Epic [Fredriksson and Gustafsson 2011] "is the most ambitious existing Agda compiler in terms of optimizations, and clearly shows the viability of compiling Agda to an untyped core language. Sadly, it has not been actively maintained and does not support all current Agda features." [Hausmann et al. 2015, p. 17]

Agda to JavaScript [Jeffrey 2013] is intended as a tool to write better web and GUI code, not as a tool to generate Agda binaries since JavaScript is an interpreted language.

Agda to UHC [Hausmann et al. 2015] transpiles Agda to a modified version of the Utrecht Haskell Compiler's core language. However, in [Hausmann et al. 2015] the authors state that this compiler does not outperform the MAlonzo backend because the Utrecht compiler does not perform optimizations.

Authors' addresses: Matteo Meluzzi, Delft University of Technology, Van Mourik Broekmanweg 6, 1234AB, Delft, The Netherlands, M.Meluzzi@student.tudelft.nl; Jesper Cockx, supervisor, JGHCockx@tudelft.nl; Lucas Escot, supervisor, lucas.escot@ens-lyon.fr.

Despite many attempts at writing a good compiler for Agda, better results might be obtained using a different core language. In this paper, we will assess whether the High-Order Virtual Machine (HVM) is a good candidate. We picked it because its author Victor Taelin claims that HVM achieves optimality by implementing Interaction Nets [Lafont 1989] to perform computations which can make it asymptotically faster than most functional runtimes in some cases but also introduces a run-time overhead.

This paper aims to test these claims and determine precisely how HVM performs compared to other core languages. It is structured in the following way:

- We give some background information on Agda and HVM.
- We describe the compilation from Agda to HVM.
- We present the results of the benchmarks of the different backends.
- We report our conclusions and our ideas for future work.

2 BACKGROUND

Agda and HVM are quite peculiar languages: the former is a dependently typed language and the latter is based on Interaction Nets.

In [Brady 2005] Edwin Brady states that the characteristic feature of dependent type systems is that types can be predicated on values which allows the programmer to give a more precise type to a value with the effect that more errors can be caught at compile time. Since Agda is a dependently typed language it has the potential to revolutionize the way we write software, leading to more robust and safe implementations through type-driven development [Brady 2017].

In [Asperti 2017] Andrea Asperti discusses the problem of sharing computations, showing that both innermost and outermost reduction strategies lead to duplication of work. To mitigate this problem, Haskell, which implements outermost reduction, uses thunks that are essentially a form of memoization and lead to more efficient reductions at the cost of using more memory. While this implementation is quite efficient, it is not optimal because there are cases where even Haskell duplicates work. In particular, Haskell does not share computations inside lambdas. Interaction Nets are a computational model developed over the years since 1989 [Lafont 1989; Lamping 1989] which allows for optimal reduction, so no work is ever duplicated. The problem with Interaction Nets is that they require bookkeeping work, which leads to less efficient code even if it is theoretically optimal.

2.1 Agda

The rest of this section will present a sample case where dependent types are useful.

Agda and many other dependently typed languages use the Peano representation of natural numbers because it allows for simple yet effective type constraints as we will see in example 2. In this representation every number is defined as the successor of another number:

Example 1. Peano representation of natural numbers

```
data Nat : Set where
   zero : Nat
   suc  : Nat → Nat

zero ⇒ 0
suc zero ⇒ 1
suc (suc zero) ⇒ 2
suc (suc (suc zero)) ⇒ 3
...
```

With this representation of numbers, it is now possible to distinguish between an empty vector and a non-empty one by indexing their type on the list length (example 2).

Example 2. Finite size list

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

Since an empty vector and a non-empty one have different types it is for example possible to define a safe *head* function without the need for *Maybe* in the return type because the compiler will reject programs that call *head* on *Vector A zero* instead of *Vector A (suc n)* (example 3).

Example 3. Safe head function

```
head : {A : Set}{n : Nat} → Vector A (suc n) → A
head (x :: xs) = x

head (1 :: 2 :: []) ⇒ 1
head [] ⇒ Type Error
```

From a performance perspective, indexed types can potentially introduce an overhead if the compiler is not optimized. This is the case because an instance of *Vector* requires its length as an additional parameter, even though it is not used at run-time but only during type-checking. The overhead can be mitigated by erasing the unused arguments. This optimization is called forcing and it is discussed in more detail in [Brady et al. 2004].

