EXTENSION OF ASSEMBLYSCRIPT FUNCTIONALITY THROUGH SOURCE-TO-SOURCE COMPILATION

An Honors Thesis Presented

By

GRIFFIN EVANS

Approved as to style and content by:

**** Marius Minea 10/11/23 18:36 ****
Chair

**** Emery D Berger 10/11/23 21:43 ****
Committee Member

**** Philip Sebastian Thomas 10/16/23 09:50 ****
Honors Program Director

ABSTRACT

WebAssembly is a language intended to be used to create high-performance programs that run within web pages. Following the model of traditional low-level assembly languages, it is designed to be a target for compilation from higher-level languages. AssemblyScript is a variant of TypeScript which is able to be statically compiled into WebAssembly code. Since TypeScript and its parent language JavaScript are already ubiquitous in web development, AssemblyScript acts as an easy platform for web developers to adapt to when needing to take advantage of the performance that WebAssembly offers. However, not all aspects of TypeScript are yet supported in AssemblyScript, including features such as closures, iterators, and *for...of* loops which are vital to many common design patterns used in TypeScript code.We introduce an extension to the AssemblyScript compiler which allows one to write AssemblyScript code using closures, iterators, and *for...of* loops by first transforming it via source-to-source compilation into behaviorally identical code which is compatible with the existing compiler. This enables programmers to more directly adapt existing code and design concepts from TypeScript into AssemblyScript and to write new code which more closely matches the capabilities of TypeScript while maintaining the performance benefits of AssemblyScript.

Table of Contents

1. Introduction

1.1. AssemblyScript and WebAssembly

WebAssembly [1][2][3] is a language intended to be used to create high-performance programs that run within web pages, but it is difficult for programmers to directly write code for it as it is modeled after traditional low-level assembly languages, rather than being a higher-level language like the other common languages used in web programming, particularly JavaScript. AssemblyScript [4] is a modified form of the language TypeScript (which in turn is a superset of JavaScript allowing for static typing) designed to be compiled into WebAssembly. This allows web programmers to take advantage of the performance benefits of WebAssembly without needing to learn a particularly different syntax or design philosophy from that which they are already familiar with from JavaScript and TypeScript. In fact in some instances it allows for existing TypeScript code to be adapted into a compiled form with minor or no change to the source itself, as most of AssemblyScript is a direct subset of the features and syntax of TypeScript.

However, AssemblyScript is ultimately a variant of TypeScript rather than strictly a subset. Though most of the features of AssemblyScript are taken directly from TypeScript, some additions and changes are made in order to allow the code to be statically compiled and to allow the features to better map to those of WebAssembly. While TypesScript enables type checking, AssemblyScript further enforces that checking be performed at compile time, meaning that overly dynamic features supported by TypeScript are removed, such as the `any` type [5], though generic types and nullable class and function types are still supported [6]. The types that are included also differ from those of TypeScript, to reflect the set of types available in WebAssembly—for instance, rather than having numbers be represented by the `number` or `bigint` types as in JavaScript and TypeScript, AssemblyScript uses a set of float and integer types with varying precisions in order to more closely reflect the types used by WebAssembly [6].

Not all features of AssemblyScript are limited to those natively built into WebAssembly however. A significant example of this is garbage collection. Most implementations of WebAssembly do not have garbage collection [7], instead requiring explicit instructions to clear memory. AssemblyScript provides its own garbage collection, with its default compilation settings adding a runtime to the compiled code which automatically performs incremental garbage collection [8][9]. However, there are other similar features, such as iterators, `for...of` loops, and closures, which would be possible to support in AssemblyScript but which are omitted as the developers plan to rely on further additions to WebAssembly in order to implement them more efficiently. The current lack of these features makes the task of adapting to AssemblyScript more difficult—both in preventing the direct reuse of existing code and in requiring programmers to learn new techniques and design patterns to work around the use of these features.

We introduce an extension [10] to the AssemblyScript compiler which allows one to write AssemblyScript code using closures and iterators by first transforming it via source-to-source compilation into behaviorally identical code which is compatible with the existing compiler. This enables programmers to write code which more closely matches the capabilities of TypeScript while maintaining the performance benefits of AssemblyScript.

1.2. Motivation

Web applications hold an ever-increasing importance in our ultra-connected modern world, and as uses for them increase so does the importance of being able to make software high-performance even when running it through a browser. The already widespread support of

WebAssembly contributes significantly to making such efficient web software possible, however the task of actually utilizing the potential efficiency that it allows for is still likely to be difficult for many web programmers. Due to the intricacy and precision needed to write WebAssembly code, various compilers exist to allow code to be written in higher level languages while still resulting in generally efficient WebAssembly code, analogously to how code is compiled into traditional assembly languages. For example, *Emscripten* compiles C and C++ into WebAssembly [11], and *wasm-pack* compiles Rust into WebAssembly [12]. Since the use of JavaScript is so prevalent in web programming (both for in-browser code and for back-ends using runtimes such as Node.js), many web programmers are likely far more experienced with JavaScript than with languages which are traditionally intended for static compilation—so even though compilers into WebAssembly exist from other languages, a web developer may still have difficulty making use of them as they may be unfamiliar with those languages. By compiling from a syntax much more familiar to most web programmers, AssemblyScript enables the avoidance of much of this difficulty, but its lack of some features common in ordinary JavaScript means that adapting code and techniques that use these features is still a challenge. Hence there is significant motivation for the expansion of AssemblyScript's capabilities in order to both better support the direct adaptation of existing code with minimal modifications, and in order to enable web developers to use a wider suite of design patterns while still gaining the potential performance benefits of AssemblyScript.

2. Background

Two commonly used concepts in JavaScript which are unsupported by AssemblyScript are closures and iterators. Closures are functions which store references to the state surrounding their declaration. This means one can create functions which are dependent on and affect the value of variables that were accessible in that state. For example, local variables declared within a function can be used by another function which is nested inside that function. Then since the state is bundled with the function in a closure, this inner function can continue to reference those variables even when being called from a scope that is no longer able to access those local variables.

Iterators are a design pattern which is used to provide a unified way to sequentially access elements within an object without requiring the code accessing it to be dependent on the object's underlying representation [13]. The exact details of the syntax and implementation differ across languages, but generally this involves the use of methods which, each time they are called, provide the next element in some sequence, meaning that all of the elements of a sequence can be accessed by repeatedly calling such a method. In languages like JavaScript, these methods are part of an object which is returned by a method of the data structure that one is iterating over.

The task of compiling closure-based code into WebAssembly has been previously considered in Chris Rybicki's Honors Thesis *Compiling from a Typed Dialect of Scheme to WebAssembly* [14], which details compilation from a dialect of Scheme, which includes closures as a feature, into WebAssembly. Rybicki's approach for supporting closures involves transforming lambda expressions (which are used to declare functions in Scheme) such that there are no free variables within their bodies, then extracting those expressions to the top-level and treating them like global functions. In order to do so, he adds a parameter representing the environment to the parameter list of each lambda, then makes all the free variables instead reference that parameter, then modifies all any application of these lambda expressions to include the relevant environment as an argument. He also adds existential types, which are given to environments that get passed into lambda expressions, in order to hide the internal representation of the environment from the code calling the lambda and to ensure only the matching environment is able to be passed into a particular lambda. Tuples containing a lambda and its environment are each “packed” into an existential type, which is then “unpacked” at the invocation, where “pack” and “unpack” are expressions introduced here to substitute out lambda expressions while checking for type. After closures have been converted as such, the compiler performs the aforementioned extraction, moving their definitions to be done in the top-level (globally) and replacing them with identifiers to the new globally scoped functions. These changes all occur before the program is actually transformed into WebAssembly code, instead restructuring the Scheme code itself into a form which can be more easily converted into WebAssembly in the later stages of the compilation process, so that it can be handled just as any code not using closures would.

Since AssemblyScript can already compile code not using closures, its compiler can be leveraged to allow for the functionality of closures through performing similar transformations before the compilation occurs. However, unlike Rybicki’s dialect of Scheme, the language itself was not built with such restructuring in mind, meaning that the transformation needs to rely on the already existing syntax for objects and methods, rather than being able to add additional types of expression.

The general concept of iterators is implemented in various different ways depending on the language, such as using objects or higher order functions. Some languages allow for iterators to use structured coroutines which yield the iterated values; this makes the process of writing code to define an iterator more convenient (as one does not need to account as much for keeping track of the pausing and restarting of the iteration) but has some limitations, particularly that it prevents the modification of the collection that one is iterating over during iteration, as one could with something like Java's iterators' remove operation, which removes the prior element reached in the iteration [15].

Jacobs et al. [13] demonstrated an approach to how C# code using iterators (including iterators nested within other iterators) could be transformed into identical code not dependent on the language having a built-in implementation of those features. This involves creating classes for each generator declaration such that the class has properties storing the current state of the iteration, where each state represents the most recently reached `yield` statement in the original code. Each time the `.next()` method is called it would then execute the original code until it reaches what would have been a `yield` statement, at which point it updates the state-tracking properties and returns an `IteratorResult` object. Since JavaScript's protocol for iterators does not require the implementation of any features besides calling `.next()`, it lacks difficulties such as the additional methods present in the more complicated iterators of other languages like Java.

Existing approaches to supporting similar features in languages which compile to WebAssembly have involved implementing extensions to the WebAssembly runtime, as demonstrated in the work of Pinckney et al. on Wasm/k [16] and developed further upon by Phipps-Costin et al. in their WasmFX [17]. Both introduce additional control flow features in the form of *delimited continuations*, which encapsulate a section of remaining computation and can be suspended and resumed in order to allow for non-local transfers of control. This allows compilers into WebAssembly to more directly represent features such as asynchronous functions and generators which are present in many high-level source languages [17]. These features can be supported by compilers using only existing WebAssembly features, but this generally results in complex and inefficient code. The WebAssembly runtime itself being modified to better support them allows for code with significant reductions in running time and file size—with code size being an especially important feature for web environments, which need to be able to download code on demand [16].

