



COMPILATION OF A TYPED R-LIKE LANGUAGE TO WEBASSEMBLY

Baljinnyam Bilguudei

Bachelor's thesis
Faculty of Information Technology
Czech Technical University in Prague
Department of Computer Science
Study program: Informatics
Specialisation: Computer Science
Supervisor: Pierre Donat-Bouillud, Ph.D.
January 3, 2026

Replace the contents of this file with official assignment.
Místo tohoto souboru sem patří list se zadáním závěrečné práce.

Czech Technical University in Prague
Faculty of Information Technology

© 2026 Baljinnyam Bilguudei. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its use, with the exception of free statutory licenses and beyond the scope of the authorizations specified in the Declaration, requires the consent of the author.

Citation of this thesis: Baljinnyam Bilguudei. *Compilation of a typed R-like language to WebAssembly*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2026.

I would like to thank my supervisor, Pierre Donat-Bouillud, Ph.D for his invaluable support and patience with me throughout the work of this thesis. I also want to show my gratitude for amazing teachers at FIT, CTU, without whom, I wouldn't have learned much. I also want to thank all my friends I found in the university for making the journey unforgettable, and for my friends and my family for supporting me, especially my partner Barbora Navrátilová and, my brothers Chinguun Baljinnyam, Subeedei Purevmaa, and finally my mother Purevmaa Badii.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on January 3, 2026

Abstract

R is a widely-used dynamic language for statistical computing, but its dynamic nature prevents efficient ahead-of-time compilation. We present the design and implementation of a compiler for a statically-typed subset of R that targets WebAssembly. The defined language retains R’s core characteristics, including first-class vector operations and lexical scoping, while introducing static typing to enable compilation. The compiler leverages the WebAssembly Garbage Collection proposal, which simplifies memory management and enables efficient representation of high-level language constructs. Source programs are transformed through parsing, type checking, and intermediate representation lowering to generate WebAssembly bytecode, supported by a runtime system providing vector operations and host environment interoperability. Evaluation on representative statistical programs demonstrates the viability of compiling R-like languages to WebAssembly using modern proposals.

Keywords WebAssembly, WASM GC, typed R, WASM bytecode, pass-based Compiler

Abstrakt

R je široce používaný dynamický jazyk pro statistické výpočty, ale jeho dynamická povaha brání efektivní kompilaci předem. Představujeme návrh a implementaci kompilátoru pro staticky typovanou podmnožinu jazyka R, která cílí na WebAssembly. Definovaný jazyk si zachovává základní vlastnosti jazyka R, včetně prvotřídních vektorových operací a lexikálního rozsahu, a zároveň zavádí statické typování pro umožnění kompilace. Kompilátor využívá návrh WebAssembly Garbage Collection, který zjednodušuje správu paměti a umožňuje efektivní reprezentaci konstrukcí jazyků vysoké úrovně. Zdrojové programy jsou transformovány pomocí parsování, kontroly typů a mezilehlé reprezentace snižování, aby se generoval bajtkód WebAssembly, podporovaný běhovým systémem poskytujícím vektorové operace a interoperabilitu hostitelského prostředí. Vyhodnocení reprezentativních statistických programů demonstruje životaschopnost kompilace jazyků podobných jazyku R do WebAssembly s využitím moderních návrhů.

Klíčová slova WebAssembly, WASM GC, typovaný R, WASM bytecode, kompilátor založený na průchodech

Contents

1	Introduction	1
1.1	Aim	2
2	Background	3
2.1	The R Programming Language	3
2.1.1	Key Characteristics	3
2.1.2	Assignment Operators	4
2.1.3	Type System	4
2.1.4	Peculiarities	5
2.2	WebAssembly	6
2.2.1	WASM Garbage Collection proposal	9
2.3	Previous works	9
3	Language Design and Type System	10
3.1	Design Goals	10
3.1.1	What makes R hard to compile	10
3.2	Typed R-like Language	12
3.3	Syntax	13
3.3.1	Abstract Grammar	16
3.3.2	Lexical Elements	16
3.4	Type System	17
3.4.1	Subtyping and Type Promotion	18
3.4.2	Type Checking	20
3.4.3	Typing Rules	21
3.4.3.1	Typing Contexts	21
3.4.3.2	Subtyping	21
3.4.3.3	Literals and Variables	22
3.4.3.4	Arithmetic Operators	23
3.4.3.5	Comparison Operators	23
3.4.3.6	Logical Operators	23
3.4.3.7	Vector Operations	24
3.4.3.8	Functions	25
3.4.3.9	Control Flow	26
3.4.3.10	Statements	27
3.4.4	Scoping and Closures	28
3.5	Evaluation Model	28

3.5.1	Evaluation Order and Type Promotion	28
3.5.2	Vector Operations	29
3.5.3	Function Evaluation and Closures	29
3.5.4	Built-in Functions	29
4	Compiler and Runtime	30
4.1	Compiler Architecture	30
4.2	WebAssembly Code Generation	31
4.2.1	Type Mapping	31
4.2.2	Expression Compilation	33
4.2.3	Statement Compilation	35
4.2.4	Function Compilation	36
4.3	Runtime System	37
4.3.1	Memory Management	37
4.3.2	Vector Operations	38
4.3.3	Built-in Function Implementations	38
4.3.4	WASI Integration	39
4.4	Implementation Details	40
4.4.1	Compilation Limitations	40
4.4.2	Design Decisions	40
4.4.3	Performance Considerations	41
4.5	Runtime System	41
4.6	Implementation Details	41
5	Evaluation	42
5.1	Correctness	42
5.1.1	Type Safety	43
5.2	Performance	44
5.2.1	Compilation Time	44
5.2.2	Runtime Performance	45
5.3	Discussion	45
5.3.1	Results	46
5.3.2	Limitations	46
5.3.3	Future Work	46
5.3.4	Related Work	47
6	Conclusion	49
A	Extra	51
	Contents of the attachment	55

