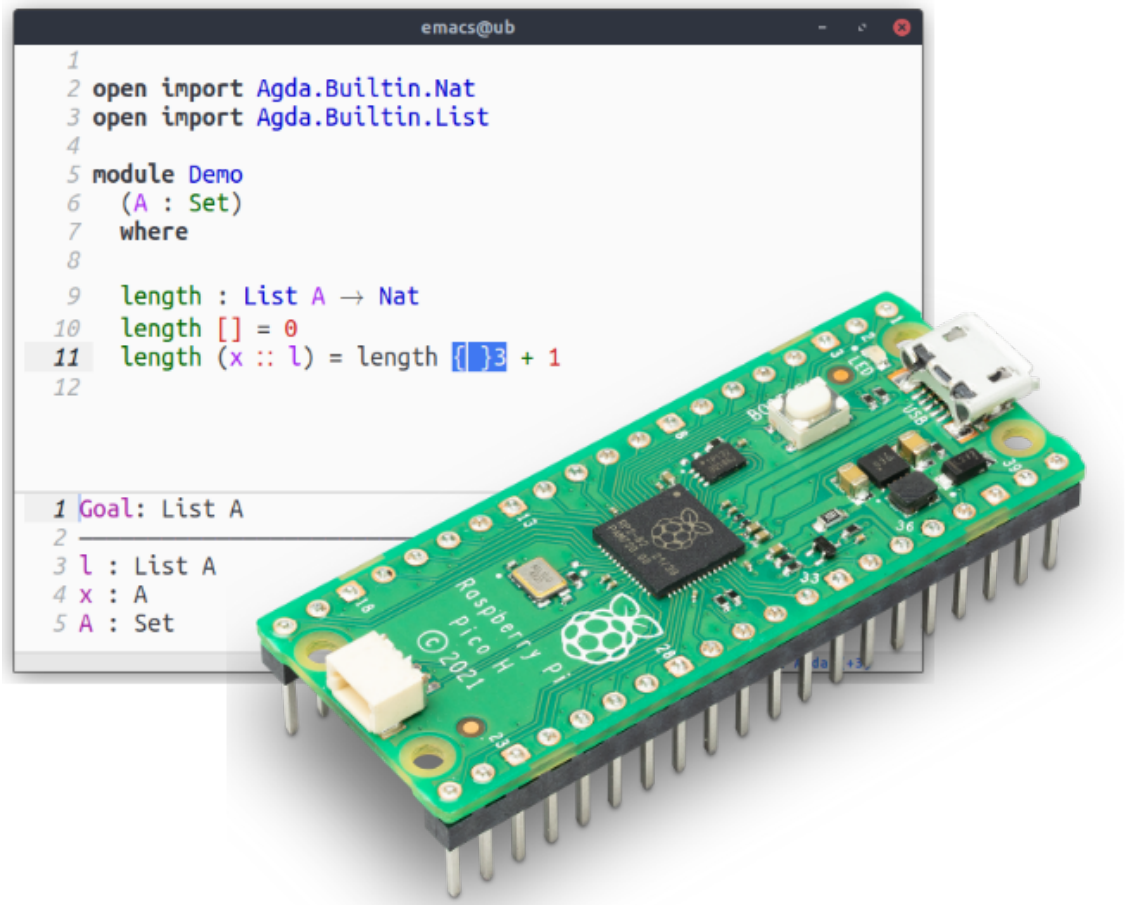
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Agda on Raspberry Pi

Master's Thesis in Computer Science and Engineering

Lawerence Chonavel

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Master’s thesis 2023

Agda on Raspberry Pi

Lawerence Chonavel

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Agda on Raspberry Pi
Lawerence Chonavel

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Examiner: Thierry Coquand, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Agda code in emacs, and the Raspberry Pi Pico microcontroller
Typeset in LaTeX
Gothenburg, Sweden 2023

Agda on Raspberry Pi
Lawerence Chonavel
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents an Agda-to-C compiler targeting the Raspberry Pi Pico microcontroller. The compiler implementation includes an unusual choice of run-time algorithm, a Foreign Function Interface generator, and surprisingly little boilerplate code.

Keywords: Agda Compiler Raspbery Pi Functional Programming BUBS

Acknowledgements

Thanks to my supervisor, Andreas Abel, and the other Agda developers.

Lawerence Chonavel, Gothenburg, June 2023

Contents

List of Figures

List of Tables

1

Introduction

Writing correct and maintainable software is difficult and expensive. Functional programming languages go a long way towards solving these problems[1], and in particular the Agda programming language[2] assists programmers with a wealth of advanced features including a powerful module system, flexible syntax, interactive editing, and a research-grade type system.

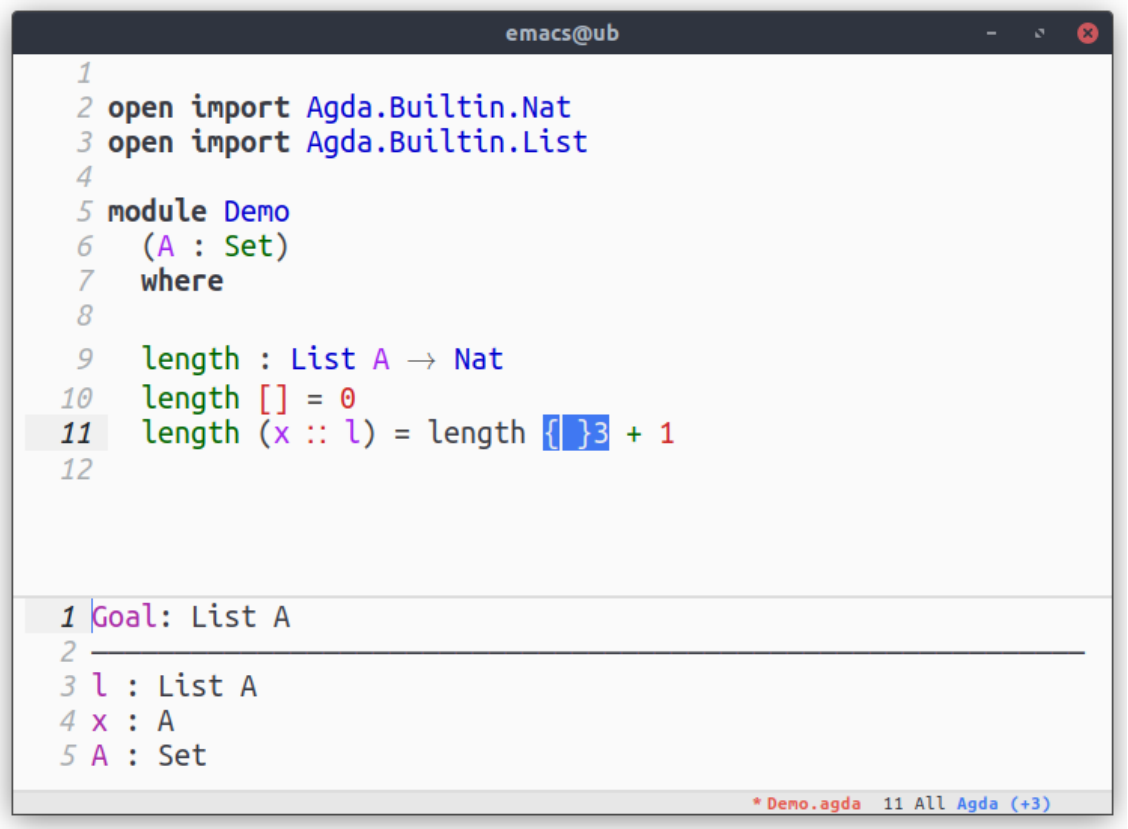


Figure 1: Interactive development with Agda

However, functional programming languages are rarely used in practice[1], perhaps due to myths such as "functional programs use megabytes of memory" or "functional programs need complex compilers" or "you can't predict the performance of functional programs".

[1] As of Oct 2022, none of the 20 most-searched-for programming languages[3] were functional.

This thesis debunks these myths, by presenting a small and simple compiler for a modern functional programming language, targeting a resource-constrained computer: the *Raspberry Pi Pico*, a low-cost microcontroller widely used in education and hobbyist electronics.

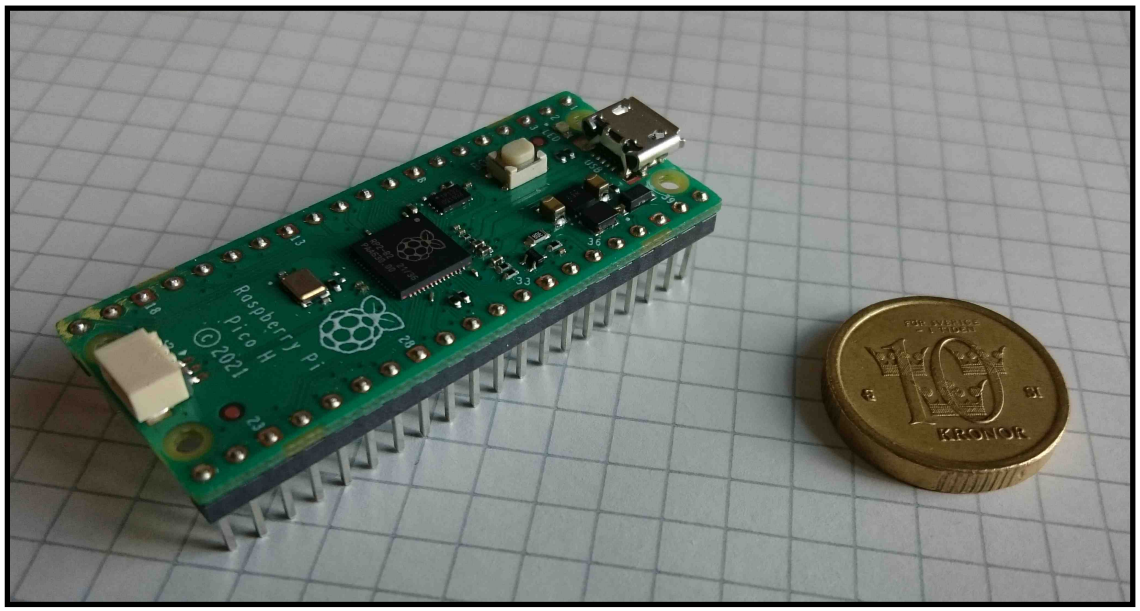


Figure 2: The Raspberry Pi Pico is not much bigger than a coin

It is equipped with 200 KiB of RAM, 2 MiB of Flash memory, two ARM Cortex M0+ processor cores, and assorted IO-related hardware. [4]

In other words, the Raspberry Pi Pico has about as much computing power as computer workstations did 30 years ago[2]. However, the Pico costs $5, fits in the palm of your hand, and has 30 digital IO pins that plug directly into an electronics breadboard.

