Utrecht University

A WasmGC backend and runtime for the MicroHs compiler

Master's thesis part 1: project proposal

Martijn Voorwinden – 7854064
Utrecht University – Department of Information and Computing Sciences
1st supervisor: Marco Vassena
2nd supervisor: Wouter Swierstra

Abstract

Contents

Introduction

Running Haskell code in the context of web browsers, cloud platforms, or similar portable environments is complicated. Approaches such as transpilation to JavaScript [1] suffer from performance issues and lack of semantic similarity (e.g. functional vs imperative, lazy vs eager eveluation, etc.), while solutions such as containerization can be too heavy-weight for resource-constrained environments and cannot be used in web browsers.

A promising alternative is compiling Haskell to WebAssembly (Wasm) [2, 3, 4], a low-level bytecode format designed for efficient execution in portable environments such as web browsers.

Traditionally Wasm did not natively support garbage collection (GC), which forced compilers targeting Wasm to ship their own memory management systems. This results in large binary sizes as well as increased complexity for the compiler/runtime developers. However, with the recent release of Wasm 3.0 [5], Wasm now natively supports GC. It does this by leveraging the host environment's GC capabilities. As current Haskell compilers targeting Wasm do not make use of Wasm's GC support, we will explore the effects of making use of it by extending an existing Haskell-to-Wasm compiler MicroHs [6] to target Wasm 3.0.

MicroHs is a small Haskell compiler (relative to mainstream compilers such as GHC) and runtime system that supports most of Haskell2010 and a subset of GHC extensions. It compiles Haskell code to SKI combinator calculus, which is then combined with a run-time system written in C and compiled to the target platform. Furthermore, MicroHs can target Wasm through its C backend by compiling the resulting C code with Emscripten [7]. Its small codebase and modular architecture make it a suitable candidate to experiment on without spending too much time on implementation details.

Previously, Anand [8] made a prototype extension to MicroHs to target the WasmGC proposal (which is the basis for Wasm 3.0's GC support) and implemented a proof-of-concept runtime system written in a combination of WasmGC and a custom domain specific language (DSL). Initial experiments showed promising results in terms of binary size, however runtime performance was not tested.

This thesis aims to extend Anand's work by:

1. Fully implementing the runtime system in a custom DSL that compiles to Wasm 3.0.
2. Extending support for more Haskell features by implementing runtime and backend support for new combinators and primitive operations, or by implementing a new simplified prelude that can compile to a smaller set of combinators.
3. Evaluating the size and runtime performance of the generated Wasm binaries against other Haskell-to-Wasm compilers and MicroHs's existing C backend.

The rest of this thesis proposal is structured as follows:

- Chapter 2 provides the reader with the necessary background on WebAssembly and the architecture of MicroHs.
- Chapter 3 discusses related work on the topic.
- Chapter 4 details the progress made on the project during Part 1 of the thesis.
- Chapter 5 outlines the plan for Part 2 of the thesis.

Background

2.1 MicroHs

1. Compilation pipeline
2. SKI combinator calculus

2.2 WebAssembly

1. Introduction to base Wasm
2. Wasm GC extension

Related Work

3.1 Haskell for the Web

Historically, running Haskell code in web browsers has been achieved using GHCJS [1], a compiler that transpiles Haskell code to JavaScript. While GHCJS allows Haskell code to run in browsers, it suffers from multiple problems. Due to it having to include large parts of the GHC runtime system resulting in large bundle sizes [9]. Furthermore, the development has lagged behind GHC itself, causing GHCJS to not be able to take advantage of the latest compiler optimizations and language extensions [1].

Other Haskell-to-JavaScript compilers exist, such as UHC [10] Fay [11], and Haste [12]. However, each of these compilers have not seen active development in recent years, and thus are not suitable for modern Haskell web development.

WebAssembly has emerged as a promising alternative for running Haskell code on the web. Historically, this has been achieved through the Asterius project [4], which is a Haskell-to-Wasm compiler based on GHC. The project has however been deprecated in favour of GHC's own Wasm backend [3], which is actively being developed as part of GHC itself. Both Asterius and GHC's Wasm backend however target traditional Wasm without GC support, which results in them having to include a memory management system in produced Wasm binaries, leading to larger binary sizes and increased complexity.