While these approaches provide performance advantages over those which rely on the existing features of WebAssembly, as is done in the approach we demonstrate, they are limited in their immediate utility as they are dependent on engines implementing those features. Given WebAssembly's goal of targeting a broad array of platforms both on and off the web [18], wide support is hugely desirable. However, if similar extensions become widely implemented within WebAssembly engines, then future work could allow AssemblyScript to take advantage of them to improve performance and implement features such as asynchronous generators [19] which would not otherwise be viable.

3. Methodology

3.1. Closures

3.1.1. Closures in JavaScript

A closure is a function which stores values from the context in which it was created. In ordinary JavaScript, every function declaration results in a closure. If one needs to access a variable from the bundled state inside the body of a function, one can simply use its identifier as one would when referencing a variable in any other context. For example, a function that returns a closure which returns a number that is incremented by one each time it is called could be created as such:

```
export function(from) {
  return function() {
    return ++from;
  }
}
```

This being an example of a closure as the returned function keeps a reference to the environment of the passed-in “from” value.

3.1.2. Limitations of AssemblyScript and its Compiler

AssemblyScript does not support the capturing of local variables. The AssemblyScript documentation states that the intent is to wait for WebAssembly engines to support built-in garbage collection and typed function references before officially supporting closures.

Currently, the AssemblyScript compiler restructures functions during compilation so that the resulting WebAssembly has each function declared in the global scope. Take for example the following code which contains nested functions that do not depend on capturing scope:

```
export function outside(): i32 {
  function inside(): i32 {
    return 1;
  }
  return inside();
}
```

This compiles in part to:

```
(export "outside" (func $code/index/outside))
 (export "memory" (memory $0))
 (func $code/index/outside~inside (result i32)
  i32.const 1
 )
 (func $code/index/outside (result i32)
...
```

This results in the two functions created separately, each with their own local scope. This means that each function only has access to values that are either passed in as arguments, initialized within the body of the function, or in the global scope.

3.1.3. Transformation of Scopes

The AssemblyScript compiler provides hooks which can be used to intercept and modify the compilation process. We use a hook after parsing to access the abstract syntax tree, which is a tree of objects representing the structuring of the elements that make up the parsed code. From this tree, we identify every function that references non-global variables declared outside of that function. These functions are the closures to be transformed, and the scope in which each of these variables was declared is the environment to be captured with that closure.

Once our extension identifies a scope that needs to be captured, it creates a class representing that scope call to that class's constructor. The class declaration is placed at the start of the file, as AssemblyScript only allows classes in the global scope.

3.1.3.1. Statements and Function Calls

For non-function blocks, the original contents of the block are moved into the constructor method of the class, and a call to the constructor is made in the original location of the block.

If a block is originally:

```
{
   statements
}
```

It is transformed into:

```
class scopeClass {
   constructor() {
      statements
   }
}
new scopeClass();
```

Each function that is transformed into a class has its body moved into a method of that class, called `.func`. This method takes the same arguments and has the same return type as the original function declaration.

```
function name(param0:type0, param1:type1, /* … ,*/ paramN:typeN):returnType {
   statements
}
```

Becomes:

```
class nameClass {
  constructor() {
  }
  func(param0:type0, param1:type1, /* … ,*/ paramN:typeN):returnType {
    statements
  }
}
let name = new nameClass();
```

Each call to the function in the original code is replaced with a call to the method stored in the object corresponding to that function, by affixing `.func` to that call.

```
name(param0, param1, /* … ,*/ paramN)
```

Is transformed into:

```
name.func(param0, param1, /* … ,*/ paramN)
```

3.1.3.2. Block-scoped Variables

Each variable declared within the original scope will become a public property of the class, with the same type and name as the original variable.

```
{
  ...
  let name:type;
  ...
}
```

```
class scopeClass {
  public name:type;
  ...
}
```

Declarations and assignments of variables which have had properties made for them are both transformed into assignments of the created property:

```
{
  ...
  let name1 = value1, name2 = value2, /* …, */ nameN = valueN;
  ...
}
```

```
{
  ...
  this.name1 = value1, this.name2 = value2, /* …, */ this.nameN = valueN;
  ...
}
```

Functions that access variables declared in a non-global scope besides their own need to be able to access and modify those variables even after the function is initially declared. In order to do so, their constructor takes in the object that was created to represent the scope in which the function is being created and stores it as a property of the new object.

```
class innerScopeClass {
  public parent: outerScopeClass;
  constructor(parent: outerScopeClass) {
    this.parent = parent;
  }
}
```

This allows for a linked list structure, in which the passed-in scope acts as a parent to the new object, meaning that variables which are in a further out scope can be accessed through accessing the parent of each scope.

Each time a variable from one of the transformed scopes is accessed, the identifier being used is prefixed with “`this.`” followed by zero or more repetitions of “`parent.`”, equivalent in count to how many scopes out the declaration is from the current expression. For example, given this code which declares a function within two outer blocks:

```
{
  ...
  let variable1 = value1;
  ...
  {
    ...
    let variable2 = value2;
    ...
    function name(/* arguments */):returnType{
      ...variable1...
      ...variable2...
    }
    ...
  }
  ...
}
```

We obtain class declaration for the function as follows:

```
class nameClass {
  public parent: scopeClass;
  constructor(parent: scopeClass) {
    this.parent = parent;
  }
  func(/* arguments */):returnType {
    ...this.parent.parent.variable1...
    ...this.parent.variable2...
  }
}
```

3.1.3.3. Exported Functions

Translated functions that were originally used in `export` declarations need to be transformed into functions with the same names, arguments, and types as the original functions, along with having classes created to represent them. This is necessary as exported functions may be accessed by code which has not been run through the compiler, and thus we cannot depend on being able to modify the code which imports the function.

For a function that is originally exported:

```
export function name(arguments):returnType {
  statements
}
```

The class will be created in the same manner as any other translated function. The resulting function makes a call to the constructor that was created for the corresponding class, then returns the results of a call to the constructed object's `.func` method which passes in the same arguments as are passed into this new function.

```
class name_class {
  constructor() {
  }
  func(arguments):returnType {
    statements
  }
}
export function name(arguments): returnType {
  return (new name_class()).func(arguments);
}
```

3.2. Iterable Objects

3.2.1. Iterators in JavaScript

In JavaScript and TypeScript, iterators are used within various built-in functions and statements. Often JavaScript code will employ the functionality of iterators without making any direct reference to them. For example, `for...of` statements provide the ability to make loops which are repeated for each element contained within a data structure [20]. In order to access these elements, the statement uses an iterator given by a method of the object for that data structure. Many built-in classes provide these methods automatically [21], allowing for example to use a loop to access each member of an array:

```
let array = [1,2,3,4];
for (let n of array) {
  console.log(n); //prints “1” “2” “3” “4”
}
```

This allows for more succinct loop syntax than would be necessary for conventional a conventional `for` loop stepping through an array:

```
for (let i = 0; i < array.length; ++i) {
  console.log(array[i]);
}
```

Additionally it allows `for` loops to be written more generically, so that they can be used for other objects without changes in syntax. For example, the same loop can be used with a string, as it also has a built-in iterator method:

```
let str = "ABCD";
for (let n of str) {
  console.log(n); //prints “A” “B” “C” “D”
}
```

This can be done with any object as long as that object is made following JavaScript’s *iterable* protocol [22]. Following this protocol, an *iterable* object is any object which has a method whose key is the value of the built-in constant `Symbol.iterator`. Symbols are a primitive type in JavaScript which are guaranteed to be unique upon creation. In order to prevent an overlap between the names that a programmer may want to use for custom methods and properties of an object and the keys used by operations built into JavaScript, certain "well-known" Symbols are kept constant and are accessible as static properties of the global `Symbol` object. Any function or statement which accesses the iterator of an object does so by calling whatever method is stored within the property of that object which has the key `Symbol.iterator`.

This method should return an *iterator*, which is an object which has a `next()` method, which returns an *IteratorResult* object. *IteratorResult* objects have the properties `done` and `value`. `done` is a boolean, which has the value `true` if the iterator has completed the sequence of its iteration, and is `false` (or absent) otherwise. `value` whatever is returned by the iterator, generally the elements of the object being iterated over. `value` can also be absent in the case that `done` is `true`.

3.2.2. Limitations of AssemblyScript and its Parser

In AssemblyScript, objects must have specified types, meaning that they must be made using the constructor function of a class. Because of this, JavaScript's object initializer syntax (e.g. `let object = {key:value}`) and `Object.create()` static method both cannot be used. To create an iterable object in AssemblyScript code, one would thus need to create three new class declarations: one for the *iterable*, one for the *iterator* which is returned by a method of the *iterable*, and one for the *IteratorResult* objects returned by the *iterator*.

The exact contents of the block depend on the purpose and internal structure of the class which is being made iterable, but a general outline of the necessary steps are as follows:

```
class iterableClass {
  [Symbol.iterator]() {
    return new iterator(this);
  }
}
class iterator {
  public iterableObject: iterableClass;
  constructor(iterableObject:iterable) {
    this.iterableObject = iterableObject;
  }
  next() {
    /* code which gets the values to be passed into iteratorResult */
    return new iteratorResult(value, done);
  }
}
class iteratorResult {
  public value: type;
  public done: bool;
  constructor(value: type, done: bool) {
    this.value = value;
    this.done = done;
  }
}
```

In AssemblyScript however, the programmer is unable to declare the *iterable* class in a way that follows the JavaScript specification, as one cannot use the constant `Symbol.iterator` as a key. To use a Symbol as a key, one must use the computed property name syntax [23], where square brackets are wrapped around the expression representing the key, as shown in *iterableClass* in the above code. While AssemblyScript has the Symbol data type and one can access the Symbols corresponding to global keys such as that of `Symbol.iterator`, AssemblyScript's parser does not support the computed property name syntax, preventing the use of Symbols and other non-identifier names as keys.