List of Figures

1.1	Overview. Taken from R Bytecode Book website	1
2.1	Tree representation	7
3.1	Subtyping hierarchy showing scalar type promotion (<code>logical <: int <: double</code>) and covariant vector subtyping (<code>vector<logical> <: vector<int> <: vector<double></code>). Scalar-to-vector conversions preserve the subtyping relationship, allowing implicit promotion in both scalar and vector contexts.	19
3.2	Bidirectional type checking for variable assignment. The type annotation <code>double</code> provides the expected type (top-down), while the literal <code>3.5</code> has its type inferred as <code>double</code> (bottom-up). The unification step verifies that both types match, allowing the assignment to proceed.	20
4.1	Compiler pipeline architecture showing front-end, middle-end, and back-end phases	31
5.1	End-to-end validation test architecture. Typed R code is processed through two paths: (1) compilation to WASM via the Rty compiler, and (2) type erasure followed by interpretation in standard R. Outputs are compared to ensure semantic equivalence.	43

List of Tables

3.1	Comparison of language features supported by R and Typed R	12
3.2	Comparison of language types supported by R and Typed R . .	13
3.3	Built-in functions and their types	29
5.1	Compilation time benchmarks	44
5.2	Runtime performance benchmarks	45

List of Code listings

2.1	Module demonstrating import and export mechanisms	6
2.2	A simple WebAssembly function showing stack operations . . .	7
2.3	Conditional execution using structured control flow	8
2.4	Accessing linear memory with load and store instructions . . .	8
3.1	Variable assignment examples	13
3.2	Super-assignment example	14
3.3	Function definition examples in typed R	14
3.4	Conditional examples in typed R	14
3.5	Loop examples	15
3.6	Block with tail expression	15
3.7	Vector type examples	18
3.8	Closure example	28
4.1	Vector struct type	32
4.2	Closure struct type	32
4.3	If-else compilation	35
4.4	While loop compilation	35
4.5	For loop compilation	36
4.6	Simple function	37
4.7	Closure function	37

List of abbreviations

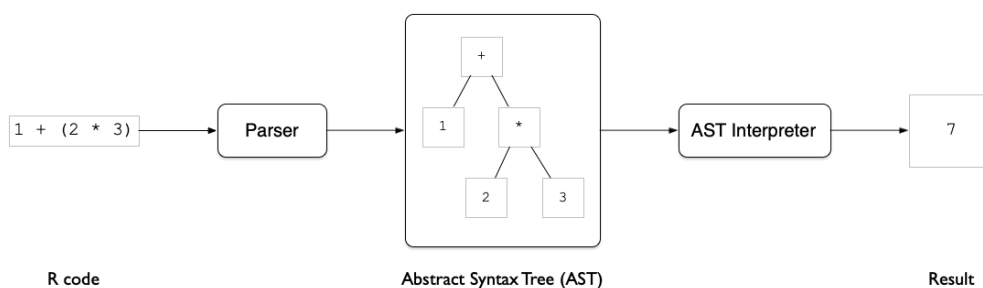
WASM	Web Assembly
WAT	Web Assembly Text
GC	Garbage Collector
IR	Intermediate Representation
NSE	Non Standard Evaluation (R)
AST	Abstract Syntax Tree
CRAN	The Comprehensive R Archive Network
OOP	Object Oriented Programming
SIMD	Single Instruction, Multiple Data
WASI	WebAssembly System Interface
FFI	Foreign function interface
I/O	Input, Output

Chapter 1

Introduction

The R programming language is ubiquitous for statistical computing and data analysis, offering powerful abstraction for data manipulation and visualization. Inspired by S-Language, the R language has been an important part for industries working with statistics and been a pioneer for introducing widespread tools currently used by the community such as DataFrame.

Traditionally, R is an interpreted language that runs on most major operating systems. By interpreted language, we mean that the language source code is not translated to certain binary to be executed by a processor like compiled languages. Instead, an interpreter is a program, usually written in low-level languages such as C, C++ or Rust, which takes the source code, does lexing and parsing to get AST and directly executes the program from its AST representation by evaluating from the top node[1]. Figure 1.1 shows an overview of R code being interpreted.



■ **Figure 1.1** Overview. Taken from R Bytecode Book website

Nowadays, the industry has been expanding the environments where we can run our programs. The interpreter program we discussed about, is compiled to bytecode, and then binary that the processor can execute. However, with invention of WebAssembly, we can take this interpreter and compile it to so-called WASM bytecode, and run it on web. This is exactly what webR[2] did. This way, R code we write, can seamlessly run on the web environment on

most browsers.

Compiling the interpreter to WASM introduces certain challenges. Since the interpreter consists of a substantial amount of code, every R program executed in the browser must include the entire interpreter compiled to WASM. This increases the overall memory footprint and can negatively impact performance, particularly in resource-constrained environments.

Thus, this thesis explores direct compilation of R-like language to WebAssembly. I create subset of R with type annotations, in order to make compilation possible to WASM, as it needs static typing. Then explore the challenges of mapping a dynamic high-level language to WASM, and finally evaluate the performance and correctness of my compiler.