Other functional programming languages that share a large amount of similarities with Haskell have also been used for web development. Elm [13] is a purely functional programming language that compiles to JavaScript and is designed specifically for building web applications. It features a strong type system and emphasizes simplicity and ease of use. Similarly, PureScript [14] is another strongly-typed functional programming language that compiles to JavaScript with a syntax and semantics inspired by Haskell. While they may look like Haskell, there are many semantic differences and are thus not directly applicable to this thesis's goals.

3.2 WebAssembly with Garbage Collection

WebAssembly's GC support is a relatively recent addition to the WebAssembly specification, aimed at improving the performance and efficiency of languages that rely on GC, such as Haskell, Java, and C#. Since its introduction, it has seen wide adoption in various projects and languages. It is supported by all major web browsers, including Chromium-based browsers, Firefox, and Safari, as well as non-web environments like Node.js, Deno and Wasmtime [5]. Furthermore there is support for GC specific optimizations in popular Wasm optimzers such as Binaryen [5, 15]. And it is targeted by multiple compilers, such as Dart, Kotlin/Wasm, WasOCaml, Hoot, and J2CL/Wasm [16, 17, 18, 19, 20, 21].

As demonstrated by the V8 team, using Wasm's with GC is highly optimizable, both by compile time optimizations as well as runtime optimizations [22]. They demonstrated that using a compile time optimizer such as Binaryen with Wasm with GC can produce binaries that have up to 2 times higher throughput (higher is better). Likewise, they demonstrated that enabling runtime optimizations in V8 can lead to lower latency (lower is better) by porting the Google Sheets Calc Engine to Wasm with GC.

Performance testing of binaries produced by the Kotlin compiler shows that the Wasm backend produces binaries that are significantly faster than the JavaScript backend, and is approaching the performance of the JVM backend [17]. When looking at functional programming languages, the WasOCaml compiler targets Wasm with GC, per their benchmarks, binaries produced by WasOCaml are slower by an almost constant factor of circa 2 times when compared to native OCaml binaries [18]. They show that a major slowdown is caused by the exception handling mechanism in Wasm when using V8, i.e. a raise instruction is about 100 times slower in Wasm when compared to native OCaml. Due to their dependence on other Wasm proposals such as tail calls, WasOCaml could only be benchmarked on V8, which supports all required proposals. When looking at Wasm backends for Haskell, the GHC Wasm backend has been benchmarked against GHC's native backend, showing that the Wasm backend is between 4 to 1 times slower than the native backend depending on the benchmark, most likely due to not using Wasm with GC and other modern Wasm features [21].

3.3 Combinator Based Compilers and Runtime Systems

The combinator based compiler backend and runtime MicroHs uses is based on the work of Turner [23], who introduced the idea of compiling programs to SKI combinator calculus and executing them on an SK-reduction machine. Turner made use of this system in multiple functional programming languages, such as the SASL and KRC languages which were intended for research and teaching purposes [24, 25]. Later, the system was also used in the production Miranda language [26], which has had significant influence on the design of Haskell.

While modern runtimes for lazy functional programming languages such as GHC are implemented using abstract machines such as the Spineless Tagless G-machine (STG) [27], experimenting with combinator based runtimes is still relevant as our focus is not on optimizing the runtime system itself, but rather on exploring the effects of using Wasm GC support as opposed to shipping our own GC when compiling Haskell to Wasm.

Progress

4.1 DSL for Generating Wasm 3.0 Code

To help in generating Wasm 3.0 code, we have designed and implemented a small DSL embedded in Haskell. The DSL is a small deeply embedded imperative language [28] that abstracts away the explicit stack manipulation that is necessary when writing Wasm code directly. The DSL is close to Wasm 3.0 in its structure, in the sense that it has a notion of a module `RtSpec` that has functions `RtFunc`, imports, types `RtTypeDef`, and functions that have explicitly declared local variables. The body of a function consists of a single expression `RtExpr` that can a be a block of expressions, a control flow construct, or a simple expression. A selected set of the constructs provided by the DSL can be seen in Figure 4.1.

```haskell
data RtSpec = RtSpec
    { specTypes   :: [RtTypeDef]
                  --  Name     Args      Results
    , specImports :: [(String, [RtTyp], [RtTyp])]
    , specFuncs   :: [RtFunc]
    , specMain    :: RtMainFunc
    }
data RtFunc = RtFunc
    { funcName   :: String
    , funcArgs   :: [(String, RtTyp)]
    , funcRetTyp :: [RtTyp]
    , funcLocals :: [(String, RtTyp)]
    , funcBody   :: RtExpr
    }
data RtTypDef = RtTypDef
    { typeName :: String
    , typeDef  :: RtTyp
    }
data RtExpr
    = Block [RtExpr]
    | If RtExpr RtExpr RtExpr
    | While RtExpr RtExpr
    | Case RtExpr [RtExpr]
    | Call String [RtExpr]
    | Let String RtExpr
    | Return RtExpr
              ⋮
    | IntLit Int
```

Listing 4.1: Selected constructs provided by the DSL.