Besides the inability to use Symbols as keys, all of the functionality in the above code works in AssemblyScript. Hence one can make such code compilable if it is first transformed such that the iterable uses an identifier as the key, rather than a Symbol.

```
class iterableClass {
  constructor(/* arguments */) { /* ... */ }
  iteratorIdentifier() {
    return new iterator(this);
  }
  /*  any additional properties and methods */
}
/*  iterator and iteratorResult remain unchanged */
```

Since using a non-identifier as a key raises an error during parsing, rather than in the compilation done after the parse is complete, one cannot use any of the hooks provided by the frontend of the AssemblyScript compiler, as they are all called after parsing.

3.2.3. Transformation of Declaration and Creation of Iterators

Since the hooks cannot be used, transformations are done before the code is passed into the parser. The text of the source code is first read to find any class body in which `[Symbol.iterator]` appears in the declarations. Each bracketed key expression is then replaced with an ordinary identifier.

This change causes a class declaration as shown below:

```
class iterable {
  constructor(/* arguments */) { /* ... */ }
  [Symbol.iterator]() {
    statements
  }
}
```

To instead become:

```
class iterable {
  constructor(/* arguments */) { /* ... */ }
  iteratorIdentifier() {
    statements
  }
}
```

By default *iteratorIdentifier* is set to `Symbol_iterator`, however if the parse finds that sequence of characters to already exist within the code, it will prefix it with underscores until it becomes a unique sequence.

The same identifier is then used to replace any calls to the now-modified iterator method. Following the transformation shown above, an expression which initially reads:

```
iterableObject[Symbol.iterator]()
```

Would then be transformed into:

```
iterableObject.iteratorIdentifier()
```

3.3. `for...of` Loops

3.3.1. `for...of` Loops in JavaScript

`for...of` statements are a common way to make use of iterators in JavaScript, without requiring the programmer to manually access the iterator and its results' properties. A `for...of` statement has syntax as follows [20]:

```
for (variable of iterable)
  statement
```

When the `for...of` statement is run, it calls the `Symbol.iterator` method of *iterable* and then calls the `next()` method of the returned *iterator*. It assigns the `value` from the returned object to *variable*, then runs *statement*. It repeats this calling of `next` and execution of the statements until the `next()` method returns an object with a `true` value for the `done` property. The loop can also be ended with `break` or `return` statements or thrown errors, just as one can for an ordinary `for` loop or `while` loop.

3.3.2. Transformation of `for...of` Loops

To make the general form shown above into an ordinary `for` loop as supported by AssemblyScript's compiler, the iterator method needs to be explicitly called, and the returned iterator needs to be stored and have its `.next()` method called with each iteration until its `result` has a `true` value in its `done` boolean, with the *variable* being assigned to equal each result's `value` property.

The results of the transformation are dependent on the form of the *variable* expression in the original `for...of` statement. In JavaScript's syntax for this statement, *variable* can be either a declaration using any of `const`, `var`, or `let`, or a previously declared variable or property [20].

Given a `for...of` loop which uses a declaration for *variable*:

```
for (declarationKeyword variableName of iterable)
  statement
```

The transformation modifies it as follows:

```
for (declarationKeyword iterator = iterable.Symbol_iterator(),
                        result = iterator.next(),
                        variableName = result.value;
     !result.done;
     result = iterator.next(), variableName = result.value)
  statement
```

variableName, *iterable*, and *statement* are unchanged from the original code, and *iterator* and *result* are newly created identifiers. The new identifiers default to the names `iterator` and `result`, but if these names are already in use then they are prefixed with underscores until becoming ones that are not.

If *declarationKeyword* is `let` or `var`, it is kept the same as the original: a `for...of` loop using `let` results in a `for` loop using `let` and a loop using `var` results in a loop using `var`.

`let` and `const` declarations both use block scoping, differing only in that variables created with `let` can be reassigned while constants created with `const` cannot. Because of this, a constant created by `const` will behave identically to a variable created by `let` that never has an assignment operation applied to it. Thus any code which uses `const` and is syntactically valid can be made into equivalent code that uses `let`.

Though a `for...of` statement stores different values with the same identifier *variableName* across different iterations of the loop, this is not considered a reassignment as each iteration counts as a separate scope, with the variable being newly declared each time. This means that a `const` may be used as long as the identifier is not reassigned within *statement*.

In a `for` loop, though a new scope is made with each iteration similarly to a `for...of` loop, the values stored in the `let`-declared variables in this scope are initialized to be the same as in the previous iteration and must separately be assigned if one wants them to be updated to new values. In the transformed `for` loop shown above, reassignments are done to both *result* and *variableName* at the end of each loop iteration. This means that the transformed loop may not use `const` in its initialization. However, since `let` and `const` behave identically for any syntactically valid code, `let` can be used here instead. The transformation thus handles `const` declarations in the original `for...of` loops as if they were `let`, replacing *declarationKeyword* with `let` in the resulting transformation.

Unlike the declaration cases described above, `for...of` loops which reference an existing *variable* lack *declarationKeyword*:

```
for (variableName of iterable)
  statement
```

This means that in order for the values of *iterator*, *result*, and *variableName* to be assigned within the same initialization expression, *iterator* and *result* need to be already declared before the loop begins. The transformation thus adds a `let` statement declaring both of them before the `for` statement, and does not include any declarations in the loop initializer:

```
let iterator, result;
for (iterator = iterable.Symbol_iterator(),
      result = iterator.next(),
      variableName = result.value;
      !result.done;
      result = iterator.next(), variableName = result.value)
  statement
```

4. Results

Since existing AssemblyScript code does not utilize these features, benchmarks were created specifically to make use of them. For each benchmark, code was written that would be translated by our extension into ordinary AssemblyScript then compiled into WebAssembly. A TypeScript counterpart was made for each with minimal changes in code structure: types not present in TypeScript (e.g. `i32`, `f64`) were changed into their TypeScript counterparts (e.g. `number`), and `unchecked()` annotations (which are used in AssemblyScript to perform faster array access when indices are already known to be within bounds) were removed. The running time of each benchmark was measured using JavaScript's `performance.mark()` and `performance.measure()` methods, with code running in a cross-origin isolated webpage allowing for a minimum resolution of 5 microseconds [24]. For examples of the code used in each benchmark, see the appendix in section 7.

4.1. Closures

4.1.1. Correctness Testing

Correctness of the transformation was evaluated through the equivalence of the outputs of transformed programs using each of the extension's features with the same programs in TypeScript. Since our objective was to bring functionality from closures as they are supported in TypeScript into AssemblyScript, using the features must produce identical results in both languages. Our implementation of closures needs to properly handle functions which are declared within non-global scopes, regardless of the kind of enclosing block: `if` statements, `else` clauses, `while` loops, `for` loops (including when an inner function is dependent on the `for` loop's initialization; `for...of` loops don't need to be handled separately as they are converted into standard `for` loops in the other part of our transformation), `do...while` loops and other function bodies. Additionally, converted function bodies need to work for variables passed in as arguments as well as those declared within them.

A set of functions was made which each used different features of the closure transformations. This was repeated with each kind of enclosing scope, to ensure each behaves properly. This code was made to compile in both TypeScript and our extended AssemblyScript, with two versions that only differed in the types used: `f64` for AssemblyScript and `number` for TypeScript. The results were then checked for equality.

```
let alpha: i32 = 1;
function basicClosure(): i32 {
  return alpha;
}
```

```
let beta: i32 = 1;
function changingValueClosure(): i32 {
  beta++;
  return beta;
}
```

```
let gamma: i32 = 3;
function twoLayerClosure(): i32 {
  function innerClosure(): i32 {
    return gamma;
  }
  return innerClosure();
}
```

```
let epsilon: i32 = 6;
function sameValueA(): i32 {
  epsilon++;
  return epsilon;
}
function sameValueB(): i32 {
  epsilon *= 2;
  return epsilon;
}
```

```
let delta: i32 = 5;
function returningTwoLayerClosure(): ()=>i32 {
  function innerClosure(): i32 {
    return delta;
  }
  return innerClosure;
}
```

```
function returningStoringParameters(x:i32): () => i32 {
  function innerClosure(): i32 {
    return x;
  }
  return innerClosure;
}
```

All tests performed produced identical results in TypeScript as in our extended AssemblyScript.

4.1.2. Performance Metrics

Performance benefits were seen in cases involving the repeated use of a set of calculations. For example, taking advantage of the built-in math libraries, one can make a closure that performs a logarithm using a specified base without needing to recalculate the denominator each time:

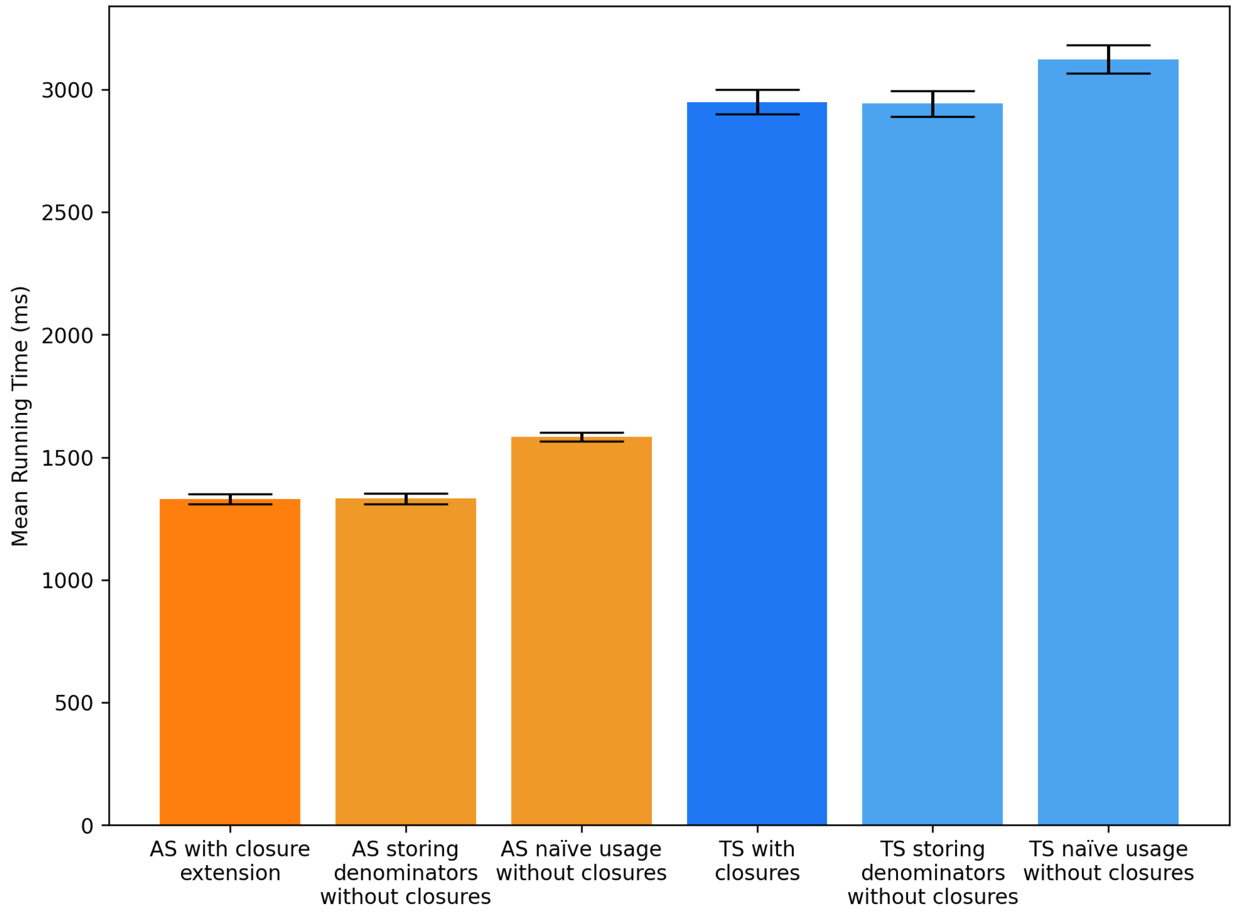
```
function setBase(base: f64) {
  const denominator: f64 = 1 / Math.log(base);
  function logBase(x: f64): f64 {
    return Math.log(x) * denominator;
  }
  return logBase;
}
```

Since the returned `logBase` function is dependent on the value of the denominator from the scope of the outer `setBase` function, it is transformed by our extension before the main compilation occurs. Just as with closures in ordinary JavaScript, the `setBase` function can be called several times with each returned closure having a separately stored value for its `denominator`. Similar behavior can be created without the need for closures, e.g. by storing the calculated `Math.log(base)` values in an array, but the ability to directly store those values with the functions allows for code to be created more flexibly and concisely.

The resulting transformed code for this closure was compared in performance to two approaches not using closures: one being shorter code which naïvely performs the logarithm on the base each time, while another stores the reciprocal of the values in an array. Using these approaches, we ran logarithms and exponentiation with an input array of bases on the first 10,000,000 natural numbers. These were compiled into WebAssembly and compared in running time against direct conversions of the same code into ordinary TypeScript which was compiled into JavaScript.

All three of the AssemblyScript approaches ran substantially faster than TypeScript, as one would expect given the performance advantages of WebAssembly in the operations relevant to these trials. The transformed closure and value storing methods were the fastest, each having an average running time of 1330 milliseconds. The approach which does not store the denominators took 19% longer than the transformed closures on average, while all three of the TypeScript versions were significantly slower: the fastest, storing the denominators without taking advantage of closures, took 121% longer than the fastest AssemblyScript method. TypeScript using the closure approach performed slightly worse, taking 122% longer than the same approach in AssemblyScript, with the non-storing approach again performing the worst of the three.

Figure 1: Benchmark: Closures of logarithms with varying bases



Benchmark type	Mean Time (ms)	Sample Standard Deviation
AssemblyScript with closure extension	1330	21
AS storing denominators without closures	1330	22
AS naïve usage without closures	1580	18
TypeScript with closures	2950	50.
TS storing denominators without closures	2940	52
TS naïve usage without closures	3120	58

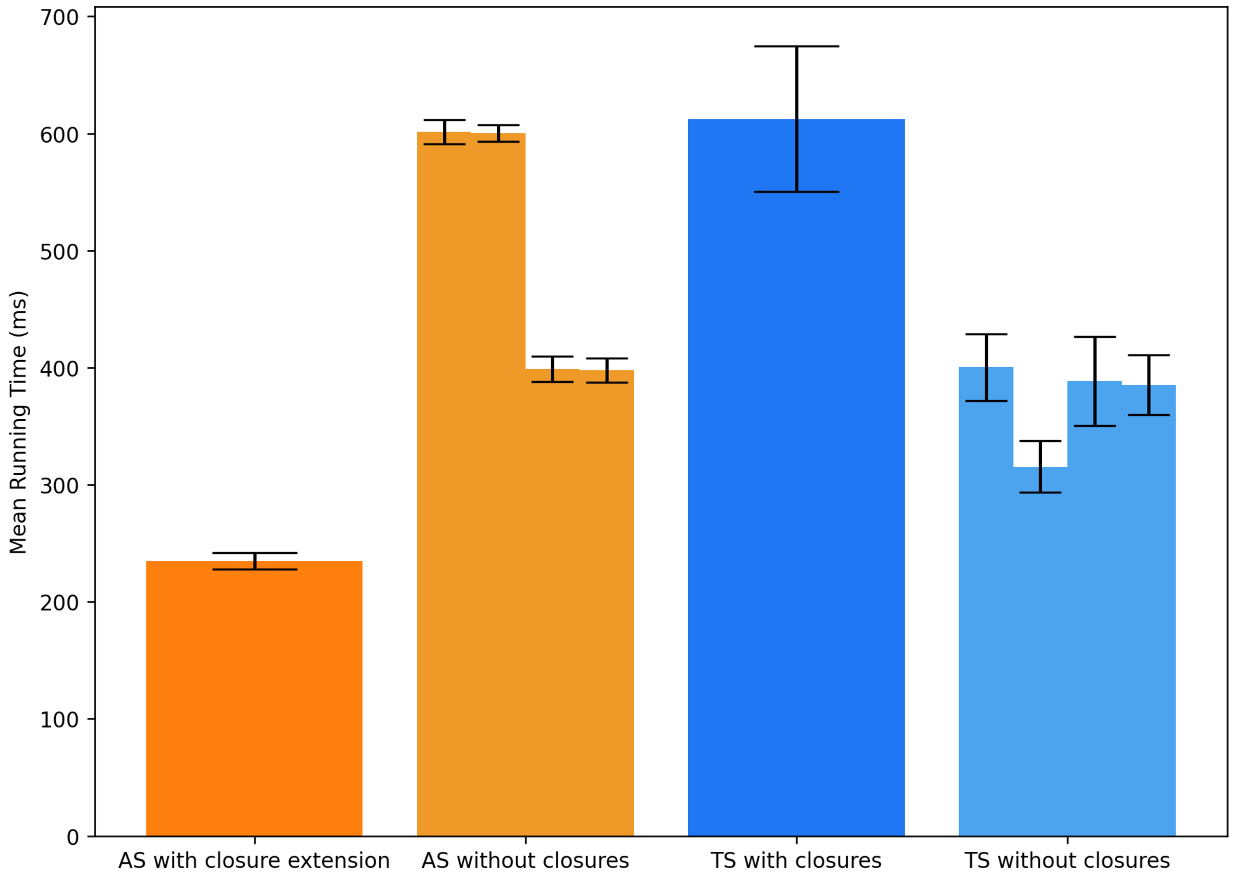
Since the closure-transforming version appeared to have no speed disadvantage as compared to the other AssemblyScript approaches—in fact being the fastest by a narrow margin—the overhead added by the additional constructor and method calls made by the transformation appears to be negligible. Alongside the fact many calculation-heavy behaviors can be performed more efficiently through code compiled using AssemblyScript, this means that we can take advantage of the utility of closures at the same time as improving performance.

Performance advantages were also seen in benchmarks testing the storing of functions alongside variables that are modified with each call. In these benchmarks, an array is filled with various functions that take two floats as arguments. A closure is then made for each of these functions. Each closure takes one float and passes it in as the first argument of the stored function, with the second argument being a variable that is incremented each time the closure is called.

```
function makeIncrementY(f:(x:f64,y:f64)=>f64, y:f64):(x:f64)=>f64 {
  function incrY(x:f64):f64 {
    return f(x,y++);
  }
  return incrY;
}
```

The benchmark then calls each of the closures repeatedly and returns an output made by passing the result of each function into the next. This code was then modified to replicate the same behavior without relying on closures. Four variations were made, with differing amounts of adherence to the structure of the original closure-based version of the function. These functions were all then ported into TypeScript with minimal modifications to account for the differing types and default libraries.

Figure 2: Benchmark: Closures storing varying functions and incrementing variables



Benchmark type	Mean Time (ms)	Sample Standard Deviation
AssemblyScript with closure extension	235	7.1
AS without closures version 1	601	10.
AS without closures version 2	600	7.3
AS without closures version 3	399	11
AS without closures version 4	398	10.
TypeScript with closures	612	62
TS without closures version 1	400.	29
TS without closures version 2	315	22
TS without closures version 3	389	38
TS without closures version 4	385	26

The AssemblyScript code which used transformed closures ran substantially faster than the same code ported into TypeScript, with the TypeScript version taking 612 milliseconds, 160% longer than the 235 milliseconds of the AssemblyScript version.

As well as being more complicated to write, the non-closure AssemblyScript versions had slower performance—the fastest of them still on average took 69% longer than the closure version. In fact they all performed worse than most of their TypeScript counterparts—the second fastest of the benchmarked functions was one of the non-closure functions run in TypeScript, which took only 34% longer than the AssemblyScript with transformed closures.

Since our extension's translation results in ordinary AssemblyScript, such fast performance must be obtainable without using the extension. The implementations without closures kept numbers and functions in separate arrays, but all performed worse than the transformation closures. This suggests that the performance differences may be due to a difference in memory layout. The transformed closures result in a single array storing objects which contain the values of each function and its related number together, while having separate arrays requires disparate locations in memory to be accessed each time. As the performance difference appears to be tied to this multi-array approach, the creation of a class to store functions paired with numbers may be necessary for optimal performance in this benchmark. Our extension creates this class automatically and allows for terser and simpler code than one would have by manually creating this class. Hence the extension allows for code to be competitive in performance while remaining simpler to read and write than other equivalent code.