The structure of the thesis will follow the same premise. I'll introduce the reader to the concepts of R, WASM and previous works. Then I'll state the rationale for creating typed R in Chapter 3, where I also introduce the language itself, both formally and informally with examples. We'll then go over the building of the compiler and runtime, and lastly will discuss the evaluation of the compiler and the performance of the generated binary in comparison to standard R environment and WebR.

1.1 Aim

It's important to note this focuses on exploring possibility and options of compiling R to WASM. Creation of type system and annotations are a vehicle to reach there and not full focus of this thesis, as there are better typing proposals[3][4][5]. Primarily the aim is to define a statically-typed subset of R, that is simple enough to be able to be compiled to WASM, but also expressive enough to show R's core such as mathematical operations and vectors. Then show by implement an efficient compiler for typed R to WASM and evaluating the generation of WASM.

Background

2.1 The R Programming Language

R is a domain-specific language designed for statistical computing and graphics, originally developed by Ross Ihaka and Robert Gentleman in the early 1990s as an open-source implementation of the **S** language. With its CRAN package manager providing more than 20000 packages, it's a widely programming language used by industries where experimenting with data is involved, such as data analytics, data mining and bio-informatics.

2.1.1 Key Characteristics

- **Dynamic typing:** Variables have no declared types; types are determined at runtime. This enables rapid prototyping but prevents static verification and certain compiler optimizations.
- **Vectorization:** Operations apply element-wise to vectors, matrices, and arrays. For example,

$$c(1,2,3) + c(4,5,6) \Rightarrow c(5,7,9)$$

without the need for explicit loops.

- **Lexical scoping:** R uses lexical (static) scoping with closures. Functions capture their defining environment, enabling functional programming patterns and higher-order abstractions.
- **Lazy evaluation:** Function arguments are evaluated lazily (call-by-need), allowing non-standard evaluation mechanisms that support domain-specific sublanguages.
- **Copy-on-modify semantics:** R employs implicit copying to preserve referential transparency. While this simplifies reasoning about programs, it may introduce performance overhead in memory-intensive workloads.

2.1.2 Assignment Operators

R provides multiple assignment operators with distinct scoping behavior:

- `<-`: Regular assignment in the current environment.
- `<<-`: Superaassignment, which modifies the nearest existing binding in an enclosing environment.

2.1.3 Type System

R employs a dynamic type system with several foundational types, all of which are first-class objects. Unlike statically-typed languages, type information is associated with values at runtime rather than with variable declarations. Atomic Types R provides six atomic vector types:

- **Logical:** Boolean values `TRUE`, `FALSE`, and `NA` (missing).

```
x <- TRUE
typeof(x)      # returns "logical"
```

- **Integer:** Whole numbers, denoted with an `L` suffix.

```
x <- 42L
typeof(x)      # returns "integer"
```

- **Double:** Floating-point numbers (default numeric type).

```
x <- 3.14
typeof(x)      # returns "double"
```

- **Character:** Strings of text.

```
x <- "hello"
typeof(x)      # returns "character"
```

- **Complex:** Complex numbers with real and imaginary parts.

```
x <- 2 + 3i
typeof(x)      # returns "complex"
```

- **Raw:** Raw bytes (rarely used).

```
x <- charToRaw("A")
typeof(x)      # returns "raw"
```


Composite Types Beyond atomic vectors, R supports several composite data structures:

- **List:** Heterogeneous collections that can hold elements of different types.

```
x <- list(42, "text", TRUE)
typeof(x)      # returns "list"
```

- **Function:** Functions are first-class objects.

```
f <- function(x) x + 1
typeof(f)      # returns "closure"
```

This means functions are treated as normal variables. It can be nested definition, passed as parameter and returned from a function.

- **Environment:** Hash-like structures for variable scoping.

```
e <- new.env()
typeof(e)      # returns "environment"
```

Type Coercion R performs implicit type coercion following a hierarchy: logical \rightarrow integer \rightarrow double \rightarrow character. When combining types, R automatically converts to the most general type:

```
c(TRUE, 1L, 2.5, "text") # returns character vector:
# "TRUE" "1" "2.5" "text"
```

This automatic coercion simplifies interactive use but can lead to unexpected behavior if types are not carefully managed.

2.1.4 Peculiarities

As every programming language comes with their nuances and uniqueness, R is no short of those. Below, the ones mention worthy examples

- Vectors and lists are indexed by 1. For example,

```
v <- c(10,20,30) // initializes a vector of 10,20,30
v[1]             // returns 10
v[2]             // returns 20
```

- Even a scalar is represented as a vector of 1 element.

```
x <- 5           // assign 5
is.vector(x)     // return TRUE
```

The reason for it is to recycle the vectors easily. The language is designed for vector operations

```
x <- 5           // assign 5
v <- c(10,20,30) // initializes a vector of 10,20,30
x + v           // returns a vector of 15,25,35
```

2.2 WebAssembly

WebAssembly is a low-level assembly-like language that is made to run in modern browser[6]. It's designed to work together with JavaScript, the language of the web environment, offering flexibility and performance. Currently most modern languages support WASM as compilation target; for example, Rust through rustc, C/C++ through Emscripten.

Practically, WebAssembly offers myriad of possibilities to developers. From Image-processing libraries in hidden behind C to desktop apps written in C#, with WASM, they can run on the web with near native-like performance. Notable examples of adoption of WASM are Figma[7], Autodesk AutoCAD Web App[8], and Google Earth[9].

Moreover, for readability, WebAssembly also has so-called Web Assembly Text Format, in short WAT. It's a form where bytecode itself is more readable, with more syntactic sugars. Let's iterate over properties of WASM and how it's structured and written in WAT format.