Before this project, The Raspberry Pi Pico was already a compilation target for several high-level programming languages, including C, Rust and MicroPython. However, none of the previously supported languages[3] were purely functional like Agda.

1.1 Contributions

This thesis presents an Agda-to-C compiler (`agda2c`) capable of producing code for the Raspberry Pi Pico.

New & unusual parts of the implementation (§ 2) include:

[2] For example, the Raspberry Pi Pico's on-chip clock runs at 133MHz, which is a typical speed for processors released between 1992-1996[5].

[3] This blog post[6] summarizes the programming languages previously available on the Raspberry Pi Pico.

- A runtime library implementing (and extending) Shivers & Wand's Bottom-Up β-Reduction algorithm[7] for strong evaluation of the untyped λ-calculus (§ 2.3).
- A user-friendly (but safe and easily implemented) Agda EDSL for specifying a compiled Agda program's C interface (§ 2.4).
- A substantial compiler pass implemented in Agda itself (§ 2.5).

I also discuss compiler components (both by me (§ 5.5), and by other people (§ 4)) that *didn't* make it into `agda2c`, and explain why.

The compiler's source code is freely available online at `https://github.com/lawcho/agda2c`[8].

2

Compiler Design

The compiler is structured as a three-stage pipeline (fig. 3): the user's code is first passed to the *Agda frontend* (§ 2.1), which checks it and performs some translation, then to the *AST Compiler* (§ 2.2), which translates it again, then finally to *GCC* (§ 2.6), which translates it to machine code ready to be run by the Raspberry Pi Pico.

The Agda source code also includes *Foreign Function Interface (FFI) Descriptions* (§ 2.4), which are compiled separately (§ 2.5), then combined with the rest of the code by *GCC* (§ 2.6), along with a substantial *runtime library* (§ 2.3) which interprets the compiled program.

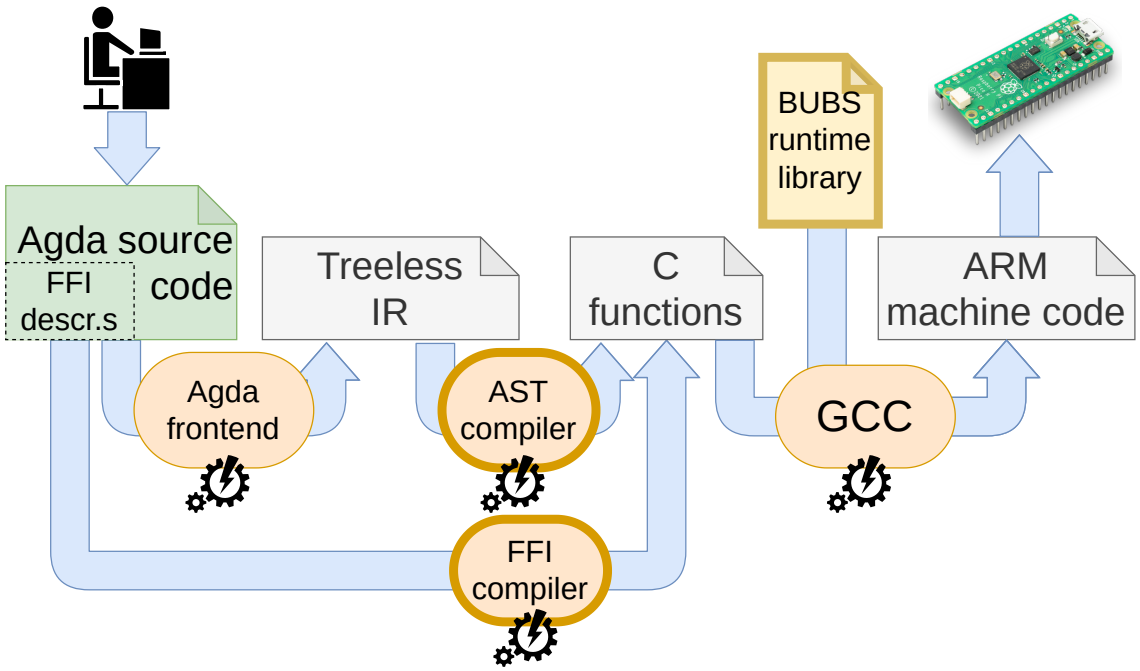


Figure 3: Pipeline for compiling Agda to ARM machine code

Many of these components are existing open-source software, but some are custom-built (thick outline). Let's look at each of them in more detail.

2.1 Agda Frontend

Agda source code is fed in to the *Agda frontend* (an existing component), which type-checks and simplifies programs (fig. 4).

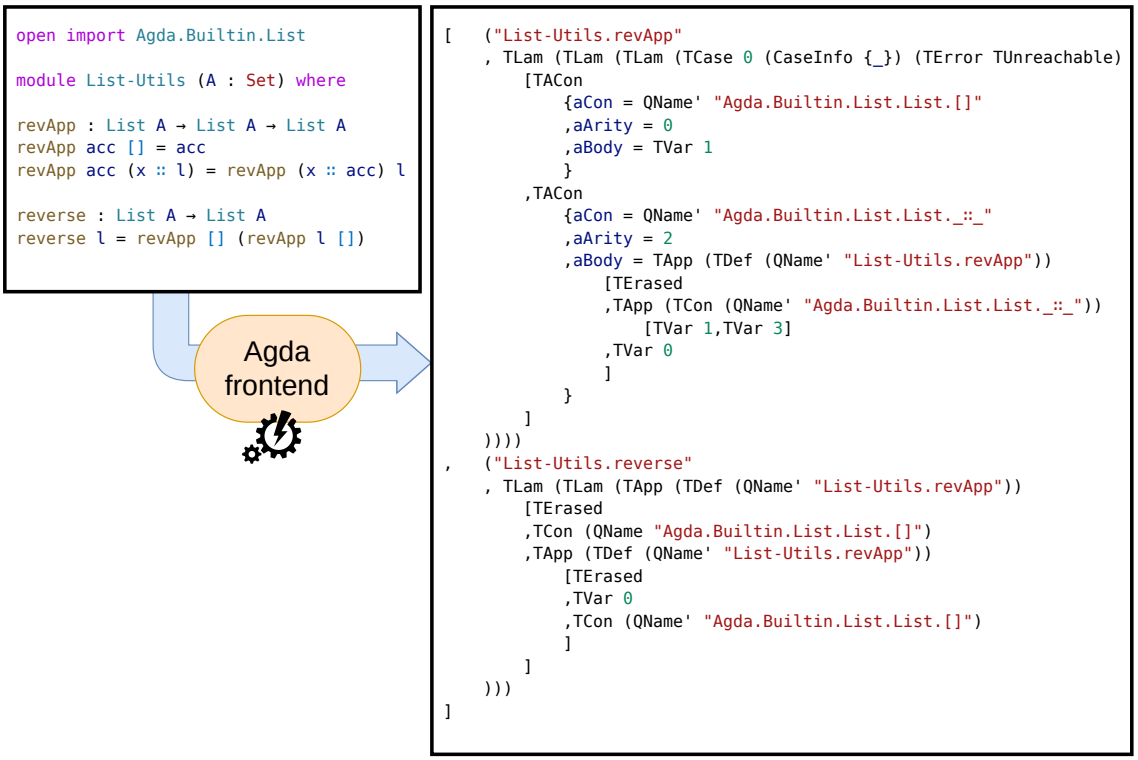


Figure 4: The Agda frontend compiles Agda to Treeless IR

The simplified programs may contain (for example) `λ`- and `case`- expressions, but no `module` declarations or type signatures. These programs form a language called *Treeless IR*, and are stored as Haskell data for easy further processing.

The Agda frontend is an existing open-source project[2] with extensive documentation for both the user-facing language[9] and many of the implementation's algorithms [10] [11] [12] [13], so I will not discuss it further.