2.2 HVM

HVM is a functional programming language with a very simple syntax. It is compiled to C, which makes it quite fast since C compilers perform a multitude of code optimizations. It is lazily evaluated and supports lambdas and pattern matching. It does not have a type system, which simplifies the compilation from Agda since the problem of "converting" a dependently typed system to a statically typed system is avoided while maintaining type safety since Agda type checks the program before the compilation to HVM. Another feature of HVM is that it performs computations and memory management with Interaction Nets [Lafont 1989] which eliminate the need for a garbage collector, allow it to automatically parallelize work, and share computations inside lambdas. If the reader is interested in learning about Interaction Nets, how they are related to lambda calculus and how they can be manipulated to reduce a program to its normal form we recommend reading paper [Hassan et al. 2010]. Despite having multiple benefits, HVM also has disadvantages that are discussed in section 4.4.

Example 4. An HVM program to calculate prime numbers

```
(If 1 t e) = t
(If 0 t e) = e

(In (Cons y ys) x) = (If (== x y) 1 (In ys x))
(In Nil x) = 0

(ListDiff (Cons x xs) ys) = (If (In ys x) (ListDiff xs ys) (Cons x
    (ListDiff xs ys)))
(ListDiff Nil ys) = Nil
```

```
(Range bot top step) = (If (< bot top) (Cons bot (Range (+ bot step) top
    step)) Nil)

(Sieve m (Cons x xs)) = (Cons x (Sieve m (ListDiff xs (Range x m x))))
(Sieve m Nil) = Nil

(Main n) = (Sieve n (Range 2 n 1))
```

3 METHODOLOGY

This research is comprised of an initial literature survey on existing Agda backends and on the concept of Interaction Nets upon which HVM is implemented, followed by the implementation of a new backend targeting the HVM language and consequent benchmarks and comparisons with the existing ones.

The implementation of the new backend is open source and can be found on GitHub[1]. We have largely taken inspiration from Jesper Cockx's implementation of a Chez Scheme backend [Cockx 2022] and partly from the MAlonzo compiler which can be found in the official Agda repository [Benke 2022]. These three compilers follow the same implementation design: they are written in Haskell and they are built around the Agda Language Library [Norell and Team 2022]. This library provides a parser and some additional transformations of the input Agda program that are described in section 4, and it is designed to facilitate the implementation of new backends through user-provided hooks.

The benchmark programs used to test our implementation are incorporated in the *agda2HVM* repository and are discussed in further detail in section 5. To automate the benchmarking we have written a Python script that runs two executables with multiple inputs and plots the execution time and another script to measure the memory usage.

4 COMPILING AGDA TO HVM

The task of transforming Agda code into machine code is divided into multiple steps (figure 1). Agda provides the initial steps of parsing and type-checking, and outputs *Agda Internal Syntax*. This internal representation can be transformed into an even simpler representation called *Agda Treeless Syntax* with the function *Agda.Compiler.toTreeless* at the cost of losing the information about types and names. Agda revealed itself to be quite straightforward to get compiled to HVM, but we also encountered some inadequacies that are explained at the end of this section.

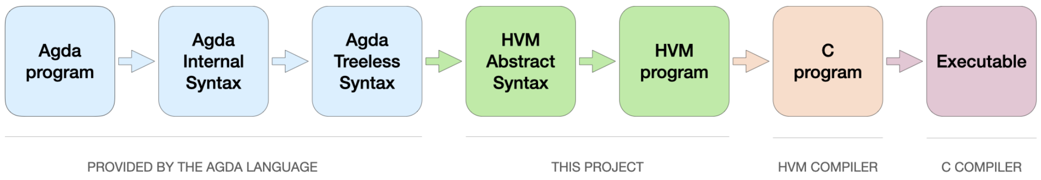

Fig. 1. The Agda compilation pipeline

[1]https://github.com/judomat/agda2hvm

4.1 Agda Internal Syntax

The *Agda Internal Syntax* represents a definition of a construct. Listed below are the most important constructs and a brief explanation of how they are compiled to HVM.

- **Axiom**: it is the simplest definition because it does not have a body. They are introduced with the *postulate* keyword.

  Example 5. An axiom definition

  ```
  postulate
      A : Set
  ```

  Postulates should only be used during type-checking and their evaluation at run-time causes an error. Thus their compilation yields no counterpart in HVM.
- **Function**: a function definition is first transformed to *Agda Treeless Syntax* and then compiled to HVM with this simpler representation.