Using the constructs provided by the DSL, we can implement functions in a way that is close to how they would be implemented in Wasm 3.0, but in and easier-to-read and write manner. An example of this can be seen in Listing 4.2, which shows the implementation of a function that calculates the length of the left spine of a graph.

```haskell
leftSpineLength :: RtFunc
leftSpineLength = RtFunc
    { funcName = _leftSpineLength
    , funcArgs = [ (_node, TNRef _Node) ]
    , funcRetTyp = [TInt]
    , funcLocals = [ (_length  , TInt       )
                   , (_current , TNRef _Node)
                   ]
    , funcBody = Block
        [ Let [_length ] (IntLit 0)
        , Let [_current] (Var _node)
        , traverseLeft _current $
            Let [_length] (increment (Var _length))
        , Return (Var _length)
        ]
    }
```

Listing 4.2: Example of a function implementation in the DSL

On top of the DSL, we have made shorthand functions that generate a block of DSL expressions for patterns used in the original DSL used by Anand [8] for implementing reduction rules for the combinators and primitives used in MicroHs. This allows us to implement the runtime system entirely in the new DSL while reusing the existing combinator and primitive implementations. As an example, the implementation of the reduction of the `S` combinator in both the original DSL and the new DSL can be seen in Listing 4.3.

```haskell
-- S x y z -> x z (y z)
-- Original DSL
redRuleS :: [MixedInstr]
redRuleS = redRule n ln rn
    where
        n  = 3
        ln = GI (MkNode (CVar (MV X))) (VVar (MV Z))
        rn = GI (MkNode (CVar (MV Y))) (VVar (MV Z))
-- New DSL
redRuleS :: RtExpr
redRuleS = redRule n ln rn
    where
        n  = 3
        ln = mkNode x z
        rn = mkNode y z
```

Listing 4.3: Implementation of the reduction of the `S` combinator in both the original DSL (left) and the new Wasm 3.0 DSL (right).

4.2 Runtime System Implementation

After reimplementing the runtime system in the new DSL, we encountered several issues that needed to be resolved before being able to properly reduce a program. One of the main issues in the old runtime system was how the left ancestor stack (LAS) is handled.

The old implementation makes use of two functions to apply reduction rules to a node `reduce` and `step`. The `reduce` function is responsible for reducing a node completely, while the `step` function is responsible for applying a single reduction step to a node. The `step` function makes use of the LAS to keep track of the ancestors of the current node being reduced, which is necessary when performing reductions to efficiently access and update the ancestors of the current node. The `reduce` function constructs the LAS for the node being reduced and then calls `step` repeatedly until the node is fully reduced.

There are two main issues with how this process handles the LAS. The first issue is that the LAS is constructed by traversing the left spine of the node being reduced twice, once to calculate its length and once to populate the LAS array. This is inefficient as it requires traversing the left spine of the node being reduced twice, which can be costly for deep left spines. The second issue is that it does not correctly keep track of the LAS, an example of this is in reducing a graph that represents the expression `I x` where `x` has a deep left spine. When reducing this graph, the LAS is constructed for the root node `I`, which is empty as `I` has no ancestors. When the `I` combinator is reduced, it replaces itself with its argument `x`, but the LAS is not updated to reflect this change. As a result, when the reduction continues on `x`, the LAS is still empty, which leads to incorrect behavior when trying to access or update the ancestors of `x`.

To resolve these issues, the LAS should be maintained throughout the entire reduction process, rather than being constructed only at the beginning of the reduction. An example of the algorithm can be seen in Algorithm 1. This algorithm maintains the LAS throughout the entire reduction process, updating it as necessary when nodes are reduced. This ensures that the LAS is always accurate and up-to-date, allowing for correct access and updates to the ancestors of the current node being reduced.