2.2 AST Compiler

Next, Treeless IR programs are compiled to C by the *AST Compiler*[14] (fig. 5) (a custom-built component). In particular, the AST Compiler is responsible for:

- Encoding Agda names in a C-compatible way
- Removing data constructors and `case`-expressions
- Printing out Treeless expressions in C syntax

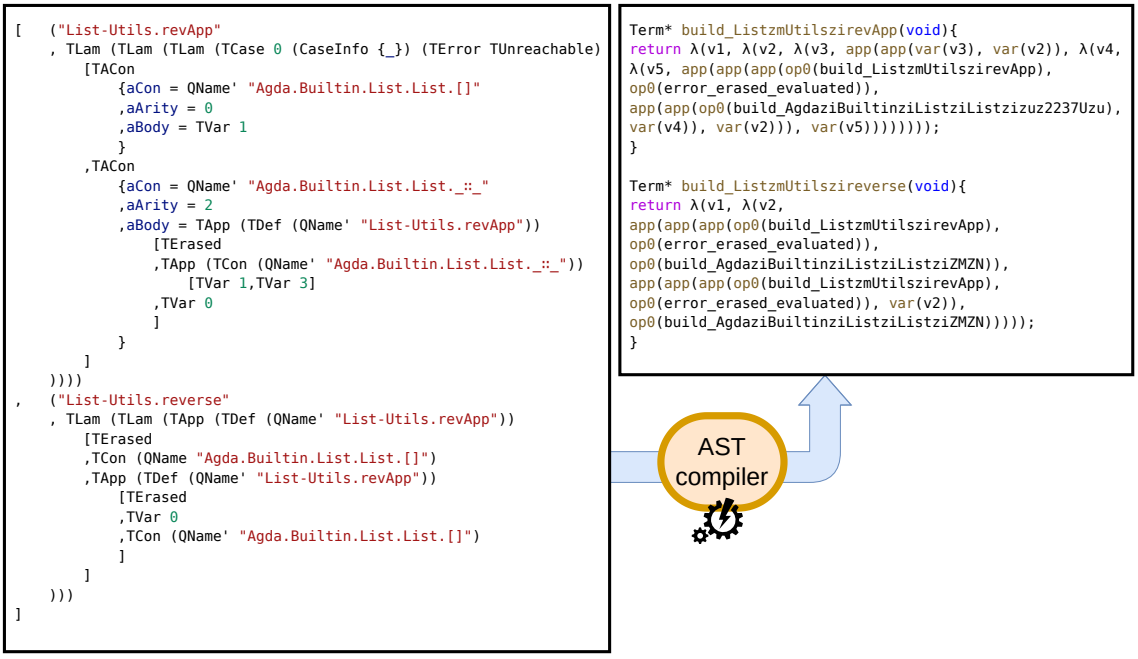


Figure 5: The AST Compiler compiles Treeless IR to C

None of these transformations are particularly advanced: names are made C-compatible by Z Encoding[15] (fig. 6), constructors and `case`-expressions are removed by Scott Encoding[16] (fig. 7), and printing is implemented by recursive descent.

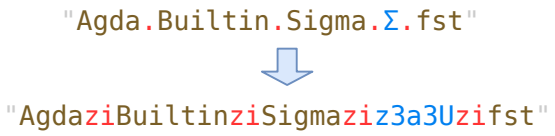


Figure 6: Z Encoding makes Agda names C-compatible

The AST Compiler gets away with doing only this small amount of work by relying heavily on other parts of `agda2c`: most of Agda's syntax is either removed by the Agda frontend (e.g. modules, implicit arguments, erased declarations) (§ 2.1), or passed on into the generated C code (e.g. λ-expressions, function application expressions) where it is evaluated at run-time by a library (§ 2.3), at which we shall look next.

```
data List (A : Set) : Set where
  nil : List A
  cons : (x : A) (xs : List A) → List A
```

```
nil  =          λ knil. λ kcons. knil

cons = λ x. λ xs. λ knil. λ kcons. kcons x xs
```

```
case l of λ where
  nil         → 0
  (cons x xs) → 1 + length xs
```

```
l (            0            )
  (λ x. λ xs.   1 + length xs)
```

Figure 7: Scott Encoding removes constructors and `case`-expressions

2.3 BUBS Runtime Library

As we just saw, the generated C code contains calls to C functions like `λ(_,_)` and `app(_,_)`. These functions are provided by a custom-built C library called the *BUBS runtime*, and are used to encode functional programs as graphs in memory (fig. 8).

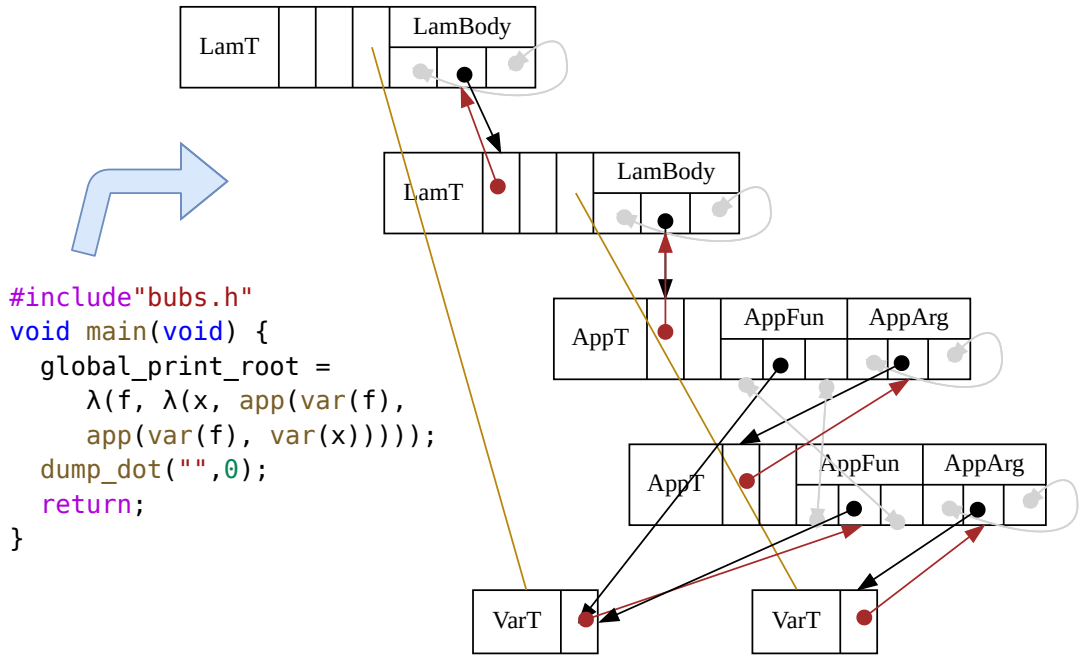


Figure 8: BUBS encodes the program `(λf.λx.f(f x))` as a graph

As well as providing the `λ(_,_)`, `app(_,_)`, and `var(_)` functions (which are enough to implement the entire *pure un-typed λ-calculus* functional programming language), the BUBS library additionally provides the functions `prim(_)`, `op0(_)`, `op1(_,_)`, and `op2(_,_,_)`, allowing for simple and efficient implementation of various language features, e.g. integer arithmetic/comparison, debug logging and recursion[1].

[1] Stand-alone examples of how to implement all these are available online in the

The BUBS library also provides a function `whnf(_)` which *evaluates* a program, by traversing its graph (fig. 9) and re-writing parts of it using simple *reduction* transformations. The algorithm exploited by `whnf(_)` has many convenient properties, such as evaluating programs *lazily*, preserving *sharing* of sub-programs, efficiently *collecting garbage* as it evaluates, and being remarkably simple. This algorithm is presented in detail by Shivers & Wand[7], who also provide a proof of correctness and a small (200 line) Standard ML implementation, which my C library is based on.