  Example 6. A function definition and its HVM counterpart

  ```
  Agda:
  if_then_else_ : {A : Set} → Bool → A → A → A
  if true then x else y = x
  if false then x else y = y
  ⇒
  HVM:
  (If_then_else__0) = (@a (@b (@c (@d (If_then_else__4 a b c d)))))
  (If_then_else__4 a b c d) = (If_then_else__split_b_4 a b c d)
     (If_then_else__split_b_4 a (False) c d) = (d)
     (If_then_else__split_b_4 a (True) c d) = (c)
  ```

  As shown in example 6, the HVM compiler produces two definitions of the if_then_else_ function. This is because HVM does not support currying of top-level definitions but only of lambdas, so we wrap *If_then_else__4* in the lambda *If_then_else__0* to solve this issue.
- **Constructor**: Since in HVM there is no difference between a constructor and a top-level function, we were able to compile Agda constructors almost in the same way as functions. Except that in this case, the body of the definition is the name of the Agda constructor. While this may be obscure at first, it becomes clear with an example:

  Example 7. A constructor definition and its HVM counterpart

  ```
  Agda:
  data List (A : Set) : Set where
    Nil : List A
    Cons : A → List A → List A
  ⇒
  HVM:
  (Nil_0) = (Nil)

  (Cons_0) = (@a (@b (Cons_2 a b)))
  (Cons_2 a b) = (Cons a b)
  ```

Like in the case of the function definition, to support currying we wrap the constructor in a lambda.

4.2 Agda Treeless Syntax

The *Agda Treeless Syntax* represents a term of an Agda program. The majority of Agda terms have direct correspondence with HVM terms. Only the compilation of case splitting is achieved indirectly. Examples of terms with direct correspondence are *variables*, *primitive operations*, *definitions*, *function application*, *lambdas*, *literals*, *constructors*, and *let expressions*.

Case split: an Agda case split tree is specific to only one variable and is comprised of multiple alternative branches and a fallback branch.

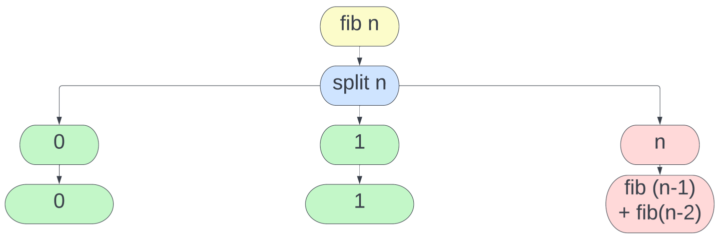

Fig. 2. A case split for the fib function. In blue the variable to split on, in green the alternative branches, and in red the fallback branch

We say that the correspondence is indirect because pattern matching in HVM can only happen in top-level definitions, so we had to lift the case-split terms to top-level (example 8).

Example 8. Case split lifting of fib function

```
Agda:
fib : Nat → Nat
fib 0 = 0
fib 1 = 1
fib (suc n@(suc m)) = fib n + fib m
⇒
HVM:
(Fib_0) = (@a (Fib_1 a))
(Fib_1 a) = (Fib_split_a_1 a)
   (Fib_split_a_1 0) = 0
   (Fib_split_a_1 1) = 1
   (Fib_split_a_1 a) = let b = (Fib_b_1 a); (+ ((Fib_0) (- (a) 1))
    ((Fib_0) (b)))
   (Fib_b_1 a) = (- (a) 2)
```

Using this case-split lifting technique we were able to achieve the correct functionality even when case splitting on multiple variables and nested constructors.

4.3 Optimizations

This research aims to determine whether HVM is a viable core language for Agda, concentrating more on the asymptotics of the running time and memory consumption rather than head-to-head efficiency comparison. The main reason is that the compiler targetted by MAlonzo, GHC, is a mature compiler developed by hundreds of people over many years while HVM is a prototype developed in a few months. It was never our goal to implement as many optimizations as GHC. Instead, we focused on one optimization that if not implemented would affect the running time asymptotics. This optimization is called *primitive data representation* and it concerns the way numbers are represented in the executable code. The default way Agda defines numbers is through the Peano representation for natural numbers presented in example 1. This representation is not space-efficient since the space required to store a number m is O(m). Furthermore, primitive operations such as addition or multiplication are also not efficient because they require linear time. By compiling natural numbers to primitive machine integers, the bounds become constant space for representation and constant time for operations.