Module system

WebAssembly code is organized into modules. Modules declare imports (functions, memory, tables, globals from the host environment) and exports (making internal definitions available externally)[10]:

```
1 (module
2   ;; Import a logging function from the host
3   (import "env" "log" (func $log (param i32)))
4
5   ;; Import memory from the host
6   (import "env" "memory" (memory 1))
7
8   ;; Internal function (not exported)
9   (func $internal_add (param $a i32) (param $b i32) (
10     result i32)
11     local.get $a
12     local.get $b
13     i32.add
14   )
```

```

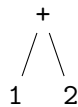
15  ;; Can write memory
16  ;; Can create data segment
17
18  ;; Public function that uses imports
19  (func $add_and_log (param $a i32) (param $b i32) (
20      result i32)
21      local.get $a
22      local.get $b
23      call $internal_add      ;; call internal function
24      local.get 0             ;; duplicate result for logging
25      call $log               ;; log the result
26  )
27
28  ;; Export the public function
29  (export "addAndLog" (func $add_and_log))

```

■ **Code listing 2.1** Module demonstrating import and export mechanisms

Stack-based machine

With goals of being fast, efficient, and portable, WebAssembly, at its heart, is a stack-based virtual machine running on the web (Possible to run on a machine through NodeJS or Wasmtime). Stack-based machine in this context is a process virtual machine that works as virtualization of computer system on top of a computer. Its workings is primarily based on interacting with the stack. For example:



■ **Figure 2.1** Tree representation

```

1  (module
2    (func $compute (result i32)
3      i32.const 1      ;; push constant 1 onto stack
4      i32.const 2      ;; push constant 2 onto stack
5      i32.add          ;; pop two values, push sum
6    )
7    (export "compute" (func $compute))
8  )

```

■ **Code listing 2.2** A simple WebAssembly function showing stack operations

Structured control flow

Unlike raw bytecode with goto-style jumps, WebAssembly uses structured control flow constructs: block, loop, if, and br (branch). Each construct creates a label that branches can target.[10]

```

1 (module
2   (func $max (param $a i32) (param $b i32) (result i32)
3     local.get $a
4     local.get $b
5     i32.gt_s           ;; signed greater-than
6                       comparison
7     (if (result i32)
8       (then
9         local.get $a
10      )
11      (else
12        local.get $b
13      )
14    )
15  )
16  (export "max" (func $max))
17 )

```

■ **Code listing 2.3** Conditional execution using structured control flow

Linear memory model

WebAssembly provides a contiguous, resizable array of bytes called linear memory. Memory is accessed via load/store instructions with explicit alignment and offset.[10]

```

1 (module
2   (memory 1)           ;; declare 1 page (64KB) of
3                       memory
4   (func $increment_at_zero
5     i32.const 0         ;; memory address
6     i32.const 0         ;; memory address (for load)
7     i32.load            ;; load 32-bit value from
8                       address 0
9     i32.const 1         ;; constant 1
10    i32.add              ;; increment
11    i32.store            ;; store result back to address
12                       0
13  )
14  (export "memory" (memory 0))
15  (export "increment" (func $increment_at_zero))
16 )

```

■ **Code listing 2.4** Accessing linear memory with load and store instructions

Type system and Determinism

A decoded module is type-safe and checked in module instantiation. Moreover, WebAssembly specification fully defines valid programs and their behaviour.[10]

2.2.1 WASM Garbage Collection proposal

WASM Garbage collection proposal, in short GC, is a new proposal that tries to provide efficient support for high-level languages. Since WASM announcement, community were coming up with their own GC as a way to compile high-level languages to WASM. With GC proposal, it helps to take load off of developers who are porting their languages to WASM. In this proposal, they introduced structs and arrays, which we'll heavily use in the implementation of this thesis.

2.3 Previous works

Work related to R about compilers and interpreter has been nothing but inspiring. The R has started with what we now call Tree-Walk interpreter[11]. Then around April 2011, with release of R 2.13.0 come the bytecode compiler by Luke Tierney, which was written almost entirely in R[12]. Essentially, it's faster to compile R into some intermediate bytecode and then have it executed by VM[13]. Thus, standard R environment execution became either with AST interpreter or with Bytecode Compiler[14]. From then on all standard functions and packages in base R were pre-compiled into bytecode. Lastly, JIT bytecode compiler became the default from R.3.4.1[15].

On top of standard implementations, execution on different environments have also been explored. WebR[2], most notably, is a recent work that compiles the R interpreter to WebAssembly using the Emscripten compiler to run R code on the web. Though as hinted at introduction, compiling the interpreter to WASM has its tradeoffs.

In terms of type system and type annotations around the R, there has been a discussion around type system in R as well[4]. There are even implementations for type system for R[3], although it's not adopted by R community. However, the purpose of the thesis is exploring the compilation of R to WASM. In that we use type annotations, but it's important to note that we won't dig deep into type systems at all.

Language Design and Type System

In this chapter, we'll design a subset of R with types and see how types and their operations could map to WASM generation.

3.1 Design Goals

Some reader at this point might be wondering why subset of R and why with types, why not just compile R to WASM. R is notoriously hard to compile to statically typed byte-code. A first reason is typing.

3.1.1 What makes R hard to compile

Dynamic typing

Let's take the following code.

```
1 x <- some_function()
```

WASM on the other hand is expecting variable `x` to have a type. There's `any` type in WASM. Maybe we can try mapping it to `anyref`. A reference to `any` element by the WASM GC proposal. So now we have

```
1 (local $x anyref) // A tagged union can be also used.
```

Then, let's have a binary operation `x + y`, where `y` is also `anyref`. Now we need to implement a generic function for plus operator that can potentially dispatch to all types of that function.

```
1 if x is int && y is int:  
2   add_int(x, y)
```

```

3  if x is double && y is double:
4      add_double(x, y)
5  ...