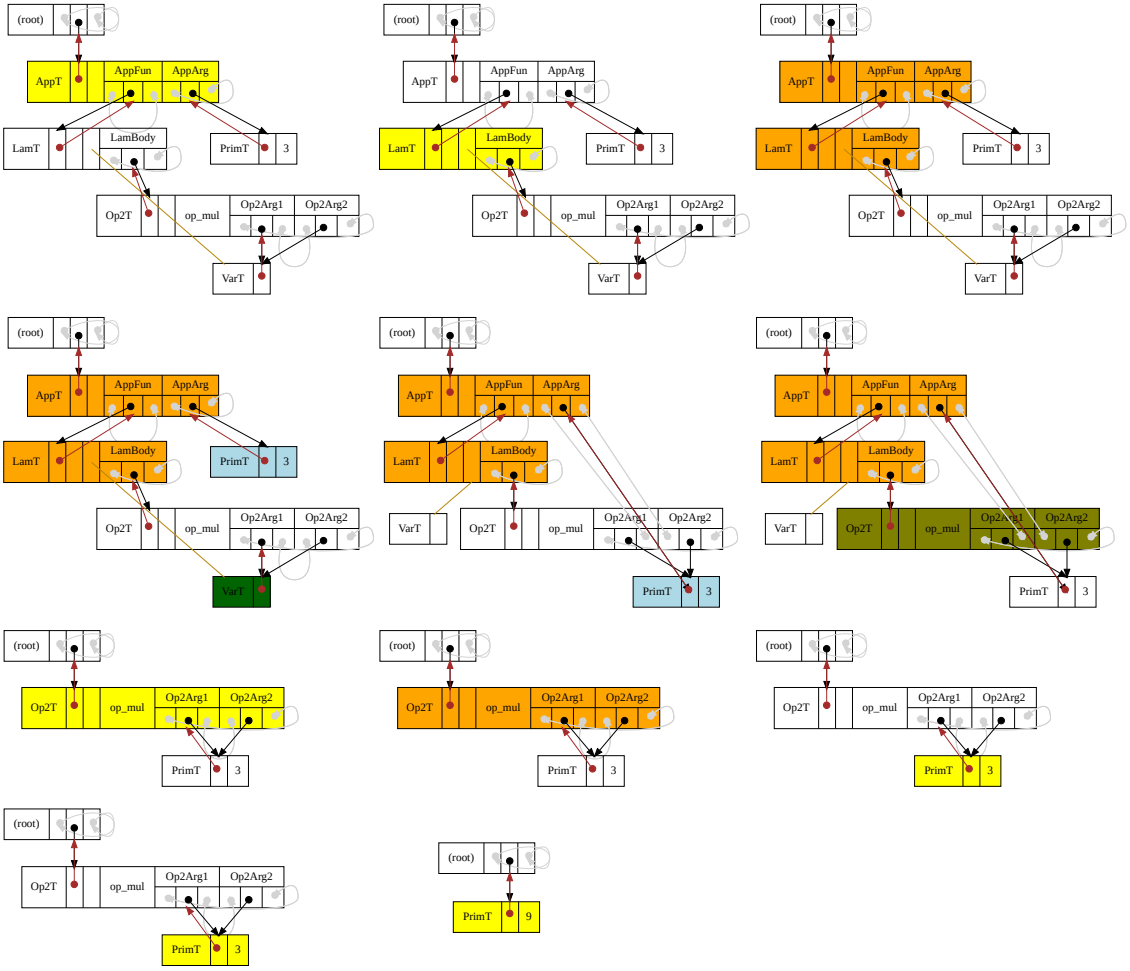


Figure 9: BUBS evaluates the program `(λx.x*x)3`

I have implemented the BUBS library as a stand-alone C library, and freely distributed it online[17], so that it can be used by third-party C programs to interface with the functions generated by the AST Compiler (§ 2.2).

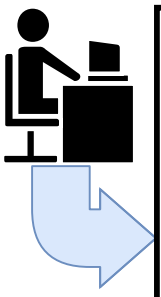
However, in practice, this sort of *Foreign Function Interface (FFI)* code is difficult to write and maintain: avoiding subtle bugs requires detailed knowledge of `agda2c`'s implementation, e.g. which function

BUBS library's GitHub repository[17].

parameters are erased by the Agda frontend, and how Scott-encoding affects data types (fig. 7). To lighten the burden on users, `agda2c` automates the generation of this low-level FFI code from high-level FFI descriptions (§ 2.4), which we shall look at next.

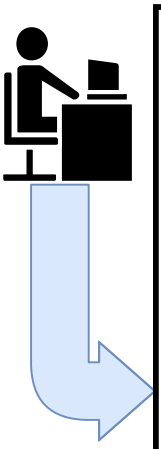
2.4 Foreign Function Interface (FFI)

Users can control the C interface of the compiled Agda code by writing `{-# COMPILE #-}` pragmas in the source code. Two types of pragma are supported: a `{-# COMPILE RAW_C #-}` pragma (fig. 10) tells `agda2c` to include some text in the generated code, whilst a `{-# COMPILE C #-}` pragma (fig. 11) tells `agda2c` to generate a C function following a specification given in an `Exposed` definition. For example, the `Exposed` definition `hello` (in fig. 11) compiles to a C function `void main ()`, with a body interpreting the stream of foreign function calls after `.function-body`.



```
c-import = "
#include<stdio.h>
#include<inttypes.h>
void put_u16(uint16_t x){
    printf(\"%c\",(char)x);
    fflush(stdout);
}"
{-# COMPILE RAW_C c-import #-}
```

Figure 10: A `{-# COMPILE RAW_C #-}` pragma



```
put-u16 = sig void "put_u16" (uint16 ∷ [])

hello : Exposed
hello .own-signature = sig void "main" []
hello .imported-sigs = put-u16 ∷ []
hello .function-body = do
    ccall put-u16 72
    ccall put-u16 101
    ccall put-u16 106
    ccall put-u16 33
    ccall put-u16 10
    () ← exit tt
{-# COMPILE C hello #-}
```

Figure 11: Users embed Agda/C FFI descriptions in Agda source code

The `Exposed` type is an ordinary Agda record type (fig. 12), so users can exploit Agda's features when developing/reading/debugging `Exposed` declarations. In fact, the simple example we have just seen (fig. 11) contains some of Agda's advanced syntax, and calls some library functions, which can be expanded to give the lower-level Agda code in fig. 13.

```
-- Type of Agda→C function exports
record Exposed : Set where field
    own-signature : Sig               -- C function signature to export the function to
    imported-sigs : List Sig          -- function signatures of C dependencies
    function-body :
        -- The body is given instance-findable handles for...
        {{Ret-Handle (ret-ty own-signature)}} → -- returning a value
        map Call-Handle imported-sigs ⋯⦃⦄→     -- calling its (pre-declared) FFs
        map ⟦_⟧a (arg-tys own-signature) ⋯→  -- the body is passed its arguments (from C)
        CCS                                -- the body then produces a stream
```

Figure 12: The `Exposed` type is defined in an Agda library

```
hello : Exposed
hello .own-signature = sig void "main" []
hello .imported-sigs = (sig void "put_u16" (uint16 ∷ [])) ∷ []
hello .function-body {{ret-handle}} {{put-u16-handle}} =
    cmd 0 (cmd-ccall _ {{put-u16-handle}} (72 , tt)) >>= λ _ →
    cmd 0 (cmd-ccall _ {{put-u16-handle}} (101 , tt)) >>= λ _ →
    cmd 0 (cmd-ccall _ {{put-u16-handle}} (106 , tt)) >>= λ _ →
    cmd 0 (cmd-ccall _ {{put-u16-handle}} (33 , tt)) >>= λ _ →
    cmd 0 (cmd-ccall _ {{put-u16-handle}} (10 , tt)) >>= λ _ →
    cmd 1 (cmd-return {void} {{ret-handle}} tt) >>= λ ()
```

Figure 13: Fully expanded code from fig. 11

The lower-level code in fig. 13 reveals several light-weight tricks used to enforce FFI safety. Most strikingly, *capability* arguments such as `ret-handle` and `put-u16-handle` exploit Agda's type- and scope-checker to statically enforce FFI safety properties like "all foreign functions are known at compile-time" and "the values returned by Agda functions have the correct C type", but are kept out of users' view by marking them as *instance arguments*[13].

As well as enforcing FFI safety, the data structures visible in fig. 13 are carefully crafted to support a simple imperative implementation of the side-effecting Input/Output (IO) code after `.function-body`. Firstly, the use of `Nat`-tagged union types allows individual IO commands to be easily decoded using a `switch` statement in C. Secondly, the IO commands are structured into an *interaction tree*[18]: a (potentially infinite) stream of commands and *resumptions*[19], which can be

easily interpreted using a REPL-style `while` loop in C. Together, these two design choices allow the generation of an IO interpreter that fits in under 50 lines of C code (fig. 14), as we shall see now.

2.5 FFI Compiler

The user's `Exposed` declarations (fig. 11) are passed to the *FFI Compiler*, which generates C code (fig. 14) implementing them.

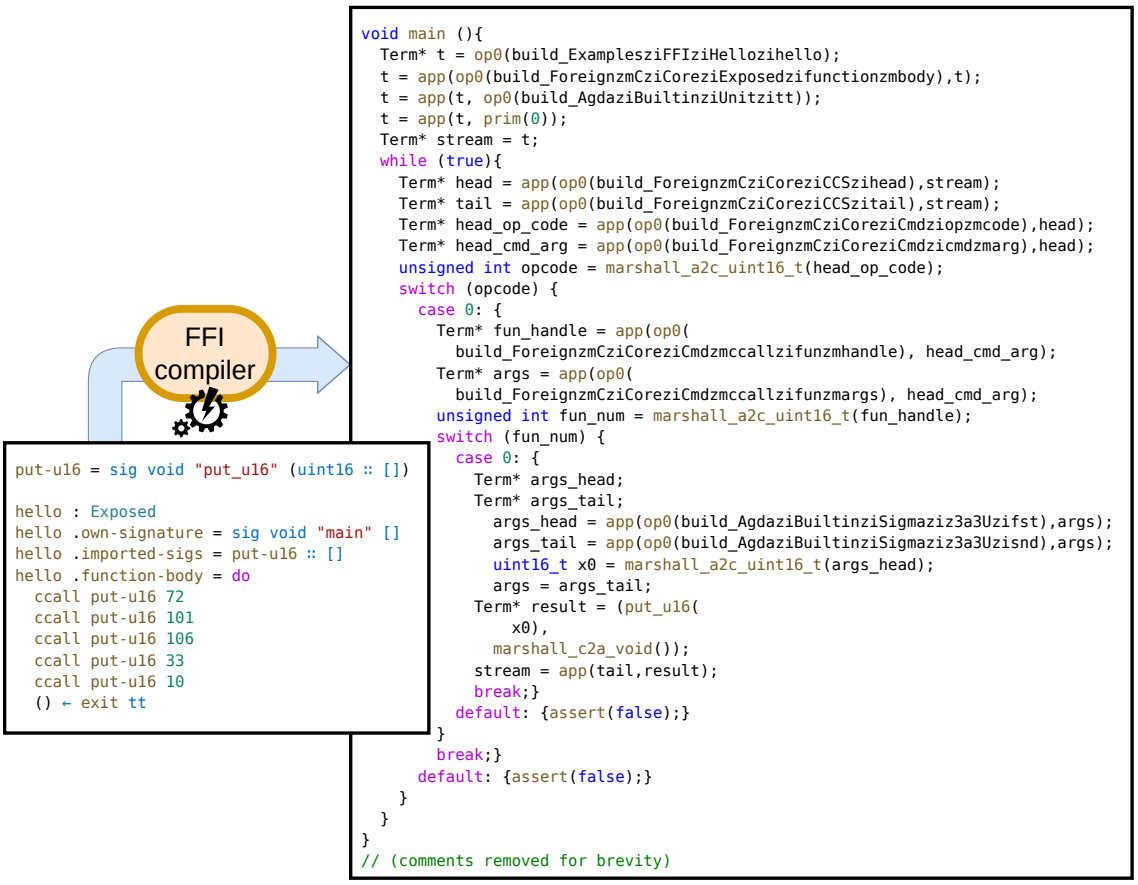


Figure 14: The FFI Compiler compiles an FFI description to C

In particular, the FFI Compiler generates an interpreter for the command stream in the `.function-body` field. This interpreter evaluates and decodes each command in turn, and contains a case covering each C function that may be called (here, just `put_u16`). The FFI Compiler determines which C functions may be called by traversing the list in the `.imported-sigs` field. The generated interpreter loop is wrapped in a C function (here, `void main()`) that sets up the command stream for the interpreter, including passing it any C arguments (here, none). The FFI compiler determines the function name, type, and argument information by traversing the `.imported-sigs` field.