4.4 Inadequacies of HVM

The fact that HVM is still a prototype manifested itself on several occasions. While some problems were due to missing features, others arose from bugs in the very implementation of the HVM compiler (version 0.1.24). It is worth noting that HVM comes with both an interpreter and a compiler, but for the sake of this project, we only used and tested the compiler.

- **Floating-point numbers** are not supported and cause a parse error.
- **Unbounded integers** are not supported. The maximum size of an integer in HVM is 32 bits. In case of an overflow, HVM does not report the error but instead silently wraps around.
- **Foreign Function Interface with C** is not supported. This severely limits the possibility of using HVM in practice since it is impossible to perform real-world IO tasks such as opening files and communicating over sockets.
- **Not all programs run as expected** even if the code is syntactically correct. The smallest program we could find to showcase this is shown in example 9. In this example, the issue seems to be variable *b* in the second line. Removing it makes the program behave as expected.

Example 9. Execution of this code outputs wrong result

```
(Append Nil ys) = ys
(Append (Cons x xs) ys) = let b = xs; (Cons x (Append xs ys))

(Main) = (Append [0, 1, 2] [3, 4, 5])
⇒
Output: Nil
Expected: [0, 1, 2, 3, 4, 5]
```

- **Running time** is sometimes asymptotically worse than Haskell. We were not expecting this to be possible since HVM is advertised as an optimal reduction machine. We give more details about this in section 5.3.
- **Memory usage** is sometimes asymptotically worse than Haskell. In [Asperti 2017] the author refers to his implementation of an optimal language (BOHM) [Asperti et al. 1996]) based on Interaction Nets and states that the main problem in the implementation was memory consumption. The author attributes the cause of this issue to the relation between time and space asymptotics where often improving the former worsens the latter and vice-versa. We give more information on which programs cause this difference in memory consumption in section 5.3.

5 BENCHMARKS

To determine in which cases HVM is better or worse than other Agda core languages, we have chosen to compare the HVM backend against the MAlonzo Haskell backend, Jesper Cockx's Scheme backend, and in certain benchmarks against Andrea Asperti's BOHM [Asperti et al. 1996]. We selected the first backend because it is the most widely used Agda compiler, and the second one because its target language is used by other dependently-typed languages such as Coq [Barras et al. 1997] and Idris [Brady 2013]. Furthermore, the BOHM language was included because, like HVM, it implements optimal higher-order computational machines based on interaction nets so we use it as a control to check that they perform similarly. The BOHM benchmarks have been manually translated from Agda because BOHM is not targeted by any Agda backends.

5.1 Optimal benchmark

Algorithm: perform 2^n compositions of the identity function
Theoretical time complexity: $O(2^n)$

HVM:	Linear time
BOHM:	Linear time
Haskell:	Exponential time
Scheme:	Exponential time

The graph (figure 3) indicates that there is a difference in the asymptotics of the running time of the two families of programming languages: the 'classical' family (Scheme and Haskell) runs in exponential time with respect to n, whereas the family of optimal languages (HVM, BOHM) runs in linear time.

This result was expected since BOHM and HVM are designed to share computations inside lambdas and this benchmark was purposely aimed at testing this feature.

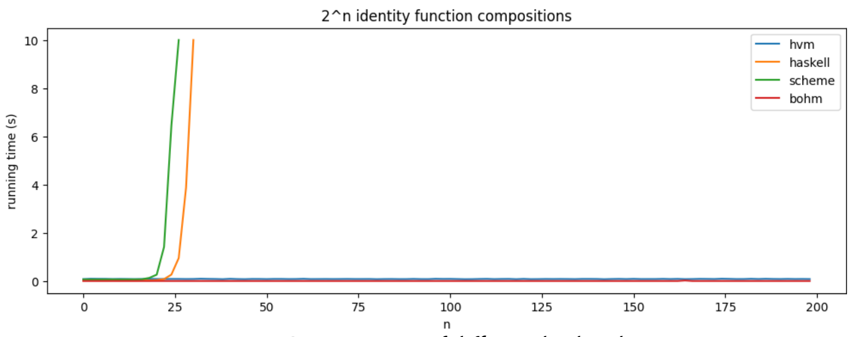

Fig. 3. Running time of different backends.
The program computes 2^n lambda compositions

5.2 Parallelizable benchmark

Algorithm: sum elements of a binary tree of size 2^n
Theoretical time complexity: $O(2^n)$

HVM:	Exponential time
Haskell:	Exponential time
Scheme:	Exponential time

As expected, all backends run in exponential time. HVM performs well and sometimes even beats Haskell. While one may think that this is due to HVM being more efficient than Haskell, this is not true. HVM performs well because this benchmark is parallelizable and HVM is designed to automatically take advantage of all CPU cores (6 on the test machine).