Algorithm 1: SK Reduction with maintained LAS

```
Input  : Node n to reduce
Output: Reduced node
$las \leftarrow \emptyset$;
$las_{pointer} \leftarrow -1$;
$cur \leftarrow n$;
while $cur$ is not fully reduced do
    if $cur$ is a combinator then
        $(cur, las_{pointer}) \leftarrow \text{step}(cur, las, las_{pointer})$;
    else
        $cur \leftarrow \text{leftChild}(cur)$;
    end
    $las[las_{pointer}] \leftarrow cur$;
    $las_{pointer} \leftarrow las_{pointer} + 1$;
end
```

This however comes with a new problem, when we have the graph of a program, we cannot statically know the maximum depth of the LAS that will be needed during the reduction of the graph, as the graph allows for recursion that can lead to arbitrarily deep left spines. To resolve this there are three possible approaches:

1. Allocate a stack with a chosen maximum depth for the LAS. If the maximum depth is exceeded during reduction, an error is raised. This approach is used by

MicroHs's C runtime system and simple to implement, but can result in errors for programs that require a deeper LAS than the chosen maximum depth, and can waste memory if the maximum depth is set too high.

2. Dynamically resize the LAS stack when the maximum depth is exceeded. This approach allows for arbitrary depth of the LAS, but a growing strategy needs to be chosen carefully to avoid excessive memory allocations and copying.
3. Reuse the Wasm stack for the LAS. This approach leverages the existing Wasm stack to store the LAS, avoiding the need for a separate stack allocation. However, this requires careful management of the Wasm stack to ensure that it is used correctly and does not interfere with other parts of the program.

Another change in the runtime system is the type used for representing nodes. In the old runtime system, nodes were represented using a struct with two fields `left` and `right`, both of type `anyref`. Where the `anyref` could either be a pointer to another node or a pointer to an integer value. This representation is not ideal as it requires frequent runtime type checks and casts when accessing the fields of a node. In the new runtime system we have opted to represent nodes using a struct with three fields `left`, `right`, and `val`. The `left` and `right` fields are references to other nodes, while the `val` field is an `anyref`, an `anyref` is needed in order to represent non-integer primitive values such as a floating-point number. Furthermore, we enforce the following constraints on the nodes: Let n be a node, then $n.left = \text{null} \Leftrightarrow n.right = \text{null}$, $n.left = \text{null} \Rightarrow n.val \neq \text{null}$, and $n.left \neq \text{null} \Rightarrow n.val = \text{null}$. This means that a node is either an internal node with both left and right children and no value, or a leaf node with no children and a value. This representation allows us to avoid runtime type checks and casts when accessing the fields of a node, as we can determine the type of a node through `null` checks on its fields.

4.3 Sanity Check

To check the feasibility of the Wasm 3.0 backend, we have ran a small Haskell program on both the Wasm 3.0 backend as well as the existing Wasm backend (through emscripten) and compared the results. To force the execution of the program, we have temporarily implemented a simple primitive `pint32` that takes a 32-bit integer value and prints it to the console after which it returns the value `0`. The Haskell program used for the sanity check can be seen in Listing 4.4. Notably the program does not make use of the Prelude, instead opting to define its own data types and functions, and making use of the primitive `Int` type defined in MicroHs's `Primitives` module. The results of the sanity check can be seen in Table 4.1.

```haskell
module Simple where
import qualified Prelude ()
import Primitives

data Peano = Zero | Succ Peano
    deriving ()
data Bool = False | True
    deriving ()

one :: Peano
one = Succ Zero
two :: Peano
two = Succ one
three :: Peano
three = Succ two
add :: Peano -> Peano -> Peano
add Zero     y = y
add (Succ x) y = Succ (add x y)
mul :: Peano -> Peano -> Peano
mul Zero     _ = Zero
mul (Succ x) y = add y (mul x y)
fac :: Peano -> Peano
fac Zero     = Succ Zero
fac (Succ x) = mul (Succ x) (fac x)
eqnat :: Peano -> Peano -> Bool
eqnat Zero     Zero     = True
eqnat Zero     (Succ _) = False
eqnat (Succ _) Zero     = False
eqnat (Succ x) (Succ y) = eqnat x y
sumto :: Peano -> Peano
sumto Zero     = Zero
sumto (Succ x) = add (Succ x) (sumto x)
n5 :: Peano
n5 = add two three
n6 :: Peano
n6 = add three three
n17 :: Peano
n17 = add n6 (add n6 n5)
n37 :: Peano
n37 = Succ (mul n6 n6)
n703 :: Peano
n703 = sumto n37
n720 :: Peano
n720 = fac n6
boolToInt :: Bool -> Int
boolToInt False = 0
boolToInt True  = 1

main :: IO ()
main = (_primitive "pint32") (boolToInt (eqnat n720 (add n703 n17)))
```

Listing 4.4: Haskell program used for the sanity check of the Wasm 3.0 backend.