These transformations are extensively documented by comments in the source code of the FFI Compiler[20], which is also highly readable

(and reliable) because it is implemented in the same language as the Exposed data type, i.e. as an Agda function (fig. 15).

```
-- Generate an Agda/C FFI implementation from an Exposed value.
compile-fun : String → String → Exposed → String
compile-fun agda-ident compiled-agda-ident exposed = "
// This C function was generated for the Exposed Agda identifier "++ agda-ident ++"
// In particular, the FFI compiler inspected the .own-signature and .imported-sigs fields.
// For reference, the values of these fields are:
//  "++ agda-ident ++" .own-signature = "++ show-sig own-signature ++"
//  "++ agda-ident ++" .imported-sigs = "++ show-list show-sig imported-sigs ++"
" ++ print-sig own-signature ++ "{
  Term* t = op0("++ compiled-agda-ident ++");
// extract the .function-body field
  t = app(op0("++ Phony-Zenc.Foreign-C.Core.function-body ++"),t);
// apply to {{tt : Ret-Handle ( "++ show-Ret-Ty (own-signature .ret-ty) ++" )}}
  t = app(t, op0("++ Phony-Zenc.Agda.Builtin.Unit.tt ++"));
" ++ unlines (for' imported-sigs λ n s →
"// apply to {{ "++ primShowNat n ++" : Call-Handle ( "++ show-sig s ++" ) }}
    t = app(t, prim("++ primShowNat n ++"));"
  )++
  unlines (for' (own-signature .arg-tys) λ an ty →
    "  t = app(t, " ++ (case ty of λ where
        uint16 → "marshall_c2a_uint16_t(arg"++ primShowNat an ++")"
        bool   → "marshall_c2a_bool(arg"++ primShowNat an ++")"
    ) ++");  // apply to C argument "++ primShowNat (suc an) ++" of "
++ primShowNat (№-args own-signature) ++ ", with type ⟦ "++ show-Arg-Ty ty ++" ⟧"
  )++ "

  // t now encodes ("++ agda-ident ++" .function-body), fully aplied to all its args

// much code omitted here

}
"   where
  open Exposed exposed
  open Sig
  №-args = λ s → length (s .arg-tys)
```

Figure 15: The FFI Compiler is implemented in Agda

2.6 GCC

Finally, the generated C code is compiled to ARM machine code by the *GNU Compiler Collection (GCC)* (fig. 16).

The compiler also relies on GCC to optimize the C code (e.g. by removing unused function definitions), and to link the generated C code with the BUBS runtime library and any other C libraries required by the application (e.g. the Raspberry Pi Pico Software Development Kit[21]). The ARM machine code emitted by GCC is then ready to upload to the Raspberry Pi Pico.

GCC is an existing and ubiquitous open-source project[22], so I will not discuss it further.

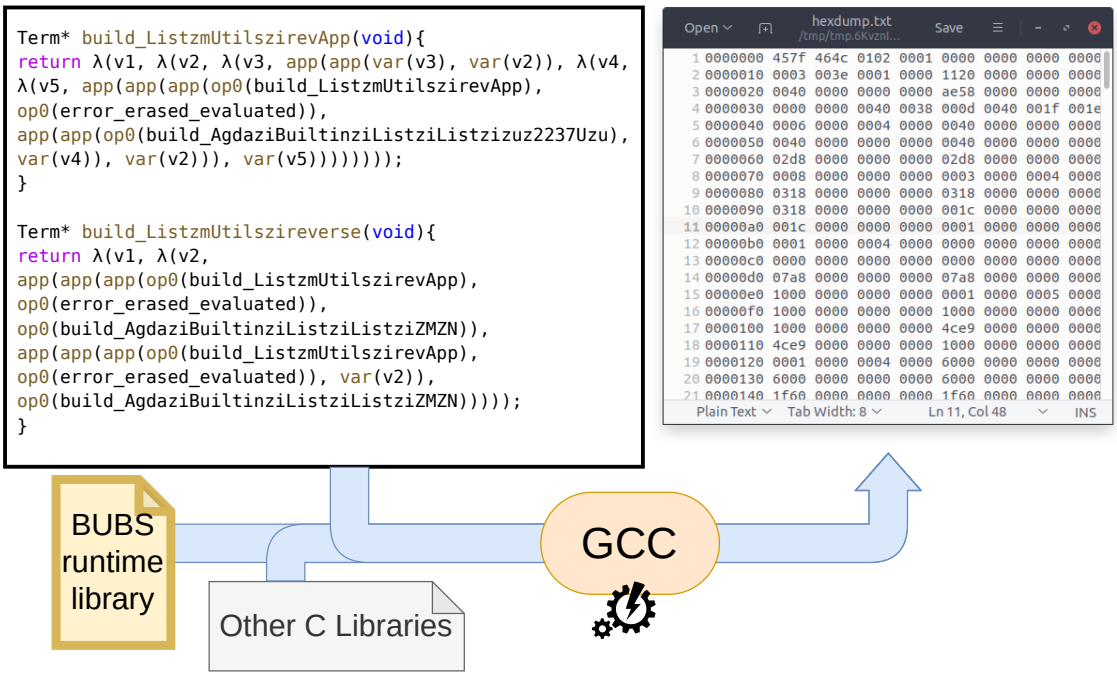


Figure 16: GCC compiles all the C to ARM Machine Code

3
Evaluation

The compiler works, but not very well.

For example, the Agda program in fig. 17 is successfully compiled, and the generated code runs as intended on the Raspberry Pi Pico – for 3 minutes, after which it crashes.

```
open import Foreign-C.Sugared

imp = "#include<pico/stdlib.h>"
{-# COMPILE RAW_C imp #-}

init = sig void "gpio_init" (uint16 ∷ [])
dir = sig void "gpio_set_dir" (uint16 ∷ bool ∷ [])
put = sig void "gpio_put" (uint16 ∷ bool ∷ [])
sleep = sig void "sleep_ms" (uint16 ∷ [])

main : Exposed
main .own-signature = sig void "main" []
main .imported-sigs = put ∷ init ∷ sleep ∷ dir ∷ []
main .function-body = do
    ccall init 25
    ccall dir 25 true
    loop
  where
    loop : CCS
    loop .head = ccall put 25 true
    loop .tail _ = do
        ccall sleep 300
        ccall put 25 false
        ccall sleep 500
        loop
{-# COMPILE C main #-}
```

Figure 17: Agda code to blink the Raspberry Pi Pico's LED

More precisely, the Raspberry Pi's LED blinks on and off (as intended) for 200 blinks, but then stops blinking (an error). I suspect that this error is caused by a slow memory leak, since about 200 blinks are always completed (regardless of what delay I set), and since the

program does not crash when running on my more memory-rich laptop.

As well as the (suspected) memory leak, `agda2c` has several known bugs:

- The FFI Compiler does not generate code implementing function return (so programs like `hello` from fig. 11 crash instead of exiting cleanly)
- The AST Compiler does not support several of Agda's primitive functions and literals. For example, `String`, `Float`, and `Word64` literals and operations currently compile to C code which crashes when evaluated (which is why fig. 11 uses character codes instead of Agda's built-in `String`s)
- The AST Compiler compiles `Natural` numbers to fixed-width integers, which can silently overflow at run-time.
- Incremental (re-)compilation of multi-file Agda programs is not enabled, greatly slowing down compilation times.
- Certain larger programs, e.g. a port of Claessen's PMC monad[23], crash when run, due to run-time assertion failures.

More speculative improvements are discussed in § 5.

Despite these bugs, `agda2c` is still capable of running simple Agda programs on the Raspberry Pi Pico, including programs with higher-order functions like `mapM'` in fig. 18. Full source code for these examples is available in `agda2c`'s GitHub repository[8].

```
-- Count in binary on LEDs attached to the RPP's pins
-- Simulator at https://wokwi.com/projects/343960210261410387

module Examples.FFI.Binary-Counter where
open import Examples.Lib.FIO

imp = "#include<pico/stdlib.h>"
{-# COMPILE RAW_C imp #-}

put = sig void "gpio_put_all" (uint16 ∷ [])
init = sig void "gpio_init" (uint16 ∷ [])
dir = sig void "gpio_set_dir" (uint16 ∷ bool ∷ [])
sleep = sig void "sleep_ms" (uint16 ∷ [])

main : Exposed
main .own-signature = sig void "main" []
main .imported-sigs = put ∷ init ∷ dir ∷ sleep ∷ []
main .function-body = run do
    mapM' setup-pin (0 ∷ 1 ∷ 2 ∷ 3 ∷ 4 ∷ 5 ∷ [])
    loop 0
  where
    setup-pin : Nat → FIO ⊤
    setup-pin n = do
        ccall init n
        ccall dir n true

    loop : Nat → FIO ⊥
    loop n _ .head = ccall' put n
    loop n k .tail c =
        ccall' sleep 200 >>=' λ _ →
        loop (n + 1) k
{-# COMPILE C main #-}
```

Figure 18: Agda code to drive a binary counter

4

Related Work

In this section, I discuss alternatives to the compiler components presented in § 2. In particular, I compare `agda2c`'s custom-built components to existing alternatives, and explain their weaknesses which motivated a new implementation.