This also illustrates that the scenarios in which translated closures result in performance benefits in AssemblyScript are not necessarily the same as scenarios where closures are optimal in TypeScript and JavaScript. While the approach using translated closures was the fastest of the AssemblyScript tests, the equivalent code in TypeScript ran the slowest of all the tests.

4.1.3. Limitations

While the objects we create from closures can be passed as arguments into functions, transformation of the called function is necessary. Since closures are transformed into objects that cannot be directly invoked as functions, instead having methods containing their original function body, any attempt to invoke a closure that has been passed in as an argument must be replaced with a call to the `.func` method of that argument. AssemblyScript code which passes an object's method directly into another function fails at runtime due to the mismatch in types caused by the argument type including the "`this`" context. In order to keep access to the proper context, the entire object must be passed into the function, with the invocations within that function being modified to call the method. Similarly, the called function must have the types of its parameters modified in order to match the class made for the closure being passed in.

Within the code being transformed, we are able to perform both of these changes. Expressions being invoked without a call signature are identified by the extension and have `.func` added. Similarly, assignment of values to variables which differ in type are identified at compile time, and the type signatures are replaced. However, in order to ensure that only the relevant types are changed, the extension currently requires them to be explicitly annotated in the code (see Appendix 7.2.1 for an example of such).

Contrastingly, these changes cannot be performed on functions from other sources, particularly those from the built-in library of AssemblyScript or those imported from a separate file. This means that the closures we transform cannot be used with built-in methods such as an array's .map or .forEach function.

Additionally, since the extension produces a separate class for each transformed function declaration, parameters and variables cannot have the same type as several different closures. This means that all closures being stored in the same variable or passed into the same function must be created from the same original function—they can differ in stored environment, such as when a closure returned by a function varies with the value of the outside function’s arguments, but not in the contained function body.

4.2. Iterators

4.2.1. Correctness Testing

As with closures, the correctness of the transformation of iterators was evaluated through the creation of programs which were made to use the features in both TypeScript and our extended AssemblyScript. This includes class declarations with `[Symbol.iterator]` methods, calls to the methods of objects from those classes, and `for...of` loops utilizing those classes. Classes can only be declared in the global scope in AssemblyScript, while method calls and `for...of` loops are able to be used in any scope and hence have their tests repeated for each potential scope. Additionally, the testing of `for...of` loops is repeated for each possible *variable* syntax: `const`, `let`, and `var` declarations as well as previously declared variables and properties.

For these tests, the following classes were made to act as an example implementation of the iterator protocol. The `NaturalGeneratorIterable` class here has an iterator which returns natural numbers from 0 to a `max` value that is passed in as an argument.

```
class NaturalGeneratorIterable {
    max: i32;
    constructor(max: i32) {
        this.max = max;
    }
    [Symbol.iterator](): NaturalGeneratorIterator {
        return new NaturalGeneratorIterator(this);
    }
}
class NaturalGeneratorIterator {
    source: NaturalGeneratorIterable;
    value: i32;
    constructor(source: NaturalGeneratorIterable) {
        this.source = source;
        this.value = 0;
    }
    next(): IteratorResult<i32> {
        if (this.value > this.source.max) {
            return new IteratorResult<i32>(true, 0);
        } else {
            return new IteratorResult<i32>(false, this.value++);
        }
    }
    [Symbol.iterator](): NaturalGeneratorIterator {
        return this;
    }
}
class IteratorResult<T> {
    done: boolean;
    value: T;
    constructor(done: boolean, value: T) {
        this.done = done;
        this.value = value;
    }
}
```

This class's iterator was used manually, making calls directly to `[Symbol.iterator]` and the returned object's methods:

```
let manualIterable = new NaturalGeneratorIterableSimple(2);
let manualIterator = manualIterable[Symbol.iterator]();
let result1 = manualIterator.next();
result1.value.toString() + " "; // 0
result1.done.toString() + " "; // false; iteration incomplete
manualIterator.next().value.toString(); //1
let result3 = manualIterator.next();
result3.value.toString() + " "; // 2
result3.done.toString() + " "; // true; iteration complete
```

As well as in `for` loops, both as an iterable stored in a variable to be used with several for loops and as an object declared within the head of the loop:

```
let storedIterable1 = new NaturalGeneratorIterableSimple(3);
//iterating over a class stored in a variable
for (let i of storedIterable1) {
  i.toString() + " "; // "1 2 3 "
}

//iterating again over an already-iterated-over iterable
for (let i of storedIterable1) {
  i.toString() + " "; // "1 2 3 "
}

//iterating over a newly created iterable
for (let i of new NaturalGeneratorIterableSimple(4)) {
  i.toString() + " "; // "1 2 3 4 "
}
```

These loops were repeated with each of the possible assignment targets listed above, and both the loops and manual iterations were repeated in different kinds of scope as was done for the closures (`if` statements, `else` clauses, `while` loops, `for` loops, `do...while` loops and function bodies). Each produced identical results when compiled in TypeScript (using the `number` type) as in our extended AssemblyScript (using the `i32` type).

4.2.2. Performance Metrics

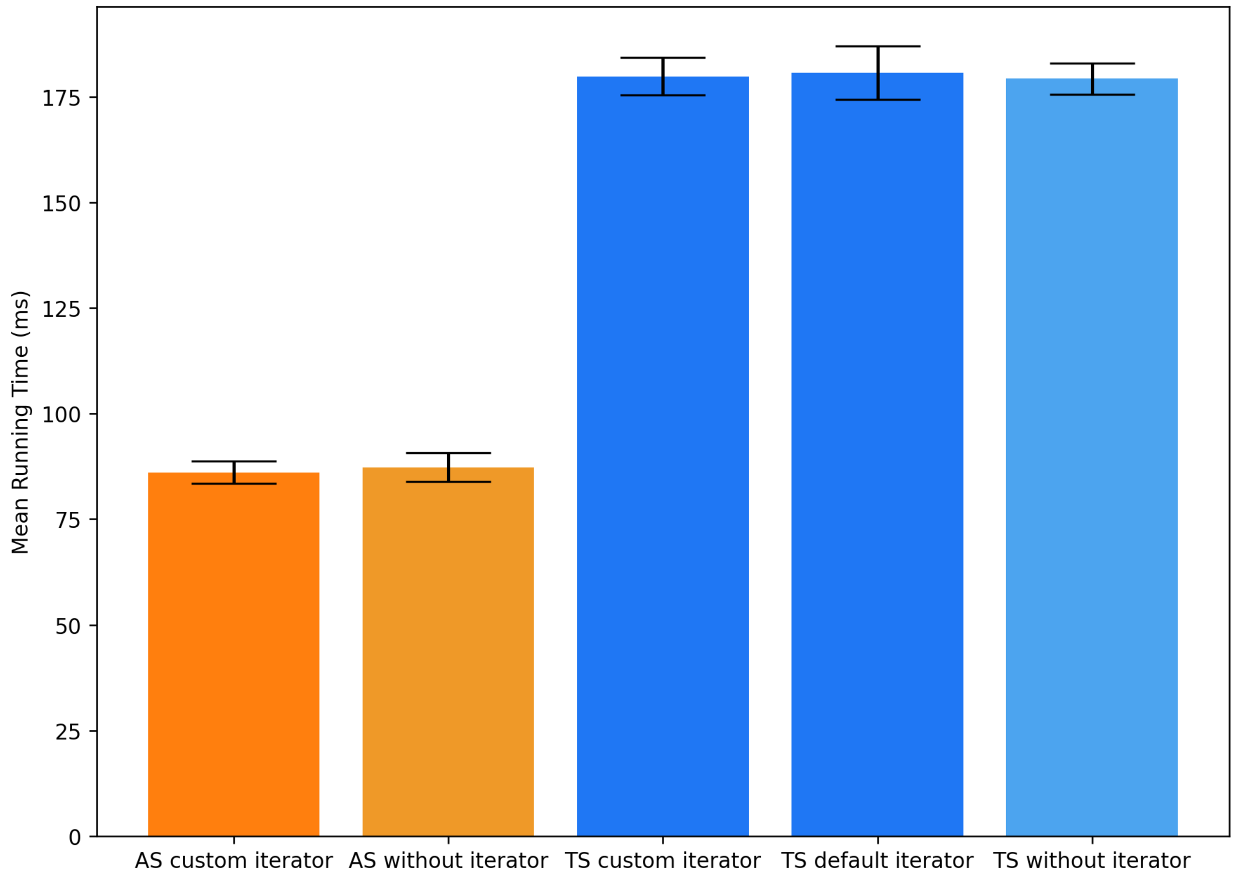
Translation of iterators does not appear to have direct performance advantages, but it is still valuable for permitting easier code writing. Once an iterable type has been made, later code can loop through the iterable's elements without the need to rewrite any logic particular to that type. Similarly, once a `for...of` loop has been written making use of one iterable type, it can be easily adapted to support any other iterable type without necessitating any change in the structure of the loop.

Due to the overhead involved in the additional objects being created, code making heavy use of iterators may perform slower than using an alternative approach natively supported by AssemblyScript. However, this does not mean that all use cases of transformed iterators will be inherently non-performant. Particularly, in cases where a `for...of` loop is used as the outermost layer of a set of nested loops, the translated code appears to have equal or marginally faster speed than similar code which does not use iterators.

For example, one benchmark stores an array of 500 floats and loops through them, using each float as a base for logarithms and exponents. Since AssemblyScript's Array class lacks the built-in iterator method that its TypeScript equivalent has, the array is stored inside a separately declared object which has an iterator that returns each member of the stored array.

Both when using a `for...of` loop with an iterator and when using a standard `for` loop with an incrementing index, AssemblyScript performed significantly faster than the TypeScript counterparts of the same code. Within the TypeScript code, no significant performance difference appeared between using a separate iterable class as was done in the AssemblyScript code and using the built-in iterator of the Array class.