```

This code will continue for every type and its combination. We'd have to also include promotion logic inside this code. Then `x` is `int` needs `try_cast` as well. We'll need to do such generic function for every operation and functions. Some will be impossible to cover. This is essentially re-implementing the R interpreter. Thus we fixate on R that's well-typed and the type is known or inferable at compile time.

Moreover, we will see the same issue of dynamic dispatching when R variables change types.

```

1  x <- 5          # numeric
2  x <- "hello"    # now a string
3  x <- list()     # now a list

```

This bounds us to either writing generic functions or using wrappers like a tagged union, which will limit our performance and waste memory.

Changes in Runtime

Change in built-in functions.

```

1  '+' <- function(x, y) x * y # redefine addition!

```

This makes the job of compiler very hard.

Lazy Evaluation

R uses lazy evaluation for function arguments; they're only evaluated when actually used:

```

1  f <- function(x, y) { if (x > 0) y else 0 }
2  f(5, expensive_computation()) # expensive_computation
    () never runs!

```

This requires complex bookkeeping that's hard to optimize.

Reflection and Metaprogramming

R code can inspect and modify itself at runtime:

```

1  x <- quote(a + b) # capture unevaluated expression
2  eval(x)           # evaluate it later

```

Non-Standard Evaluation (NSE)

Many R functions evaluate arguments in non-standard ways:

```
1 subset(df, age > 30) # 'age' isn't a variable, it's a
   column name!
```

This is extremely difficult to analyze statically.

In conclusion, all those issues outlined above, is not impossible to compile. However, they require complex implementation and huge performance and memory trade-offs. For these, reasons it's important for me to define the subset of R, which I can compile and limit the scope without losing R's main functionalities and purposes.

3.2 Typed R-like Language

This section presents the design of a statically-typed programming language inspired by R's syntax, which we refer to as the *Typed R-like Language*. The language maintains R's characteristic features such as the left-assignment operator (`<-`), first-class functions, and vector-oriented programming, while introducing a static type system to enable ahead-of-time compilation to WebAssembly. Below is table 3.1 comparing R and typed R in terms of features and design.

Language Feature (Types)	R	Typed R
Integers	✓	✓
Doubles	✓	✓
Booleans	✓	✓
Strings	✓	×
NULL/NA values	✓	×
Vectors	✓	✓
Matrices	✓	×
Lists	✓	×
Data frames	✓	×
Arrays	✓	×

■ **Table 3.1** Comparison of language features supported by R and Typed R

The design philosophy emphasizes a familiar R-like syntax while ensuring type safety and efficient compilation to WebAssembly. Unlike dynamically-typed R, all type information is resolved at compile time, enabling optimized code generation and early error detection.

Language Feature	R	Typed R
Static typing	×	✓
First-class functions	✓	✓
Vector operations	✓	✓
Math operations	✓	✓
Control flow	✓	✓
Higher-order functions	✓	✓
Lexical scoping	✓	✓
Named arguments	✓	✓
Variable arguments (...)	✓	✓
Reflection	✓	×
Runtime type changes	✓	×
Non-standard evaluation (NSE)	✓	×
Lazy evaluation	✓	×
Copy-on-modify semantics	✓	×
Garbage collection	✓	✓
Operator overloading	✓	×
Attributes/metadata	✓	×
Method dispatch (S3/S4/R6)	✓	×

■ **Table 3.2** Comparison of language types supported by R and Typed R

3.3 Syntax

The syntax of the Typed R-like Language closely follows R conventions with extensions for explicit type annotations. This section describes the core syntactic constructs.

Examples of Typed R

Variable Declaration and Assignment Variables are declared using the left-assignment operator with optional type annotations:

```

1 # Simple assignment with type inference
2 x <- 10
3
4 # Assignment with explicit type annotation
5 y: int <- 20
6
7 # Vector assignment
8 vec <- c(1, 2, 3)

```

■ **Code listing 3.1** Variable assignment examples

Variable super-assignment The super-assignment operator `<<-` modifies variables in enclosing function scopes:

```
1 outer <- function() {  
2   x <- 0  
3   inner <- function() {  
4     x <- 10 # Modifies x in outer scope  
5   }  
6   inner()  
7   return(x) # Returns 10  
8 }
```

■ **Code listing 3.2** Super-assignment example

In this case we see that it's identical to its R version

Function Definitions Functions are first-class values defined using the `function` keyword:

```
1 # Simple function with type annotations  
2 add <- function(a: int, b: int): int {  
3   return(a + b)  
4 }  
5  
6 # Function returning a vector  
7 create_vector <- function(): vector<double> {  
8   return(c(1.0, 2.0, 3.0))  
9 }  
10  
11 # Higher-order function  
12 apply_twice <- function(f: int -> int, x: int): int {  
13   return(f(f(x)))  
14 }
```

■ **Code listing 3.3** Function definition examples in typed R

Conditional Statements The language supports `if-else` statements and expressions:

```
1 # If statement  
2 if (x > 0) {  
3   print(x)  
4 }  
5  
6 # If-else statement  
7 if (x > 0) {  
8   y <- 1  
9 } else {  
10   y <- -1  
11 }  
12  
13 # If expression (returns value)
```

```
14 result <- if (x > 0) { 1 } else { -1 }
```