Backend	**Time (ms)**	**With Binaryen (ms)**	**Size (B)**
Wasm 3.0 Backend	0	0	0
Wasm Backend (Emscripten)	0	0	0
Native Backend	0	N/A	0

Table 4.1: Results of the sanity check comparing the Wasm 3.0 backend against the existing Wasm backend (through emscripten) and the native backend.

Plan

For part two of the thesis, we have one main goal, namely to be able to run a set of standard Haskell benchmarks on the Wasm 3.0 backend of MicroHs. To achieve this, we will need to perform the following tasks:

1. **Runtime Support for Necessary Combinators and Primitives:** There are two approaches we can take to achieve this. The first approach is to implement runtime support for all combinators and primitives that are used when we compile the benchmark programs using MicroHs. This involves implementing many combinators and primitives, which may be time-consuming and uninteresting for the purposes of this thesis. The second approach is to define a new simplified prelude that is optimized for generating as few different types of combinators and primitives as possible. This prelude would provide the necessary functionality for the benchmarks while minimizing the number of combinators and primitives that need to be implemented in the runtime. We will evaluate both approaches and choose the one that is most feasible within the time constraints of this thesis.

2. **Benchmark Selection and Setup:** We will select a subset of the Nofib benchmark suite [29] that does not rely on complex features such as concurrency, foreign function interfaces, or certain types of I/O. We will also need to set up an environment that can automatically run the benchmarks and collect performance data. Since we will be comparing both Wasm and native binaries, we will need to ensure that the measurements are fair and consistent across different platforms.

3. **Performance Evaluation:** There are multiple different performance comparisons we can make. First, we will compare the performance of the Wasm 3.0 backend against MicroHs's existing C backend. This will give us insight into the performance impact of using Wasm with GC compared to a traditional native backend. Second, we will compare the performance of the Wasm 3.0 backend against the MicroHs Wasm backend (through emscripten). This will allow us to evaluate the benefits of using Wasm with GC compared to traditional Wasm without GC. Third, we will compare the performance of the Wasm 3.0 backend against the GHC Wasm backend. To make this comparison fair, we can compare the relative performance of both backends to their respective native backends (i.e., MicroHs C backend and GHC's native backend). This will help us understand how well the Wasm 3.0 backend performs in relation to a mainstream Haskell compiler. Last, we can employ a Wasm-to-Wasm optimizer such as Binaryen [15] to optimize the generated Wasm binaries and evaluate the performance improvements that can be achieved through such optimizations and how this compares to the optimizability of other backends. For the general evaluation process we will make use of the ACM SIGPLAN Evaluation Guidelines [30].

If this main goal is achieved with sufficient time remaining, we can explore additional directions to further enhance the Wasm 3.0 backend of MicroHs:

1. **Lightweight Threads:** Implementing lightweight threads using stack switching can provide concurrency support in the Wasm 3.0 backend. Every thread would be represented by its own stack and the program could switch between these stacks to achieve concurrency. This would involve implementing a scheduler to manage the threads and ensure that they are executed fairly.

2. **Runtime Optimizations:** The current runtime system is not the most optimal, for example, strict primitives are implemented using recursion while this is not necessary. As a first step we can remove this recursion to potentially improve performance. Other directions are testing different stack allocation strategies for the LAS as well as removing unused combinators and primitives from the runtime when a program is compiled.

3. **Debugging Tooling:** Developing debugging tools specifically for the Wasm 3.0 backend can aid in diagnosing and fixing issues that may arise during development. This could include tools for inspecting the state of the runtime, visualizing the execution of programs, and identifying performance bottlenecks.

4. **JavaScript Interoperability:** Since we are targeting the web, adding support for the Haskell-JavaScript foreign function interface can enable Haskell code to interact with JavaScript code running in the same environment. This would potentially allow features such as DOM manipulation, event handling, and access to browser APIs from Haskell code.

Bibliography

[1] Victor Nazarov, Hamish Mackenzie, and Luite Stegeman. *GHCJS Haskell to JavaScript compiler.* 2023. URL: `https://github.com/ghcjs/ghcjs` (visited on 12/24/2025).