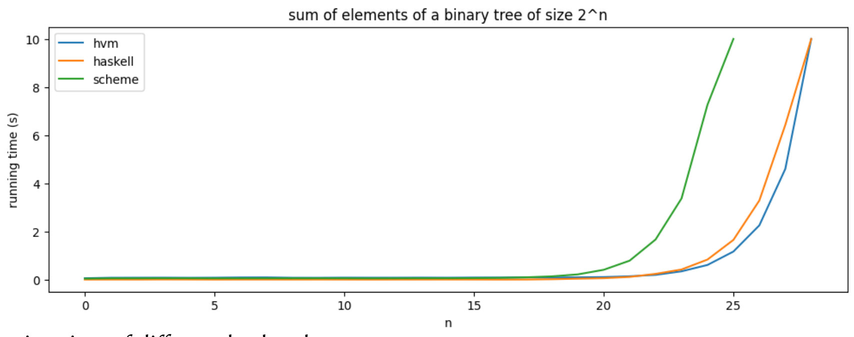

Fig. 4. Running time of different backends.
The program computes the sum of the elements of a binary tree with 2^n elements

5.3 Unexpected results

Algorithm: sum all triples (x,y,z) such that $x^2 + y^2 = z^2$ with $x, y, z < n$
Theoretical time complexity: $O(n^3)$

HVM:	Exponential time
BOHM:	Exponential time
Haskell:	Polynomial time
Scheme:	Polynomial time

We were expecting all four languages to have the same polynomial $O(n^3)$ running time for this benchmark. To our surprise, both optimal languages we tested were manifesting an exponential behavior (figure 5 left plot), to the extent that it was impossible to get a result after $n \approx 40$. After this boundary, HVM crashes and BOHM takes so long to complete that we had to kill the process.

To provide more evidence that the running time of HVM and BOHM is exponential, we plotted in figure 5 on the right plot the same data as on the left one but on a logarithmic scale and only for HVM. If the running time is exponential, the graph should show a straight line:

$$\begin{aligned} running\ time &= a^n \\ log_b(running\ time) &= log_b(a^n) \\ &= n \cdot log_b(a) \\ &\approx n \end{aligned} \tag{1}$$

The graph can in fact be approximated to a line, but only after $n \approx 20$.

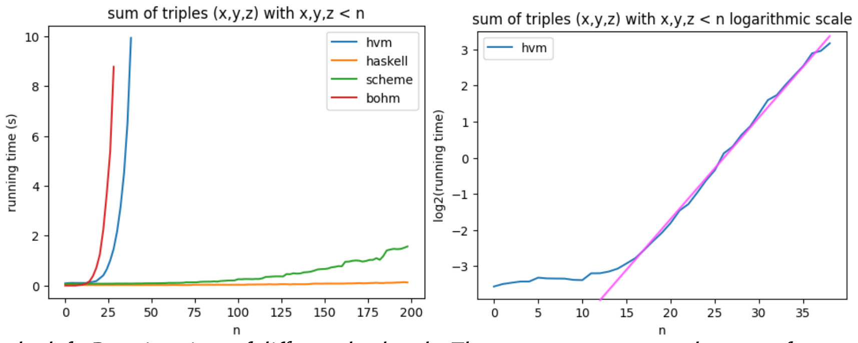

Fig. 5. On the left: Running time of different backends. The program computes the sum of cartesian triples (x,y,z) with x,y,z <= n
On the right: Running time of HVM on a logarithmic scale, fitted to a line

An initial hypothesis we formulated was that our compiler could have introduced some overhead that was negatively affecting performance. To test this hypothesis we manually translated the Agda code to HVM but obtained the same results. We thus decided to continue testing this program and measured the memory usage.

Theoretical memory complexity: $O(n^3)$

HVM:	Exponential memory
Haskell:	Polynomial memory
Scheme:	Polynomial memory

What we found was that HVM used an exponential amount of memory with respect to the input n, while Scheme and Haskell only required a polynomial amount (figure 6 left plot).