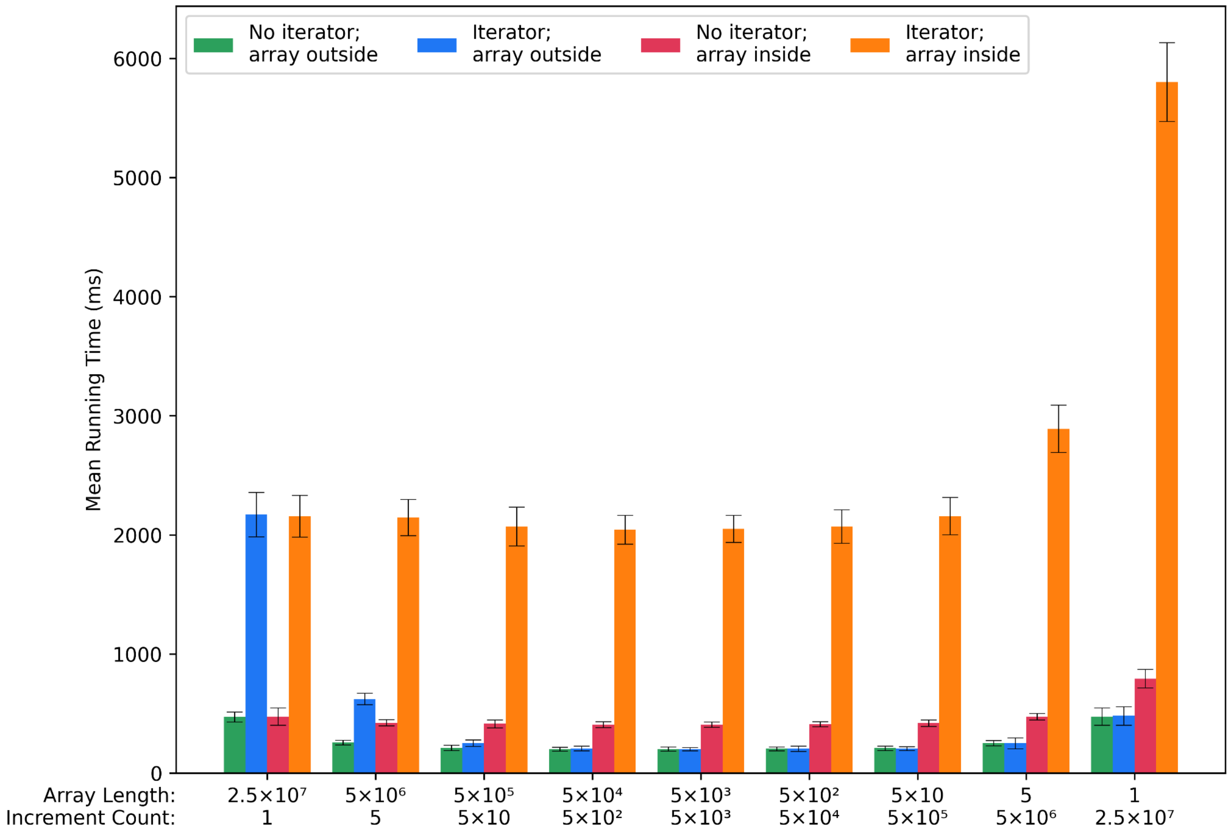
Figure 3: Benchmark: Iteration over array of bases for logarithms and exponents



Benchmark type	Mean Time (ms)	Sample Standard Deviation
AssemblyScript custom iterator	86	2.6
AssemblyScript without iterator	87	3.4
TypeScript custom iterator	180.	4.4
TypeScript default iterator	181	6.3
TypeScript without iterator	179	3.7

Since this code's performance does not appear to be detrimentally affected by the added iterator, the performance benefits of AssemblyScript can be taken advantage of while enabling structural possibilities that are normally unavailable in AssemblyScript.

Figure 4: Benchmark: Iteration over array nested with incrementation loop, with varying loop nesting and number of iterations



Whether such performance is attainable is dependent on the structure of the code and the number of elements being iterated over. The transformed iterators appear to suffer in performance when large numbers of iterators are being created—such as when `for...of` loops are nested within other loops, as overhead is introduced by the additional method calls and object creation.

Even when a single iterator is used, performance can begin to suffer when enough iterations are being performed. In benchmarking varying lengths of array iterations, arrays with up to 50,000 elements had negligible differences in performance between iterators and traditional `for` loops. Significant differences appear when iterating over 500,000 elements, with an 19% longer running time for the transformed iterator compared to an incrementing `for` loop, though this is still a relatively minor disadvantage, with the mean running times being less than two sample standard deviations apart from each other. Further increasing the length to 5,000,000 elements caused more drastic disparity, with iterators taking about 140% longer than a traditional `for` loop. These differences only appearing with large numbers of elements suggests that they are potentially related to the allocation of memory—when iterating over so many elements, the *IteratorResult* objects being created may not all be being removed from memory immediately, causing fewer spaces in memory to be available and introducing additional overhead. On the hardware used for benchmarking, this degradation of performance occurred even with the maximum possible memory (2^16 pages of 64KiB each [25]) being initially allocated to the WebAssembly runtime. Supposing that the performance limitations are connected to memory, this may suggest that the problems occurring have more to do with the layout of the memory and the speed at which the system can access different regions of memory rather than solely to the amount of space available. One may be able to obtain better performance in the same benchmark through other changes in compilation options, such as by adding manual invocations of the garbage collector to clear memory before it becomes necessary to reallocate it [9], or it may be influenced by limiting factors of the particular hardware used.

The exact performance details may differ with each iterable class, but for arrays this suggests that transformed iterators and `for...of` loops may be used without damaging performance when less than hundreds of thousands of elements are being iterated over. When iterating over hundreds of thousands of elements, the performance may begin to suffer in comparison to using standard `for` loops. However, as the effects are not necessarily immense, using iterators may still be preferable if prioritizing the kind of code flexibility enabled by them. Beyond that range, dealing with millions of elements, the detrimental effect on performance will likely outweigh the benefits of the enabled structures.

4.2.2. Limitations

Since AssemblyScript lacks native support for iterators, its built-in classes do not provide any iterators by default. This thus necessitates the creation of one's own classes and methods in order to utilize the features of our extension. For example, in TypeScript and JavaScript, arrays are iterable by default, while with this extension one needs to write a wrapper class with its own `Symbol.iterator` method in order to use them in `for...of` loops. This reduces the initial convenience enabled by supporting iterables, however there are still benefits when several iterations are being performed. Once an iterator method is written for a class it can be reused for any other objects of that class, and iterators even of different types can be interchanged while using the same loop structure.

While the lack of default iterator methods does make code more complicated to write, it does not appear to have significant effects on performance. In our benchmarks, TypeScript code using a user-provided iterator on a class which stores a single array did not perform any slower than code which used the default array iterator.

Though code including declarations of iterable classes can be compiled through the transformations, not every possible syntax for such declarations is supported. Particularly, this only enables the creation of classes which create methods by directly using the `Symbol.iterator` static property as a key within the class body. In ordinary TypeScript, one could store the value of that property in another variable and then declare a class which names a method using that variable. In AssemblyScript all properties of classes must be statically declared [8], meaning that one cannot use keys which depend on the value of a variable at runtime.

5. Conclusions

We have demonstrated an approach which provides support for closures, iterators, and `for...of` statements in AssemblyScript through an algorithmic transformation into ordinary AssemblyScript code. This allows for code structure closer to what is possible in TypeScript while showing similar or improved performance compared to analogous AssemblyScript code. As well as the direct utility in ease of programming that this provides, it also suggests that we may be able to replicate other features not provided in AssemblyScript while still leveraging the existing compiler.

Improvements to the translation process could be made possible by avoiding the use of the existing hooks. One could do so either by using a parser separate from AssemblyScript's for all of the transformations or by modifying the code of AssemblyScript's parser itself so that it can handle all of the pre-transformation structures. This would allow all of the transformations to be applied in a single pass over the code, rather than relying on separate steps to process the iterators and closures. It would also be useful as futureproofing, as the current dependence of our extension on AssemblyScript's hooking may cause incompatibilities if the compiler's API changes in the future.

A potential expansion of the project would be to enable separately declared closures to match in object type, allowing different closures to be used as arguments of the same function or members of the same array. One could identify which closures share the same parameters and return types and create a parent class that each closure's object would extend. By including a placeholder `.func` with the same parameters and return type in the parent class, we can use the parent type for each variable and parameter which needs to store any of the child classes and still be able to access the method of each object using `.func` calls as we do currently. Closure transformation could also be expanded to support other forms of creating functions.

Currently, we focus only on the support of traditional function declarations, ignoring arrow function expressions. While most of the behavior of arrow functions is identical to traditionally declared functions, there are differences in scope binding behavior that may need to be accounted for by the translation.

Another potential future step would be to expand the transformation of computed property names. Since AssemblyScript does not support any way to dynamically add properties, there is likely no way to recreate the behavior of all possible expressions, but those which have values that are guaranteed to be predictable at compilation time could be translated into valid AssemblyScript which behaves identically. For example a constant which is set to `Symbol.iterator` at declaration then used as a key can be seen at compile time to behave identically to using `Symbol.iterator` itself. This alongside the replacing of other Symbols used in property declarations with unique identifiers could permit the replication of the behavior of Symbols beyond just `Symbol.iterator`. While Symbols are implemented within the standard library of AssemblyScript [26], they are limited in functionality as they are unable to be used as keys for properties [27].

Iterator transformation could also benefit from adding support for other related features of TypeScript and JavaScript. While the extension allows iterator classes to be manually written and used in `for...of` loops, it does not provide any replacement for the built-in iterators of JavaScript which are not present in AssemblyScript's standard library [21]. The extension could be modified to provide its own implementation of these iterators, which would allow `for...of` loops to be used with built-in classes like Arrays as they are in ordinary TypeScript without requiring additional coding from the user. Supporting translation of the `Generator` class and its associated `function*` syntax would further reduce the coding necessary by allowing for simpler and more concise declarations of iterators [28].