■ **Code listing 3.4** Conditional examples in typed R

Loops Two loop constructs are provided: **for** and **while**.

```
1  # For loop iterating over range
2  for (i in 1:10) {
3      print(i)
4  }
5
6  # For loop iterating over vector
7  vec <- c(1, 2, 3, 4, 5)
8  for (x in vec) {
9      print(x)
10 }
11
12 # While loop
13 i <- 1
14 sum <- 0
15 while (i <= 5) {
16     sum <- sum + i
17     i <- i + 1
18 }
```

■ **Code listing 3.5** Loop examples

Blocks and Tail Expressions Blocks consist of zero or more statements followed by an optional tail expression. The tail expression (final expression without semicolon) determines the block's value:

```
1 f <- function(x: int): int {
2     y <- x * 2
3     z <- y + 1
4     z # Tail expression - returned automatically
5 }
```

■ **Code listing 3.6** Block with tail expression

3.3.1 Abstract Grammar

$$\begin{aligned}
e &::= x \quad (\text{variable}) \\
&| n \mid d \mid \mathbf{true} \mid \mathbf{false} \quad (\text{literals}) \\
&| \mathbf{function}(p_1, \dots, p_n) : \tau \{ e \} \quad (\text{function definition}) \\
&| e_1(e_2, \dots, e_n) \quad (\text{function call}) \\
&| e_1(x_1=e_2, \dots, x_n=e_n) \quad (\text{named argument call}) \\
&| e_1 \oplus e_2 \quad (\text{binary operation}) \\
&| \ominus e \quad (\text{unary operation}) \\
&| \mathbf{c}(e_1, \dots, e_n) \quad (\text{vector construction}) \\
&| e_1 : e_2 \quad (\text{range sequence}) \\
&| e_1[e_2] \quad (\text{vector indexing}) \\
&| \mathbf{if} \ e_1 \{ e_2 \} \mathbf{else} \{ e_3 \} \quad (\text{conditional}) \\
&| \{ e_1; \dots; e_n \} \quad (\text{block}) \\
s &::= x \leftarrow e \quad (\text{assignment}) \\
&| x : \tau \leftarrow e \quad (\text{typed assignment}) \\
&| x \llleftarrow e \quad (\text{superassignment}) \\
&| e \quad (\text{expression statement}) \\
&| \mathbf{return}(e) \quad (\text{return}) \\
&| \mathbf{for} \ (x \text{ in } e) \{ s \} \quad (\text{for loop}) \\
&| \mathbf{while} \ (e) \{ s \} \quad (\text{while loop}) \\
p &::= x : \tau \quad (\text{required parameter}) \\
&| x : \tau = e \quad (\text{parameter with default}) \\
&| \dots \quad (\text{varargs})
\end{aligned}$$

3.3.2 Lexical Elements

The language uses the following lexical tokens:

- **Keywords:** `function`, `if`, `else`, `for`, `in`, `while`, `return`
- **Type keywords:** `int`, `double`, `void`, `logical`, `vector`
- **Operators:**
 - Assignment: `<-` (assignment), `<<-` (super-assignment)
 - Arithmetic: `+`, `-`, `*`, `/`, `%%` (modulo)
 - Comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`
 - Logical: `&` (and), `|` (or), `!` (not)

- Range: `num1 : num2` (sequence generation)
- Type annotation: `:` (in declaration context)
- Function arrow: `->` (in type signatures)
- **Delimiters:** `(,), {, }, [,], ,`
- **Literals:** Numeric literals, logical literals (`TRUE, FALSE`)
- **Identifiers:** Alphanumeric sequences starting with a letter or underscore
- **Special:** `...` (varargs placeholder)

3.4 Type System

The type system ensures type safety through static analysis while supporting type inference to maintain concise syntax and ease of development. Though it's important to note that there are many programs where type inference will fail to identify the type and our compiler won't work due to lack of type information.

$\tau ::= \text{num} \mid \text{void}$	(base type)
$\mid \tau []$	(vector type)
$\mid (\tau_1, \dots, \tau_n) \rightarrow \tau$	(function type)
$\text{num} ::= \text{int} \mid \text{double} \mid \text{bool}$	(numeric types)

Primitive Types

The language provides the following primitive types:

- **int:** 32-bit signed integers (maps to WebAssembly `i32`)
- **double:** 64-bit floating-point numbers (maps to WebAssembly `f64`)
- **logical:** Boolean values (`TRUE` or `FALSE`)
- **void:** Absence of value (used for functions with no return)

Logical

Logicals will be represented as `int32` in WebAssembly, as it doesn't provide explicit type for logicals there.

Composite Types

Vector Types Vectors are homogeneous arrays parameterized by element type:

```
1 # Vector of integers
2 v1: vector<int> <- c(1, 2, 3)
3
4 # Vector of doubles
5 v2: vector<double> <- c(1.5, 2.5, 3.5)
6
7 # Vector operations (component-wise)
8 v3: vector<double> <- v2 + c(1.0, 2.0, 3.0)
```

■ Code listing 3.7 Vector type examples

Vectors support component-wise arithmetic operations when both operands have compatible vector types.