Like in the case of the time analysis we plotted the memory consumption on a logarithmic scale and overlapped it with a line to verify that the curve is exponential (figure 6 right plot). The result was strikingly similar to its time counterpart, with the line fitting after $n \approx 20$.

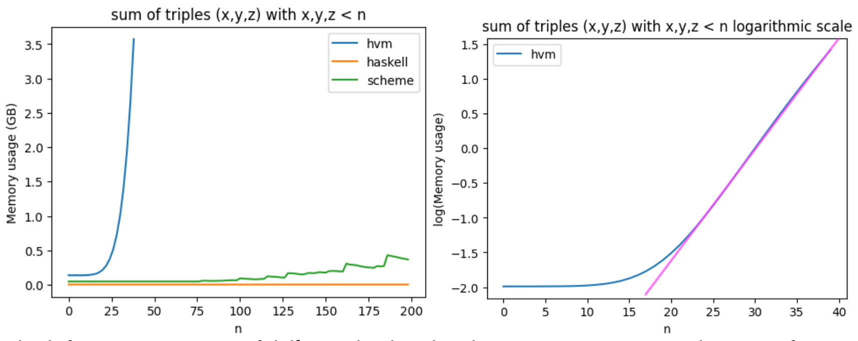

Fig. 6. On the left: Memory usage of different backends. The program computes the sum of cartesian triples (x,y,z) with x,y,z <= n
On the right: Memory usage of HVM on a logarithmic scale, fitted to a line

Despite our efforts, we do not have proof of what is causing this difference in performance. Instead, we could only formulate some hypotheses. One is that there could be something wrong in the implementation of BOHM, HVM, and possibly other optimal functional languages. Another one is that there could be an error in the theory behind these languages. Lastly, the exponential consumption of memory in this benchmark could be the cause of the exponential running time, so with infinite memory, the running time would actually be polynomial.

5.4 Interpretation

By running these benchmarks we have found that the main advantages of using HVM as a core language for Agda are:

- HVM can be asymptotically faster than Haskell when the target program performs heavy lambda computations.
- HVM parallelizes computations more often than Haskell, especially when operating on a tree data structure rather than a list.

And that the main disadvantages compared to the MAlonzo backend are:

- HVM is not stable yet and would undermine the safety of programming in Agda since unexpected HVM errors can happen at run-time.

- There are programs that if compiled to HVM perform worse than their theoretical time and memory complexity while the same program compiled to Haskell performs as expected. This is a serious problem whose causes could not be identified with certainty.

6 RELATED WORK

In section 1 of this paper, we listed the different backend compilers of the Agda languages. The most notable and used among them are the MAlonzo [Benke 2007] compiler, which targets the Haskell language and relies on GHC to produce the executables, and the EPIC compiler which targets the EPIC language and performs several optimizations but is no longer maintained and has been removed from the Agda source code. In addition to these backends, we also present the agda2scheme backend and BOHM optimal languages that are used in the benchmarks in section 5.

6.1 MAlonzo backend

The MAlonzo backend is the principal Agda compiler. Furthermore, it generally produces the fastest code since the Haskell compiler upon which it relies is heavily optimized. Still, this does not mean that there is no margin for improvement: in [Hausmann et al. 2015] the author states that this compiler overcomes the differences between the Haskell and Agda type systems using type coercions and that these coercions can lead to a blowup in the size of the generated Haskell code and can prevent GHC from applying certain type-directed optimization.

In example 10, such coercions are represented by the applications of the *coe* function, which is a wrapper around the *unsafeCoerce* function defined in the *Unsafe.Coerce* package.

One of the main advantages of the MAlonzo backend is that it provides a Foreign Function Interface (FFI) to Haskell. This FFI allows programmers to call Haskell functions from Agda, export Agda functions to Haskell, and reuse Haskell data types in Agda. This is essential to transform Agda from a proof assistant into an industry-ready programming language as it allows to reuse all the existing Haskell infrastructure. For instance, the Agda Standard Library [Community 2022] heavily relies on it to provide IO functions that directly interface with the operating system such as file manipulation.