4.1 Alternative Runtimes

Finding a runtime library for `agda2c` was difficult, due to an unusual mix of requirements: the runtime had to support *lazy evaluation*, provide *predictable performance*, and compile to *ARM machine code* capable of running in *limited memory* (200 KiB RAM, 2 MiB Flash).

In this section I describe the main contenders to BUBS (§ 2.3), and where they fell short.

4.1.1 Graph Reducers

The current implementation of the runtime library (§ 2.3) is based on Shivers & Wand's Bottom-Up β-Reduction algorithm[7] (BUBS), with several extensions. BUBS evaluates programs using *graph reduction*: programs are encoded as pointer graphs in memory, then repeatedly re-written using simple *reduction* transformations. By choosing different graph encodings (and reductions), different algorithms with different performance characteristics can be obtained. For example, BUBS, HVM[24], and Turner Combinators[25] all encode the program `(λf.λx.f(f x))` as a slightly different graph, as shown in fig. 19.

These algorithms share some common features: all three support lazy evaluation, all three can be implemented in a small and self-contained C file, and all three are compatible with precise garbage collection based on reference counting[26].

However, the algorithms have very different performance and transparency properties.

Programs compiled to use Turner Combinators can be very space efficient: the graphs can be laid out using only 2 pointers per node, and the code to evaluate them can also be made compact[1]. However, the

[1] Lennart Augustsson described an implementation that ran on a machine with under 100 KiB of memory, but I could not find it online.

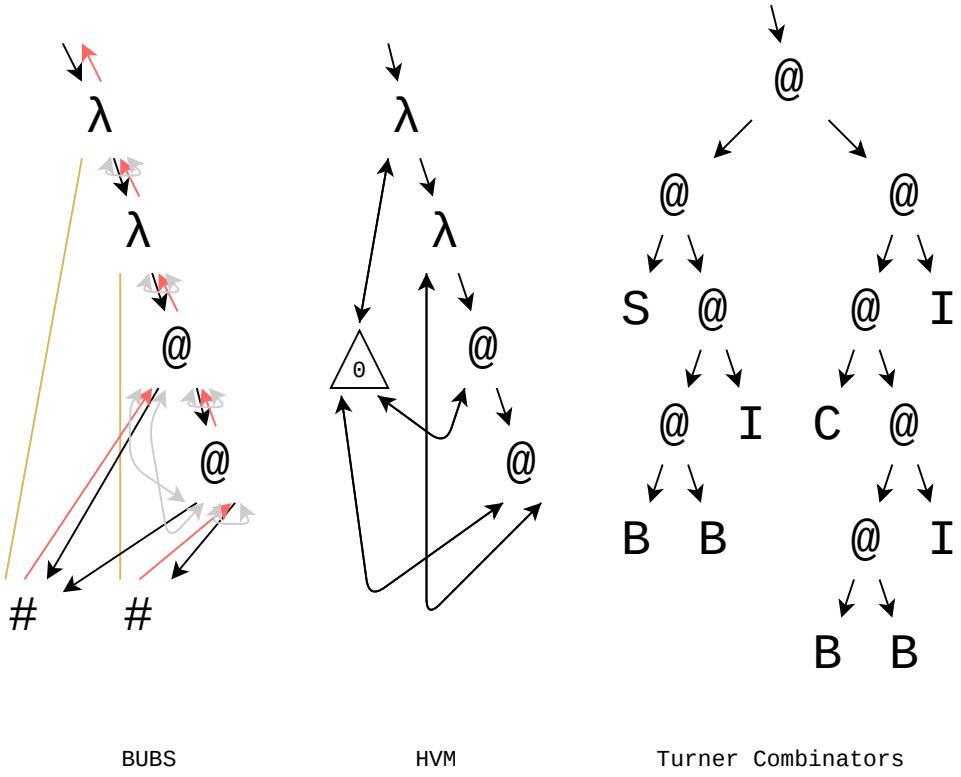


Figure 19: Different graph encodings of `(λf.λx.f(f x))`

Turner Combinator graphs can also be very space *in*-efficient: sometimes a program of length `n` is encoded as a graph with `O(n²)` nodes[2]. Turner Combinators also suffer from poor transparency: it is difficult to "read back" a λ-calculus program from a Turner Combinator graph, making it difficult to debug programs or to reason about their performance.

HVM graphs are more transparent than Turner Combinator graphs, but are disqualified for another reason: certain λ-calculus programs, such as `(λx.x x)(λf.λx.f(f x))`, are **incorrectly** evaluated by HVM. I do not know whether the 'HVM-safe' subset of λ-calculus programs covers all (compiled) well-typed Agda programs, so `agda2c` uses BUBS instead. In addition to HVM's unsafe evaluation, its reliance on *fan nodes* to track sharing of sub-terms makes HVM's performance hard to predict, both in theory[27] and in practice[28].

4.1.2 Other Interpreters

Not all runtime libraries are presented by giving a graph-encoding and a set of reductions. For example, the Three-Instruction Machine[29] and

[2]The `O(n²)` space is occupied by *director strings* of `B` and `C` combinators which guide top-down substitution during β-reduction. The BUBS paper[7] discusses director strings in more detail.

Sestoft's Lazy Virtual Machines[30], instead are described by a translation of to low-level byte code, and reductions for *machine states* including bytecode and one or more data structures, as in fig. 20.

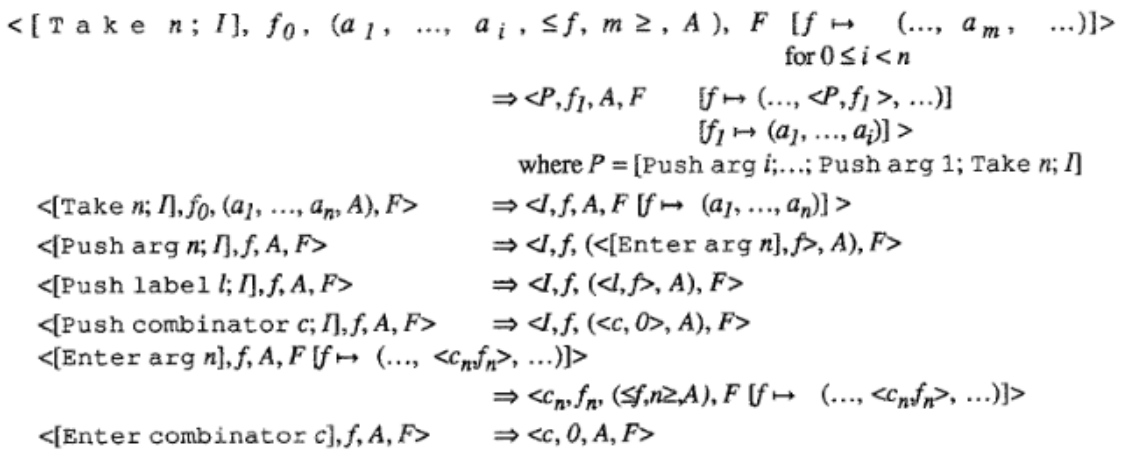


Figure 20: Three-Instruction Machine reductions, from [29]

Unfortunately, the intermediate states of these virtual machines do not obviously correspond to λ-calculus programs, making it difficult to debug programs or to reason about their performance. Additionally, the reductions in fig. 20 leave many implementation details implicit: for example which data is stored behind a pointer (as opposed to contiguously), and when garbage collection occurs. For these reasons, `agda2c` uses a graph-reduction-based runtime instead.

4.1.3 User-Facing Languages

A short-cut to finding a suitable runtime library is to take one from an existing implementation of another functional programming language. Promising candidates included implementations of other lazy languages (e.g. Haskell's *GHC* compiler[31], or its *Hugs* interpreter[32]) or of languages known to run on the Raspberry Pi Pico (e.g. JavaScript's *Kaluma* interpreter[33], or the *µLisp* interpreter[34]) or similar microcontrollers (e.g. Erlang's *GRiSP* microcontroller[35]).

Unfortunately, adapting these language implementations proved difficult: *GHC* requires an advanced build to emit ARM machine code (which the Raspberry Pi Pico executes), *Hugs* has a large implementation which has been unmaintained for 6 years, *µLisp* implements strict evaluation (and naive attempts at adding laziness via *delay arguments* led to an exponential slowdown), whilst *Kaluma* requires heavy modification to extend its C FFI, and the *GRiSP* microcontroller has substantially more memory than the Raspberry Pi Pico.

Additionally, the many of these implementations provide unwanted features such as interactive Read-Eval-Print Loops, concurrency, or invasive optimizations; which complicate their runtime libraries.

4.2 Alternative FFIs

I do not know of any other Agda-to-C compilers, so `agda2c`'s C FFI was designed from scratch. In this section I compare it to the C FFIs of several other programming languages.