6. References

[1] A. Rossberg, B. Titzer, A. Haas, D. Schuff, D. Gohman, L. Wagner, A. Zakai, J.F. Bastien, and M. Holman, “Bringing the Web Up to Speed with WebAssembly,” in *Communications of the ACM Vol. 61 No. 12*, 2018. Available at https://dl.acm.org/doi/pdf/10.1145/3282510.

[2] MDN contributors. *WebAssembly*, 2023. Available at https://developer.mozilla.org/en-US/docs/WebAssembly. Retrieved August 25, 2023.

[3] *WebAssembly*, 2023. Available at https://webassembly.org/. Retrieved August 25, 2023.

[4] The AssemblyScript Authors. *Introduction*, 2023. Available at https://www.assemblyscript.org/introduction.html.

[5] The AssemblyScript Authors. *Concepts*, 2023. Available at https://www.assemblyscript.org/concepts.html.

[6] The AssemblyScript Authors. *Types*, 2023. Available at https://www.assemblyscript.org/types.html.

[7] *Roadmap*, 2023. Available at https://webassembly.org/roadmap/. Retrieved August 25, 2023.

[8] The AssemblyScript Authors. *Implementation Status*, 2023. Available at https://www.assemblyscript.org/status.html.

[9] The AssemblyScript Authors. *Runtime*, 2023. Available at https://www.assemblyscript.org/runtime.html.

[10] https://github.com/grievans/Transpiler-Project

[11] Emscripten Contributors. *Emscripten 3.1.45-git (dev) documentation*, 2015–2023. Available at https://emscripten.org/. Retrieved August 25, 2023.

[12] Ashley Williams, 2018. Available at https://github.com/rustwasm/wasm-pack. Retrieved August 25, 2023.

[13] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte, *Iterators revisited: proof rules and implementation*, Formal Techniques for Java-like Programs, 2005. Available at http://www.cs.ru.nl/~erikpoll/ftfjp/2005/Jacobs.pdf.

[14] C. Rybicki, *Compiling from a Typed Dialect of Scheme to WebAssembly*, Honors Thesis, University of Massachusetts Amherst, 2020. Available at https://people.cs.umass.edu/~arjun/archive/rybicki-honors-2020.pdf.

[15] J. Liu, A. Kimball, and A. C. Myers. “Interruptible iterators,” in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06)*, pages 283–294, Association for Computing Machinery, New York, NY, USA, 2006. Available at https://doi.org/10.1145/1111037.1111063.

[16] D. Pinckney, A. Guha, and Y. Brun. “Wasm/k: delimited continuations for WebAssembly,” in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2020)*, pages 16–28. Association for Computing Machinery, New York, NY, USA, 2020. Available at https://doi.org/10.1145/3426422.3426978.

[17] L. Phipps-Costin, A. Rossberg, A. Guha, D. Leijen, D. Hillerström, KC Sivaramakrishnan, M. Pretnar, and S. Lindley. “Continuing WebAssembly with Effect Handlers,” 2023. Available at https://doi.org/10.48550/arXiv.2308.08347

[18] *Portability - WebAssembly*, 2023. Available at https://webassembly.org/docs/portability/. Retrieved October 6, 2023.

[19] MDN contributors. *AsyncGenerator - JavaScript*, 2023. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AsyncGenerator. Retrieved October 6, 2023.
[20] MDN contributors. *for...of - JavaScript*, 2023. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of. Retrieved August 25, 2023.
[21] MDN contributors. *Iterator - JavaScript*, 2023. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Iterator#description. Retrieved August 25, 2023.
[22] MDN contributors. *Iteration protocols - JavaScript*, 2023. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols. Retrieved August 25, 2023.
[23] MDN contributors. "Computed Property Names," in *Object initializer - JavaScript*, 2023. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#computed_property_names. Retrieved August 25, 2023.
[24] MDN contributors. "High precision timing," 2023. Available at https://developer.mozilla.org/en-US/docs/Web/API/Performance_API/High_precision_timing. Retrieved October 10, 2023.
[25] MDN contributors. "WebAssembly.Memory() constructor," 2023. Available at https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Memory/Memory. Retrieved October 10, 2023.
[26] The AssemblyScript Authors. *Symbol*, 2023. Available at https://www.assemblyscript.org/stdlib/symbol.html.
[27] MDN contributors. *Symbol - JavaScript*, 2023. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol. Retrieved August 25, 2023.
[28] MDN contributors. *Generator - JavaScript*, 2023. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator. Retrieved August 25, 2023.

7. Appendix: Benchmark Code and Transformation Output Examples

7.1. “Closures of logarithms with varying bases”

See Figure 1 in section 4.1.2.

7.1.1. AssemblyScript with Closure Extension

```
//In this particular example, a closure approach is lengthier but
may be preferred for its clearer delineation of sections of the
function (keeping the logarithm code separate from the code using
those logarithms). The extension's advantages are more relevant in
longer code, e.g. that which needs to perform logarithms in
multiple places throughout, as it allows for the functions to be
reused later or exchanged for other operations with more ease than
in 7.1.2 or 7.1.3.
export function logNaturalsArrTest(maxCount: f64, bases:
f64[]):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (bases.length < 1) {
        return result;
    }
    function setBase(base: f64) {
        const denominator: f64 = 1 / Math.log(base);
        function logBase(x: f64): f64 {
            return Math.log(x) * denominator;
        }
        return logBase;
    }
    const logFuncs = [setBase(bases[0])];
    for (let i = 1; i < bases.length; ++i) {
        //"unchecked" annotation used in AssemblyScript to skip
extra checks on array access, for cases where the index is already
known to be in bounds
        unchecked(logFuncs.push(setBase(bases[i])));
    }
    for (let i = 0; i < maxCount; ++i) {
        let val:f64 = i;
        for (let j = 0; j < logFuncs.length; ++j) {
            unchecked(val = logFuncs[j](val));
        }
        for (let j = bases.length - 1; j >= 0; --j) {
            unchecked(val = bases[j] ** val);
        }
        unchecked(result[i] = (val));
    }
    return result;
}
```

7.1.1.1. Source-to-Source Compilation Output

```
class setBase_0_class_parameters {
public denominator: f64;
constructor(base: f64){
this.base = base;
}
public base: f64;
}

class logBase_1_class {
public parent: setBase_0_class_parameters;
constructor(parent: setBase_0_class_parameters){
this.parent = parent;
};
func(x: f64): f64 {
            return Math.log(x) * this.parent.denominator;
        }
}
export function logNaturalsArrTest(maxCount: f64, bases:
f64[]):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (bases.length < 1) {
        return result;
    }
    function setBase(base: f64):logBase_1_class {
let setBase_0_class_parameters_obj = new
setBase_0_class_parameters(base, );
        setBase_0_class_parameters_obj.denominator= 1 /
Math.log(setBase_0_class_parameters_obj.base);
```

```
        let logBase = new
logBase_1_class(setBase_0_class_parameters_obj);
        return logBase;
    }

    const logFuncs = [setBase(bases[0])];
    for (let i = 1; i < bases.length; ++i) {
        unchecked(logFuncs.push(setBase(bases[i])));
    }

    for (let i = 0; i < maxCount; ++i) {
        let val:f64 = i;
        for (let j = 0; j < logFuncs.length; ++j) {
            unchecked(val = logFuncs[j].func(val));
        }
        for (let j = bases.length - 1; j >= 0; --j) {
            unchecked(val = bases[j] ** val);
        }
        unchecked(result[i] = (val));
    }
    return result;
}
```

7.1.1.2. TypeScript Conversion

```
//Change in types ("f64" and "i32" are instead "number") and
removal of "unchecked()" annotations
export function logNaturalsArrTest(maxCount: number, bases:
number[]):Float64Array {
    const result = new Float64Array(maxCount);
    if (bases.length < 1) {
        return result;
    }
    function setBase(base: number) {
        const denominator: number = 1 / Math.log(base);
        function logBase(x: number): number {
            return Math.log(x) * denominator;
        }
        return logBase;
    }
    const logFuncs = [setBase(bases[0])];
    for (let i = 1; i < bases.length; ++i) {
        logFuncs.push(setBase(bases[i]));
    }
    for (let i = 0; i < maxCount; ++i) {
        let val: number = i;
        for (let j = 0; j < logFuncs.length; ++j) {
            val = logFuncs[j](val);
        }
        for (let j = bases.length - 1; j >= 0; --j) {
            val = bases[j] ** val;
        }
        result[i] = val;
    }
    return result;
}
```

7.1.2. AssemblyScript Storing Denominators without Closures

```
export function logNaturalsArrTestNoClosureSavingDenoms(maxCount:
f64, bases: f64[]):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (bases.length < 1) {
        return result;
    }
    const denoms: f64[] = [];
    for (let i = 0; i < bases.length; ++i) {
        unchecked(denoms.push(1 / Math.log(bases[i])));
    }
    for (let i = 0; i < maxCount; ++i) {
        let val: f64 = i;
        for (let j = 0; j < denoms.length; ++j) {
            unchecked(val = Math.log(val) * denoms[j]);
        }
        for (let j = bases.length - 1; j >= 0; --j) {
            unchecked(val = bases[j] ** val);
        }
        unchecked(result[i] = val);
    }
    return result;
}
```

7.1.3. AssemblyScript “Naïve” Approach

```
//Shorter code but worse performance than the others above
export function logNaturalsArrTestNoClosureNoSavingDenoms(maxCount:
f64, bases: f64[]):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (bases.length < 1) {
        return result;
    }
    for (let i = 0; i < maxCount; ++i) {
        let val: f64 = i;
        for (let j = 0; j < bases.length; ++j) {
            unchecked(val = Math.log(val) / Math.log(bases[j]));
        }
        for (let j = bases.length - 1; j >= 0; --j) {
            unchecked(val = bases[j] ** val);
        }
        unchecked(result[i] = val);
    }
    return result;
}
```

7.2. “Closures storing varying functions and incrementing variables”

See Figure 2 in section 4.1.2.

Note that the AssemblyScript code in this example uses `max()` and `min()` functions not prefixed with `Math.`; in AssemblyScript, these compile directly to the equivalent WebAssembly instructions. Since TypeScript does not have these functions, these are replaced with `Math.max()` and `Math.min()` in the TypeScript equivalent of this code, along with replacing the types and removing the `unchecked()` annotations.