Example 10. Hello World Agda program compiled with MAlonzo

```
main = coe d_main_2
-- HelloWorld.main
d_main_2 ::
  MAlonzo.Code.Agda.Builtin.IO.T_IO_8
    AgdaAny MAlonzo.Code.Level.T_Lift_8
d_main_2
  = coe
      MAlonzo.Code.IO.Base.du_run_88 (coe MAlonzo.Code.Level.d_0l_22)
      (coe
         MAlonzo.Code.IO.Finite.d_putStrLn_34
         (coe MAlonzo.Code.Level.d_0l_22)
         (coe ("Hello, World!" :: Data.Text.Text)))
```

6.2 Epic backend

The Epic backend [Fredriksson and Gustafsson 2011] targets the Epic[2] language, which is a core language specifically designed by Edwin Brady for dependently typed languages. Unlike HVM, it

[2]**Epi**gram **C**ompiler

is strictly evaluated and has type annotations, but they are not checked. Epic also has a Foreign Function Interface but instead of targeting a high-level language like the MAlonzo backend, it targets the C language.

What differentiates this backend from other ones is that it implements multiple code optimizations, which are mostly inspired by Edwin Brady's Ph.D. thesis [Brady 2005]. The most important ones are:

- **Forcing**: consists in removing unused arguments from functions and constructors. It is particularly important when compiling a Dependently Typed language where many arguments are used only during type-checking and not at run-time.
- **Primitive Data**: instead of representing numbers through the default Peano representation (example 1) which is not efficient, numbers are compiled to machine words to improve the running time and space asymptotical bounds (section 4.3).

Despite implementing all these features, this backend is no longer maintained and has been removed from the Agda language repository. For this reason, we decided not to use it in the benchmarks (section 5).

6.3 agda2scheme backend

The agda2scheme backend has been implemented by the supervisor of this research Professor Jesper Cockx. The target language of this backend is Chez Scheme [Dybvig 2009], which is a LISP-like untyped strict functional language. Although Scheme is strictly evaluated by default, this backend also supports lazy evaluation by applying the *delay* and *force* operators in the generated Scheme code. Like the Epic backend, agda2scheme implements forcing and primitive data representation.

6.4 BOHM language

In paper [Asperti et al. 1996], the authors of The Bologna Optimal Higher-order Machine (BOHM) state that their language is an implementation of Lamping's optimal graph reduction technique for reducing λ-expressions. The authors further explain that this technique consists in representing all syntactical operators as nodes in a graph and reducing the program with well-defined interaction rules among these nodes.

For these characteristics, BOHM is similar to HVM and could be considered its ancestor. Given the latter, we decided to include BOHM in some of our benchmarks (section 5) even though no Agda backend targets BOHM.

BOHM and HVM also have some differences, the main ones being that BOHM cannot be compiled but only interpreted, and unlike HVM it relies on a garbage collector to perform memory management.

7 REPRODUCIBILITY OF RESULTS

The source code of agda2HVM and the other backends we have used for benchmarking are open source and freely available:

Software	**Dependencies**
agda2HVM	Haskell Stack
agda2scheme [Cockx 2022]	Haskell Cabal
MAlonzo [Benke 2022]	Haskell Stack
HVM [Taelin 2022]	Rust compiler
BOHM [Andrea Asperti 2022]	gcc
Benchmark scripts	Python, GNU time, Agda Standard Library

We wrote every benchmark program in two slightly different versions: one without importing the Agda Standard Library that we compiled with the HVM and Scheme backends, and another with this import that we compiled with the MAlonzo backend. We use the Agda standard library to read the program arguments supplied by the benchmarking scripts. Since this library cannot be compiled with the HVM and Scheme backends, we use specific features of these languages to get the program arguments. This does not affect the benchmark results because apart from this difference in how the input is fetched the benchmark programs are identical.

8 CONCLUSIONS

We have implemented a compiler for Agda targeting the HVM language. We were able to compile programs that use dependent types without inserting type-casts thanks to the untyped nature of HVM and transform Agda constructs to HVM relatively easily due to HVM's combination of simplicity and functional programming features. We have tested our compiler against the state-of-the-art of Agda backends. Our HVM compiler can in some cases be much faster than the Haskell backend, in other cases it can match the performance, and we also found some programs for which it is asymptotically slower. This was surprising because HVM should at least in theory have the same asymptotical running time. We performed a multitude of tests but were unable to identify the causes. Further research is needed to investigate the effectiveness of optimal functional languages as a target language for compilers of dependently typed languages.