4.2.1 Annotation-Based FFIs

Other compilers targeting C often include passes to compile FFI annotations to C glue code. For example, the Haskell[36], Zig[37], Rust[38], and Idris2[39] programming languages all use annotations to guide generation of C glue code.

Implementing these annotations can require a lot of effort: compilers must parse FFI annotations, check them, and compile them to glue code. Additionally, for the annotations to be easy to learn, they must produce informative error messages, and be thoroughly documented.

Some or all of these tasks can be automated by re-using existing language features. For example, Zig[37] presents its `@cImport` and `@cInclude` FFI annotations as built-in functions (providing cheap parsing).

`agda2c` goes further and specifies the syntax of FFI annotations as a type definition in an Agda library (§ 2.4; providing cheap documentation, parsing, type-checking and error messages), which are translated to glue code by an Agda function (§ 2.5; providing cheap compilation). Both Zig and `agda2c` also exploit compile-time evaluation (providing cheap flexibility).

4.2.2 Dynamic FFIs

Some scripting languages, for example *mJS*[40] and *bash*[41], allow C functions to be called given strings known only at run-time. However, the interpreters for these languages are made complex by code for resolving these strings at run-time: *mJS* uses an operating system's *dynamic linker*, and *bash* uses an operating system's *file system*. Neither of these operating system services are available on microcontrollers like the Raspberry Pi Pico.

4.2.3 Other Type-based FFIs

`agda2c` is not the first compiler to exploit its source language's user-defined data types to enforce a safe C FFI. For example, Blume describes a similar technique[42][3], implemented in the SML/NJ compiler for Standard ML. However, Blume's NLFFI models C's type system far more precisely than `agda2c`'s FFI. The NLFFI supports basic C

[3] A short presentation introducing Blume's NLFFI can be found at `https://www.jeffvaughan.net/docs/nlffi.pdf`.

types like `bool`, `int`, and `float`, but also more advanced C types like `structs`, data pointers, function pointers, and arrays, including arrays with known length information. In contrast, `agda2c`'s FFI only support `integers`, `boo`leans, and the `void` type. Nonetheless, these three C types are sufficient to interface with many C functions used to program the Raspberry Pi Pico [4], and `agda2c`'s FFI is simple enough to fit in a single Agda library file.

More recently, Idris1's C FFI[44] is partially documented using an Idris1 data type in a library, although instances of this data type are not constructed by users: instead, users write annotations to guide the Idris1 FFI Compiler. Another recent development is Hausmann's Contracts library[45], which (like `agda2c`) uses Agda data types to enforce a safe FFI, but targets Haskell rather than C.

I do not know of any other compilers whose FFI Compiler (§ 2.5) is implemented in a source-language library.

[4] The Raspberry Pi Pico Software Development Kit provides over 300 library functions. Out of all of their (over 600) arguments, there are: ~10 function pointers, ~20 enums, ~200 struct pointers, and all rest are basic types (e.g. `bool`,`uint32_t`,`void`). I got this data by `grep`ping for function signatures in https://github.com/raspberrypi/pico-sdk/tree/1.4.0/src/rp2_common[43].

5
Future Work

In this section, I describe future work motivated by the development of `agda2c`.

5.1 Benchmarking

When searching for a runtime library, I was surprised to find very little empirical research comparing the performance of different λ-calculus evaluators. Individual papers presenting a single evaluator (for example the BUBS paper[7]) often compare their evaluator to one or two other (usually simpler) evaluators, but I could not find a single survey paper. The most comprehensive dataset I found was the Benchmarks Game Website[46], however this compares entire language implementations, so the performance of the runtime libraries is obscured by language differences and compiler optimization passes. The Benchmarks Game also does not include many lazy languages.

Such a survey paper would provide valuable information for compiler developers, so I would like to see one written. I propose the Agda compiler as a test harness for comparing the evaluators: sometimes when type-checking a program, the Agda compiler must evaluate subprograms. Swapping out the current Agda compile-time evaluator for different evaluator algorithms would provide a labor-efficient way to benchmark (and test) these algorithms on a large, common, and already existing data set: the evaluation problems encountered when type-checking existing Agda programs.

5.2 BUBS Development

To my knowledge, `agda2c` is the first compiler to use a runtime library based on the BUBS algorithm, and has exposed several opportunities for improvement.

Firstly, the BUBS algorithm is easy to mis-implement: I spent many hours debugging memory safety failures, and my implementation likely has several remaining faults. These faults could be discovered by attempting to formally verify the implementation. Such a verification project is within reach: the BUBS paper[7] contains an (informal)

proof of correctness, which could guide a formal verification, and the amount of proof code could be kept manageable by rewriting BUBS in a carefully-designed embedded imperative language (§ 5.5).

Secondly, `agda2c`'s coverage of the Agda language and the flexibility of `agda2c`'s FFI are both constrained by the BUBS runtime's lack of support for *binary large objects (BLOB)* nodes to carry opaque data that may take an arbitrary amount of space (e.g. a C `struct` or `char[]`), or require advanced de-allocation procedures (e.g. operating system file handles, variable-precision integers[47], or Haskell-style `Text` values[48]).

Thirdly, the BUBS runtime's data structures could be tuned for (constant-factor) performance gains. Low-hanging fruit includes the memory wasted by storing co-parent pointers (light gray in fig. 9) even for non-shared nodes, and reliance on the C stack (rather than extra 'navigation bits' in graph nodes) to control traversal of the graph during evaluation.

Finally, specializing the BUBS runtime library towards Agda's syntax (e.g. by supporting constructors, `case` expressions, and `let` bindings) would increase `agda2c`'s transparency and simplify compiler development, for example by making Scott-encoding redundant.

5.3 FFI Improvements

`agda2c`'s FFI currently only supports the `void`, `uint16_t` and `bool` C types. Adding support for other C types like `float`, `char`, pointers, and `struct` types is also easy, if the expressions of these types are made *opaque* to Agda programs. A more advanced design, allowing *data-level interoperability* could be based on Blume's "No Longer Foreign" SML/C interface[42]

5.4 First-Class Optimization

We saw in § 2.4 how an Agda EDSL configures `agda2c`'s FFI generator, and that this brings usability and compiler-development advantages. Perhaps similar benefits could be reaped for other compiler components (e.g. the Agda frontend's optimizer) if they were also configured by an Agda EDSL. Previous work by Hagedorn et al.[49] shows how to configure an optimizer for a (domain-specific) functional-programming language with another stand-alone domain-specific language. Perhaps their design can be extended to Agda (a general-purpose language), and made easier to learn by embedding the configuration language in Agda.

5.5 C-in-Agda

One way of finding bugs in the BUBS runtime library (§ 2.3) would be to re-write it in a C-like Agda EDSL that compiles trivially to C, but has a stronger type checker. I have written a prototype[50] of such an Agda EDSL, which allows imperative programs to be written in Agda (fig. 21), with C-like syntax and type-checking.

```
-- Selection sort
sel-sort : Exp' kt1 nat → Exp' kt2 nat → Stmt
sel-sort base n = do
    i ← mkvar base
    n ← mkvar n
    while (i < n) do
        j ← mkvar i + 1
        while (j < n) do
            if heap[ j ] < heap[ i ]
                then swap i j
                else skip
            j ++
        i ++
```

Figure 21: Selection sort, in C, in Agda

Although this EDSL provides very C-like syntax, it would be difficult to implement and verify BUBS in this EDSL since it is missing:

- Compound C data types (e.g. `structs` and pointer types)
- C function declarations and calls (e.g. for functions like `swap`, which are currently implemented as *Agda* functions, so are inlined during compilation to C)
- Correctness-oriented features (e.g. statically checked assertions) that would allow for more static verification than writing programs directly in C

Extending the current implementation to support these features would require solving several problems:

- How to provide C-like syntax, without cluttering up the types of expressions (e.g. the type of `n` in fig. 21, which is currently `Exp' kt2 nat` but would be more readable as `Exp nat`)
- How to extend the reference interpreter to support these features, without destroying its type-safety or conciseness. In particular, I am not sure how to extend the support for *name-binding syntax*, which is currently based on András Kovács's pure `runST` function[51] and only supports *local variables* (e.g. `j` in fig. 21), to support new forms of name-binding syntax such as user-declared `struct` types, of user-declared functions.

6

Conclusion

This thesis has presented the design of the `agda2c` compiler (§ 2), including its heavy reliance on the Agda Frontend (§ 2.1), a surprisingly small post-processing stage (§ 2.2), and an unusual choice of runtime library (§ 2.3). We also saw how `agda2c` hides the complexity of writing a C FFI behind a high-level language of FFI descriptions (§ 2.4), and how `agda2c` compiles these FFI descriptions (§ 2.5) to executable code.

Throughout the development of `agda2c`, several development techniques and decisions stood out:

- Storing partially-compiled programs in typed EDSLs helped keep the compiler small (due to lack of parser) and robust (due to type checking of code that traverses or constructs EDSL values). `agda2c` compiles programs through (at least) three typed EDSLs:
  - The *Treeless IR* language of simplified Agda programs, embedded in Haskell.
  - The *Exposed* data type of FFI descriptions, embedded in Agda.
  - The *`bubs.h`* interface to the BUBS runtime library, embedded in C.
- Choosing a *transparent* runtime algorithm (BUBS) made `agda2c` easier to debug.
- Re-using and combining existing tools (e.g. PolyML, `valgrind`, `gdb`, `graphviz`) also aided debugging.