Function Types

Function types represent callable values with parameter and return types:

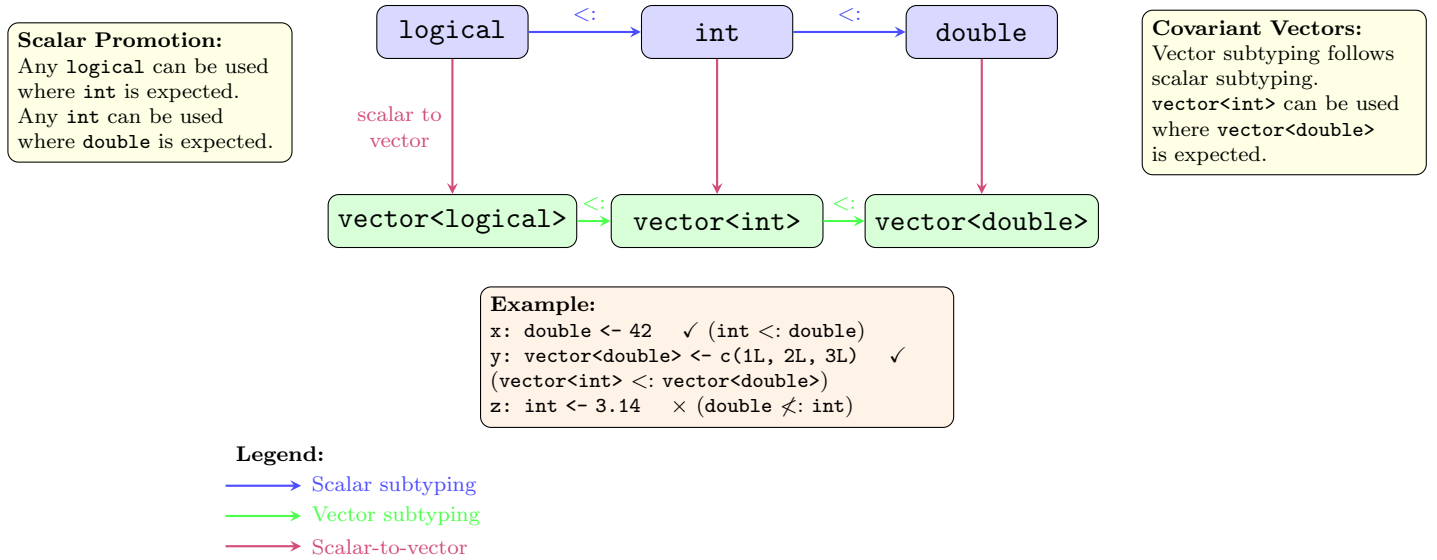
- `int -> int`: Function taking an integer, returning an integer
- `int, int -> double`: Function taking two integers, returning a double
- `(int -> int) -> int`: Higher-order function taking a function as parameter
- `int, (int, int -> int) -> int`: Function taking an integer and a function

3.4.1 Subtyping and Type Promotion

Subtyping relation between logical, int, double and vectors are shown below 3.1. For operations types are always implicitly casted to the wider types. From `logical <: int <: double`. There's also vectors which are covariant to these types. For example `vector<int>` can be used in place of `vector<double>`.

It's important to note, the language doesn't allow there's no user creatable subtyping. Currently, we don't support S3 system, or any way to do OOP in Typed R.

Subtyping Hierarchy: $\tau_1 <: \tau_2$ means τ_1 is a subtype of τ_2



■ **Figure 3.1** Subtyping hierarchy showing scalar type promotion (`logical <: int <: double`) and covariant vector subtyping (`vector<logical> <: vector<int> <: vector<double>`). Scalar-to-vector conversions preserve the subtyping relationship, allowing implicit promotion in both scalar and vector contexts.

3.4.2 Type Checking

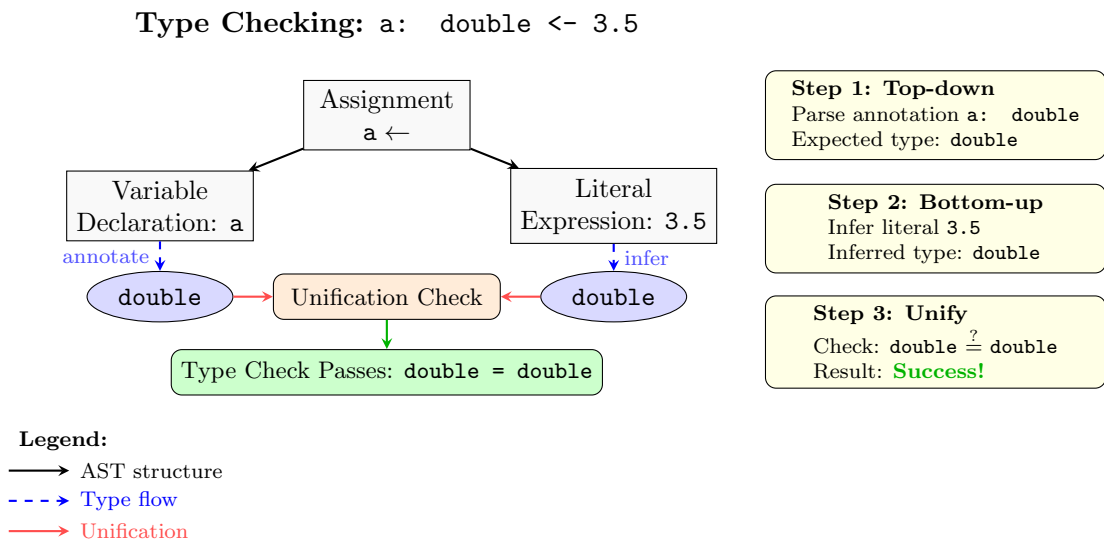
The type checker performs bidirectional type inference[16]:

- **Bottom-up inference:** Expression types are inferred from literal values and operator signatures
- **Top-down checking:** Function return types and variable annotations provide expected types
- **Unification:** Type constraints are solved to determine concrete types

Type annotations are required in the following contexts:

- Function parameters
- Function return types (when not inferable from return statements)
- Ambiguous variable declarations

Example: $3 + 2.5$ evaluates as $3.0 + 2.5 = 5.5$ after promoting 3 to double.