[2] Andreas Haas et al. “Bringing the web up to speed with WebAssembly”. In: *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation.* 2017, pp. 185–200.

[3] *GHC Wasm Meta Repository.* 2025. URL: `https://gitlab.haskell.org/haskell-wasm/ghc-wasm-meta` (visited on 12/24/2025).

[4] *Asterius: A Haskell to WebAssembly compiler.* 2022. URL: `https://github.com/tweag/asterius` (visited on 12/24/2025).

[5] Andreas Rossberg. *WebAssembly 3.0.* 2025. URL: `https://webassembly.org/news/2025-09-17-wasm-3.0/` (visited on 12/24/2025).

[6] Lennart Augustsson. “MicroHs: A small compiler for haskell”. In: *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium.* 2024, pp. 120–124.

[7] Alon Zakai. “Emscripten: an LLVM-to-JavaScript compiler”. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion.* 2011, pp. 301–312.

[8] Apoorva Anand. “Towards a WebAssembly Backend for MicroHs”. MSc Thesis. Utrecht University, 2025.

[9] Yorhel. *An Opiniated Survey of Functional Web Development.* May 2017. URL: `https://dev.yorhel.nl/doc/funcweb`.

[10] Atze Dijkstra et al. “Building javascript applications with haskell”. In: *Symposium on Implementation and Application of Functional Languages.* Springer. 2012, pp. 37–52.

[11] *Fay programming language.* 2021. URL: `https://github.com/faylang/fay` (visited on 12/27/2025).

[12] Anton Ekblad. “Towards a declarative web”. In: *Master of Science Thesis, University of Gothenburg* (2012).

[13] Evan Czaplicki. “Elm: Concurrent frp for functional guis”. In: *Senior thesis, Harvard University* 30 (2012).

[14] *PureScript.* URL: `https://purescript.org/` (visited on 12/27/2025).

[15] *Binaryen: A compiler and toolchain infrastructure library for WebAssembly.* URL: `https://github.com/WebAssembly/binaryen` (visited on 12/29/2025).

[16] *Support for WebAssembly (Wasm).* 2025. URL: `https://docs.flutter.dev/platform-integration/web/wasm` (visited on 12/29/2025).

[17] *Kotlin/Wasm.* URL: `https://kotlinlang.org/docs/wasm-overview.html` (visited on 12/29/2025).

[18] Léo Andrès, Pierre Chambart, and Jean-Christophe Filliâtre. "Wasocaml: compiling OCaml to WebAssembly". In: (2023).

[19] Andy Wingo. *Guile Hoot.* 2023. URL: `https://gitlab.com/spritely/guile-hoot` (visited on 12/29/2025).

[20] *Getting Started for J2CL/Wasm (Experimental).* 2025. URL: `https://github.com/google/j2cl/blob/master/docs/getting-started-j2wasm.md` (visited on 12/29/2025).

[21] Manuel Serrano and Robert Bruce Findler. "A Snapshot of the Performance of Wasm Backends for Managed Languages". In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes.* 2025, pp. 93–106.

[22] Alon Zakai. *A new way to bring garbage collected programming languages efficiently to WebAssembly.* 2023. URL: `https://v8.dev/blog/wasm-gc-porting` (visited on 12/29/2025).

[23] David A Turner. "A new implementation technique for applicative languages". In: *Software: Practice and Experience* 9.1 (1979), pp. 31–49.

[24] David A Turner and MS Joy. *SASL language manual.* University of Warwick, Department of Computer Science, 1987.

[25] David A Turner. "The semantic elegance of applicative languages". In: *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture.* 1981, pp. 85–92.

[26] David A Turner. "Miranda: A non-strict functional language with polymorphic types". In: *Conference on Functional Programming Languages and Computer Architecture.* Springer. 1985, pp. 1–16.

[27] Simon L Peyton Jones and Jon Salkild. "The spineless tagless G-machine". In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture.* 1989, pp. 184–201.

[28] Jeremy Gibbons and Nicolas Wu. "Folding domain-specific languages: deep and shallow embeddings (functional pearl)". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming.* 2014, pp. 339–347.

[29] Will Partain. "The nofib benchmark suite of Haskell programs". In: *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992.* Springer. 1993, pp. 195–202.

[30] *Empirical evaluation guidelines.* 2026. URL: `https://www.sigplan.org/Resources/EmpiricalEvaluation/` (visited on 01/26/2026).