7.2.1. AssemblyScript with Closure Extension

`__inferType` is here used to indicate types to be changed during the source-to-source compilation process, as described in section 4.1.3. The name used to annotate this is configurable in the arguments of our compiler.

```
export function varyingFunctionTest(operators: Int32Array,
operands: Float64Array, repeats:i32):f64 {
    function chooseOperator(operator:i32):(x:f64,y:f64)=>f64 {
        if (operator == 0) {
            return (x:f64, y:f64) => x + y;
        } else if (operator == 1) {
            return (x:f64, y:f64) => x - y;
        } else if (operator == 2) {
            return (x:f64, y:f64) => x * y;
        } else if (operator == 3) {
            return (x:f64, y:f64) => x / y;
        } else if (operator == 4) {
            return (x:f64, y:f64) => x ** y;
        } else if (operator == 5) {
            return (x:f64, y:f64) => Math.log(x) / Math.log(y);
        } else if (operator == 6) {
            return (x:f64, y:f64) => Math.atan2(y, x);
        } else if (operator == 7) {
            return (x:f64, y:f64) => Math.hypot(x, y);
        } else if (operator == 8) {
            return (x:f64, y:f64) => max(x, y);
        } else if (operator == 9) {
            return (x:f64, y:f64) => min(x, y);
        } else if (operator == 10) {
            return (x:f64, y:f64) => Math.random() * (y-x) + x;
        } else {
            return (x:f64, y:f64) => 0;
        }
    }
    function makeIncrementY(f:(x:f64,y:f64)=>f64,
y:f64):(x:f64)=>f64 {
        function incrY(x:f64):f64 {
            return f(x,y++);
        }
        return incrY;
    }
    const functionArray: StaticArray<(x:f64,y:f64)=>f64> = new
StaticArray<(x:f64,y:f64)=>f64>(operators.length);

    const closureArray: StaticArray<__inferType> = new
StaticArray<__inferType> (operators.length * operands.length);

    for (let i = 0; i < operators.length; ++i) {
        unchecked(functionArray[i] = chooseOperator(operators[i]));
    }
    for (let i = 0; i < operators.length; ++i) {
        for (let j = 0; j < operands.length; ++j) {
            unchecked(closureArray[i * operands.length + j] =
makeIncrementY(functionArray[i], operands[j]));
        }
    }
    let output:f64 = 0;
    for (let i = 0; i < repeats; ++i) {
        for (let j = 0; j < closureArray.length; ++j) {
            unchecked(output = closureArray[j](output));
        }
    }
    return output;
}
```

7.2.2. AssemblyScript without Extension

Shown here is the best-performing of the equivalents tested in ordinary AssemblyScript.

Additional versions can be seen as examples in the project source code [10].

```
export function varyingFunctionTestNoClosures(operators:
Int32Array, operands: Float64Array, repeats:i32):f64 {
    function chooseOperator(operator:i32):(x:f64,y:f64)=>f64 {
        if (operator == 0) {
            return (x:f64, y:f64) => x + y;
        } else if (operator == 1) {
            return (x:f64, y:f64) => x - y;
        } else if (operator == 2) {
            return (x:f64, y:f64) => x * y;
        } else if (operator == 3) {
            return (x:f64, y:f64) => x / y;
        } else if (operator == 4) {
            return (x:f64, y:f64) => x ** y;
        } else if (operator == 5) {
            return (x:f64, y:f64) => Math.log(x) / Math.log(y);
        } else if (operator == 6) {
            return (x:f64, y:f64) => Math.atan2(y, x);
        } else if (operator == 7) {
            return (x:f64, y:f64) => Math.hypot(x, y);
        } else if (operator == 8) {
            return (x:f64, y:f64) => max(x, y);
        } else if (operator == 9) {
            return (x:f64, y:f64) => min(x, y);
        } else if (operator == 10) {
            return (x:f64, y:f64) => Math.random() * (y-x) + x;
        } else {
            return (x:f64, y:f64) => 0;
        }
    }
    const functionArray: StaticArray<(x:f64,y:f64)=>f64> = new
StaticArray<(x:f64,y:f64)=>f64>(operators.length);

    const yArray: Float64Array = new Float64Array(operators.length
* operands.length);

    const functionArray2: StaticArray<(x:f64,y:f64)=>f64> = new
StaticArray<(x:f64,y:f64)=>f64>(operators.length * yArray.length);

    for (let i = 0; i < operators.length; ++i) {
        unchecked(functionArray[i] = chooseOperator(operators[i]));
    }
    for (let i = 0; i < operators.length; ++i) {
        for (let j = 0; j < operands.length; ++j) {
            const index:i32 = i * operands.length + j;
            unchecked(yArray[index] = operands[j]);
            unchecked(functionArray2[index] =
chooseOperator(operators[i]));
        }
    }
    let output:f64 = 0;
    for (let i = 0; i < repeats; ++i) {
        for (let j:i32 = 0; j < yArray.length; ++j) {
            unchecked(output = functionArray2[j](output,
yArray[j]++));
        }
    }
    return output;
}
```

7.3. Iterator Benchmarking

7.3.1. Custom Iterator Classes

These classes were used within multiple benchmarks, as shown in 7.3.2 and 7.3.3.

```
class Float64ArrayIterable {
    array: Float64Array;
    constructor(array: Float64Array) {
        this.array = array;
    }
    [Symbol.iterator](): Float64ArrayIterator {
        return new Float64ArrayIterator(this);
    }
}
class Float64ArrayIterator {
    source: Float64ArrayIterable
    constructor(source: Float64ArrayIterable) {
        this.source = source;
        this.index = 0;
    }
    index: i32;
    next(): IteratorResult<f64> {
        if (this.index >= this.source.array.length) {
            return unchecked(new IteratorResult<f64>(true,
this.source.array[0]));
        } else {
            return unchecked(new IteratorResult<f64>(false,
this.source.array[this.index++]));
        }
    }
    [Symbol.iterator](): Float64ArrayIterator {
        return this;
    }
}
class IteratorResult<T> {
    done: boolean;
    value: T;
    constructor(done:boolean, value:T) {
        this.done = done;
        this.value = value;
    }
}
```

7.3.2. “Iteration over array of bases for logarithms and exponents”

See Figure 3 in section 4.2.2.

7.3.2.1. AssemblyScript using Custom Iterator

```
export function logarithmTestWithIterators(maxCount: f64, bases:
Float64Array):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (bases.length < 1) {
        return result;
    }
    for (let i = 0; i < maxCount; ++i) {
        unchecked(result[i] = i);
    }
    for (let base of new Float64ArrayIterable(bases)) {
        const denom = Math.log(base);
        for (let i = 0; i < maxCount; ++i) {
            unchecked(result[i] = base ** (Math.log(result[i]) /
denom));
        }
    }
    return result;
}
```

7.3.2.1.1. Source-to-Source Compilation Output

```
export function logarithmTestWithIterators(maxCount: f64, bases:
Float64Array):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (bases.length < 1) {
        return result;
    }
    for (let i = 0; i < maxCount; ++i) {
        unchecked(result[i] = i);
    }
    for (let __iterator = new
Float64ArrayIterable(bases).Symbol_iterator(),
_result = __iterator.next(),
base  = _result.value; !_result.done;
_result = __iterator.next(), base  = _result.value)  {
        const denom = Math.log(base);
        for (let i = 0; i < maxCount; ++i) {
            unchecked(result[i] = base ** (Math.log(result[i]) /
denom));
        }
    }
    return result;
}
```

7.3.2.2 AssemblyScript without Iterators

```
export function logarithmTestWithoutIterators(maxCount: f64, bases:
Float64Array):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (bases.length < 1) {
        return result;
    }
    for (let i = 0; i < maxCount; ++i) {
        unchecked(result[i] = i);
    }
    for (let j = 0; j < bases.length; ++j) {
        const base = unchecked(bases[j]);
        const denom = Math.log(base);
        for (let i = 0; i < maxCount; ++i) {
            unchecked(result[i] = base ** (Math.log(result[i]) /
denom));
        }
    }
    return result;
}
```

7.3.3. “Iteration over array nested with incrementation loop, with varying loop nesting and number of iterations”

See Figure 4 in section 4.2.2.

```
export function noIteratorArrayOutside(maxCount: f64, factors:
Float64Array):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (factors.length < 1) {
        return result;
    }
    for (let i = 0; i < maxCount; ++i) {
        unchecked(result[i] = i);
    }
    for (let j = 0; j < factors.length; ++j) {
        const val = unchecked(factors[j]);
        for (let i = 0; i < maxCount; ++i) {
            unchecked(result[i] *= val);
        }
    }
    return result;
}

export function iteratorArrayOutside(maxCount: f64, factors:
Float64Array):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (factors.length < 1) {
        return result;
    }
    for (let i = 0; i < maxCount; ++i) {
        unchecked(result[i] = i);
    }
    const values = new Float64ArrayIterable(factors);
    for (let val of values) {
        for (let i = 0; i < maxCount; ++i) {
            unchecked(result[i] *= val);
        }
    }
    return result;
}

export function noIteratorArrayInside(maxCount: f64, factors:
Float64Array):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (factors.length < 1) {
        return result;
    }
    for (let i = 0; i < maxCount; ++i) {
        unchecked(result[i] = i);
    }
    for (let i = 0; i < maxCount; ++i) {
        for (let j = 0; j < factors.length; ++j) {
            const val = unchecked(factors[j]);
            unchecked(result[i] *= val);
        }
    }
    return result;
}

export function iteratorArrayInside(maxCount: f64, factors:
Float64Array):Float64Array {
    const result = new Float64Array(<i32>maxCount);
    if (factors.length < 1) {
        return result;
    }
    for (let i = 0; i < maxCount; ++i) {
        unchecked(result[i] = i);
    }
    const values = new Float64ArrayIterable(factors);
    for (let i = 0; i < maxCount; ++i) {
        for (let val of values) {
            unchecked(result[i] *= val);
        }
    }
    return result;
}
```