At its current state, our backend is not ready to be used for practical purposes given that HVM is still in its early development. Before HVM can be considered a viable and reliable target language it would need to become stable, support all primitive data like floating-point numbers and unbounded integers, and have a foreign function interface with C to allow low-level communication with the operating system. Lastly, the problem of unexpected exponential time and memory consumption would have to be studied in greater detail and be solved.

REFERENCES

Andrea Naletto Andrea Asperti, Cecilia Giovanetti. 2022. BOHM GitHub page. https://github.com/cls/bohm. [Online; accessed 29 May 2022].

Andrea Asperti. 2017. About the efficient reduction of lambda terms. https://doi.org/10.48550/ARXIV.1701.04240

Andrea Asperti, Cecilia Giovannetti, and Andrea Naletto. 1996. The Bologna optimal higher-order machine. *Journal of Functional Programming* 6, 6 (1996), 763--810. https://doi.org/10.1017/S0956796800001994

Lennart Augustsson. 1998. Cayenne—a Language with Dependent Types. *SIGPLAN Not.* 34, 1 (sep 1998), 239–250. https://doi.org/10.1145/291251.289451

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 1997. *The Coq proof assistant reference manual: Version 6.1.* Ph. D. Dissertation. Inria. https://flint.cs.yale.edu/cs430/coq/pdf/Reference-Manual.pdf

Marcin Benke. 2007. Alonzo—a compiler for Agda. *Talk at Agda Implementors Meeting* 6 (2007). Issue 4. https://www.mimuw.edu.pl/~ben/Papers/TYPES07-alonzo.pdf

Marcin Benke. 2022. MAlonzo GitHub page. https://github.com/agda/agda/tree/master/src/full/Agda/Compiler/MAlonzo. [Online; accessed 25 May 2022].

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Edwin Brady. 2017. *Type-driven development with Idris.* Simon and Schuster, New York, NY, USA.

Edwin Brady, Conor McBride, and James McKinna. 2004. Inductive Families Need Not Store Their Indices. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115--129. https://doi.org/10.1007/978-3-540-24849-1_8

Edwin C Brady. 2005. *Practical Implementation of a Dependently Typed Functional Programming Language.* Ph. D. Dissertation. Durham University. https://eb.host.cs.st-andrews.ac.uk/writings/thesis.pdf

Jesper Cockx. 2022. agda2scheme GitHub page. https://github.com/jespercockx/agda2scheme. [Online; accessed 25 May 2022].

Agda Community. 2022. Agda Standard Library. https://github.com/agda/agda-stdlib. [Online; accessed 25 May 2022].

R. Kent Dybvig. 2009. *The Scheme Programming Language, 4th Edition* (4th ed.). The MIT Press, Cambridge, MA, USA. https://www.scheme.com/tspl4/

Olle Fredriksson and Daniel Gustafsson. 2011. *A totally Epic backend for Agda*. Master's thesis. Chalmers University of Technology. https://hdl.handle.net/20.500.12380/146807

Abubakar Hassan, Ian Mackie, and Shinya Sato. 2010. A lightweight abstract machine for interaction nets. *ECEASST* 29 (01 2010). https://doi.org/10.14279/tuj.eceasst.29.416

Philipp Hausmann, Atze Dijkstra, and Wouter Swierstra. 2015. The Utrecht Agda Compiler. (2015). https://webspace.science.uu.nl/~swier004/publications/2015-tfp-draft.pdf

Alan Jeffrey. 2013. Dependently Typed Web Client Applications. In *Practical Aspects of Declarative Languages*, Kostis Sagonas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 228--243. https://doi.org/10.1007/978-3-319-51676-9

Yves Lafont. 1989. Interaction Nets. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 95–108. https://doi.org/10.1145/96709.96718

John Lamping. 1989. An Algorithm for Optimal Lambda Calculus Reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/96709.96711

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer, Göteborg, Sweden. https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

Ulf Norell and The Agda Team. 2022. Agda Language Library. https://hackage.haskell.org/package/Agda. [Online; accessed 6 June 2022].

Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf --- A Meta-Logical Framework for Deductive Systems. In *Automated Deduction --- CADE-16*. Springer Berlin Heidelberg, Berlin, Heidelberg, 202--206. https://doi.org/10.1007/3-540-48660-7_14

Victor Taelin. 2022. HVM GitHub page. https://github.com/Kindelia/HVM. [Online; accessed 25 May 2022].