7

References

[1] J. Hughes, “Why functional programming matters,” *The Computer Journal*, vol. 32, no. 2, pp. 98–107, Apr. 1989, doi: 10.1093/comjnl/32.2.98.

[2] “Agda 2.” Agda Github Community, Jun. 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/agda/agda

[3] “TIOBE Index,” *TIOBE*. Accessed: Jun. 21, 2023. [Online]. Available: https://www.tiobe.com/tiobe-index/

[4] “Raspberry Pi Documentation - Raspberry Pi Pico and Pico W,” *www.raspberrypi.com*. Accessed: Jun. 21, 2023. [Online]. Available: https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html

[5] “CPU DB - Looking At 40 Years of Processor Improvements A complete database of processors for researchers and hobbyists alike.” *cpudb.stanford.edu*. Accessed: Jun. 21, 2023. [Online]. Available: http://cpudb.stanford.edu/

[6] A. Allan, “Multilingual blink for Raspberry Pi Pico,” *Raspberry Pi*. Jan. 2022. Accessed: Jun. 21, 2023. [Online]. Available: https://www.raspberrypi.com/news/multilingual-blink-for-raspberry-pi-pico/

[7] O. Shivers and M. Wand, “Bottom-up β-reduction: Uplinks and λ-DAGs,” *Fundamenta Informaticae*, vol. 103, no. 1–4, pp. 247–287, Jan. 2010, Accessed: Jun. 21, 2023. [Online]. Available: https://dl.acm.org/doi/10.5555/1922521.1922535

[8] L. Chonavel, “Lawcho/agda2c.” May 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/lawcho/agda2c

[9] “Welcome to Agda’s documentation! — Agda 2.6.3 documentation,” *agda.readthedocs.io*. Accessed: Jun. 21, 2023. [Online]. Available: https://agda.readthedocs.io/en/v2.6.3/

[10] U. Norell, “Towards a practical programming language based on dependent type theory,” 2007. Accessed: Jun. 21, 2023. [Online]. Available: https://www.semanticscholar.org/paper/Towards-a-practical-programming-language-based-on-Norell/345b2a3a2e99221f180b752b06b7f229993f5c9a

[11] A. Abel and B. Pientka, “Higher-Order Dynamic Pattern Unification for Dependent Types and Records,” in *Typed Lambda Calculi and Applications*, L. Ong, Ed., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 10–26. doi: 10.1007/978-3-642-21691-6_5.

[12] N. A. Danielsson and U. Norell, “Parsing mixfix operators,” in *Proceedings of the 20th international conference on Implementation and application of functional languages*, in IFL’08. Berlin, Heidelberg: Springer-Verlag, Sep. 2008, pp. 80–99. Accessed: Jun. 21, 2023. [Online]. Available: https://dl.acm.org/doi/10.5555/2044476.2044481

[13] D. Devriese and F. Piessens, “On the bright side of type classes: Instance arguments in Agda,” *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 143–155, Sep. 2011, doi: 10.1145/2034574.2034796.

[14] L. Chonavel, “Lawcho/agda2c.” May 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/lawcho/agda2c/blob/35deceb95b0b725b0a9a0e72c9ce521303f5710f/agda2c.hs

[15] “Zenc,” *Hackage*. Accessed: Jun. 21, 2023. [Online]. Available: https://hackage.haskell.org/package/zenc

[16] P. Koopman, R. Plasmeijer, and J. M. Jansen, “Church Encoding of Data Types Considered Harmful for Implementations: Functional Pearl,” in *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*, in IFL ’14. New York, NY, USA: Association for Computing Machinery, Oct. 2014, pp. 1–12. doi: 10.1145/2746325.2746330.

[17] L. Chonavel, “Bottom-up β-reduction.” Apr. 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/lawcho/bubs

[18] L. Xia *et al.*, “Interaction trees: Representing recursive and impure programs in Coq,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, Jan. 2020, doi: 10.1145/3371119.

[19] S. Goncharov, L. Schröder, C. Rauch, and J. Jakob, “Unguarded Recursion on Coinductive Resumptions,” *Logical Methods in Computer Science*, vol. Volume 14, Issue 3, Aug. 2018, doi: 10.23638/LMCS-14(3:10)2018.

[20] L. Chonavel, “Lawcho/agda2c.” May 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/lawcho/agda2c/blob/35deceb95b0b725b0a9a0e72c9ce521303f5710f/agda-src/Foreign-C/FFI-Compiler.agda

[21] "Raspberry Pi Pico SDK." Raspberry Pi, Jun. 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/raspberrypi/pico-sdk

[22] "GCC, the GNU Compiler Collection - GNU Project," *gcc.gnu.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://gcc.gnu.org/

[23] K. Claessen, "A poor man's concurrency monad," *Journal of Functional Programming*, vol. 9, no. 3, pp. 313–323, May 1999, doi: 10.1017/S0956796899003342.

[24] "Higher-order Virtual Machine (HVM)." HigherOrderCO, Jun. 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/HigherOrderCO/HVM

[25] D. A. Turner, "A new implementation technique for applicative languages," *Software: Practice and Experience*, vol. 9, no. 1, pp. 31–49, Jan. 1979, doi: 10.1002/spe.4380090105.

[26] D. F. Bacon, P. Cheng, and V. T. Rajan, "A unified theory of garbage collection," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, in OOPSLA '04. New York, NY, USA: Association for Computing Machinery, Oct. 2004, pp. 50–68. doi: 10.1145/1028976.1028982.

[27] J. L. Lawall and H. G. Mairson, "Optimality and inefficiency: What isn't a cost model of the lambda calculus?" *ACM SIGPLAN Notices*, vol. 31, no. 6, pp. 92–101, Jun. 1996, doi: 10.1145/232629.232639.

[28] "Quadaric time & space on heavily shared lazy lists · Issue #60 · HigherOrderCO/HVM," *GitHub*. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/HigherOrderCO/HVM/issues/60

[29] J. Fairbairn and S. Wray, "TIM: A simple, lazy abstract machine to execute supercombinators," in *Proc. Of a conference on Functional programming languages and computer architecture*, Berlin, Heidelberg: Springer-Verlag, Oct. 1987, pp. 34–45. Accessed: Jun. 21, 2023. [Online]. Available: https://dl.acm.org/doi/10.5555/36583.36586

[30] P. Sestoft, "Deriving a lazy abstract machine," *Journal of Functional Programming*, vol. 7, no. 3, pp. 231–264, May 1997, doi: 10.1017/S0956796897002712.

[31] "Home — The Glasgow Haskell Compiler," *www.haskell.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://www.haskell.org/ghc/

[32] "Hugs 98," *www.haskell.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://www.haskell.org/hugs/

[33] “Kaluma,” *kalumajs.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://kalumajs.org/

[34] “uLisp,” *www.ulisp.com*. Accessed: Jun. 21, 2023. [Online]. Available: http://www.ulisp.com/

[35] “GRiSP,” *www.grisp.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://www.grisp.org/

[36] “Foreign Function Interface - HaskellWiki,” *wiki.haskell.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://wiki.haskell.org/Foreign_Function_Interface

[37] “Documentation - The Zig Programming Language,” *ziglang.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://ziglang.org/documentation/master/#Import-from-C-Header-File

[38] “FFI - The Rustonomicon,” *doc.rust-lang.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://doc.rust-lang.org/nomicon/ffi.html

[39] “FFI Overview — Idris2 0.0 documentation,” *idris2.readthedocs.io*. Accessed: Jun. 21, 2023. [Online]. Available: https://idris2.readthedocs.io/en/latest/ffi/ffi.html

[40] “mJS: Restricted JavaScript engine.” Cesanta Software, Jun. 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/cesanta/mjs

[41] “Bash - GNU Project - Free Software Foundation,” *www.gnu.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://www.gnu.org/software/bash/bash.html

[42] M. Blume, “No-Longer-Foreign: Teaching an ML compiler to speak C “natively”,” *Electronic Notes in Theoretical Computer Science*, vol. 59, no. 1, pp. 36–52, Nov. 2001, doi: 10.1016/S1571-0661(05)80452-9.

[43] “Pico-sdk/src/rp2_common at 1.4.0 · raspberrypi/pico-sdk,” *GitHub*. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/raspberrypi/pico-sdk

[44] “New Foreign Function Interface — Idris 1.3.3 documentation,” *docs.idris-lang.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://docs.idris-lang.org/en/latest/reference/ffi.html

[45] P. Hausmann, “The Agda UHC Backend,” Master’s thesis, 2015. Accessed: Jun. 21, 2023. [Online]. Available: https://studenttheses.uu.nl/handle/20.500.12932/28189

[46] “Which programming language is fastest? (Benchmarks Game),” *benchmarksgame-team.pages.debian.net*. Accessed: Jun. 21, 2023. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

[47] “The GNU MP Bignum Library,” *gmplib.org*. Accessed: Jun. 21, 2023. [Online]. Available: https://gmplib.org/

[48] “Text,” *Hackage*. Accessed: Jun. 21, 2023. [Online]. Available: https://hackage.haskell.org/package/text-2.0.2

[49] B. Hagedorn, J. Lenfers, T. Kœhler, X. Qin, S. Gorlatch, and M. Steuwer, “Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–29, Aug. 2020, doi: 10.1145/3408974.

[50] L. Chonavel, “A subset of C embedded in Agda.” Mar. 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/lawcho/c-in-agda

[51] A. Kovács, “AndrasKovacs/misc-stuff.” Apr. 2022. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/AndrasKovacs/misc-stuff/blob/1c6426871dea46edcadbf3da94ce0228442de09f/agda/ST/ST2.agda