■ **Figure 3.2** Bidirectional type checking for variable assignment. The type annotation `double` provides the expected type (top-down), while the literal `3.5` has its type inferred as `double` (bottom-up). The unification step verifies that both types match, allowing the assignment to proceed.

3.4.3 Typing Rules

This section formalizes the type system using inference rules. We use **typing judgments** of the form:

$$\Gamma \vdash e : \tau$$

This is read as: “under typing context Γ , expression e has type τ .” The typing context Γ maps variable names to their types. The rules are presented in natural deduction style, where premises appear above the horizontal line and the conclusion below. A rule with no premises (nothing above the line) is an **axiom**—it holds unconditionally.

Our type system follows standard approaches from programming language theory [17], extended with subtyping for numeric promotion and vector operations.

3.4.3.1 Typing Contexts

A **typing context** (or type environment) is a finite mapping from variable names to types:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

The empty context \emptyset contains no bindings. The notation $\Gamma, x : \tau$ extends context Γ with a new binding of variable x to type τ . We write $x : \tau \in \Gamma$ to indicate that Γ contains the binding $x : \tau$.

3.4.3.2 Subtyping

Subtyping defines when one type can be safely used where another is expected. We write $\tau_1 <: \tau_2$ to mean “ τ_1 is a subtype of τ_2 ” (equivalently, τ_1 can be used wherever τ_2 is expected).

Reflexivity and Base Cases

Every type is a subtype of itself, and we define the numeric promotion hierarchy:

$$\frac{}{\tau <: \tau} \text{ S-REFL}$$

$$\frac{}{\text{logical} <: \text{int}} \text{ S-LOGICALINT}$$

$$\frac{}{\text{int} <: \text{double}} \text{ S-INTDOUBLE}$$

Explanation: S-REFL states that any type is a subtype of itself (reflexivity). S-LOGICALINT allows logical values (represented as 0 or 1) to be used as integers. S-INTDOUBLE allows integers to be promoted to doubles, enabling mixed arithmetic like $3 + 2.5$.

Transitivity

Subtyping is transitive, allowing chains like `logical <: int <: double`:

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \text{S-TRANS}$$

Explanation: If τ_1 can be used as τ_2 , and τ_2 can be used as τ_3 , then τ_1 can be used as τ_3 . This gives us `logical <: double` automatically.

Vector Covariance

Vectors are **covariant** in their element type:

$$\frac{\tau_1 <: \tau_2}{\text{vector}\langle\tau_1\rangle <: \text{vector}\langle\tau_2\rangle} \text{S-VECTOR}$$

Explanation: If element type τ_1 is a subtype of τ_2 , then a vector of τ_1 can be used where a vector of τ_2 is expected. For example, `vector<int> <: vector<double>`, so an integer vector can be passed to a function expecting a double vector.

Subsumption

The **subsumption rule** connects subtyping to the type system, allowing implicit type promotion:

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2} \text{T-SUB}$$

Explanation: If expression e has type τ_1 and τ_1 is a subtype of τ_2 , then e can also be given type τ_2 . This rule enables all implicit conversions in the language. For example, if x has type `int`, we can use it where `double` is expected.

3.4.3.3 Literals and Variables

These rules assign types to the basic building blocks of expressions.

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \text{int}} \text{T-INT}$$

$$\frac{d \in \mathbb{R}}{\Gamma \vdash d : \text{double}} \text{T-DOUBLE}$$

$$\frac{b \in \{\text{TRUE}, \text{FALSE}\}}{\Gamma \vdash b : \text{logical}} \text{T-BOOL}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{T-VAR}$$

Explanation:

- T-INT: Integer literals like 42 or -7 have type `int`.
- T-DOUBLE: Floating-point literals like 3.14 have type `double`.
- T-BOOL: The literals `TRUE` and `FALSE` have type `logical`.
- T-VAR: A variable x has whatever type the context Γ says it has. This rule performs a lookup in the typing environment.

3.4.3.4 Arithmetic Operators

Arithmetic operators (+, -, *, /, %) follow the same typing pattern. We show rules for addition; other operators are analogous.

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ T-ADDINT}$$

$$\frac{\Gamma \vdash e_1 : \text{double} \quad \Gamma \vdash e_2 : \text{double}}{\Gamma \vdash e_1 + e_2 : \text{double}} \text{ T-ADDDOUBLE}$$

Explanation: When both operands have the same numeric type, the result has that type. Note that mixed-type arithmetic like `3 + 2.5` is handled by first applying subsumption (T-SUB) to promote 3 from `int` to `double`, then applying T-ADDDOUBLE.

3.4.3.5 Comparison Operators

Comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) compare values and produce logical results:

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\text{int}, \text{double}, \text{logical}\}}{\Gamma \vdash e_1 \odot e_2 : \text{logical}} \text{ T-COMPARE}$$

where $\odot \in \{==, !=, <, <=, >, >=\}$.

Explanation: Comparisons require both operands to have the same type (after potential promotion via subsumption) and always produce a `logical` result. For example, `3 < 5` produces `TRUE`.

3.4.3.6 Logical Operators

Logical operators work on boolean values:

$$\frac{\Gamma \vdash e_1 : \text{logical} \quad \Gamma \vdash e_2 : \text{logical}}{\Gamma \vdash e_1 \ \& \ e_2 : \text{logical}} \text{ T-AND}$$

$$\frac{\Gamma \vdash e_1 : \text{logical} \quad \Gamma \vdash e_2 : \text{logical}}{\Gamma \vdash e_1 \ | \ e_2 : \text{logical}} \text{ T-OR}$$

$$\frac{\Gamma \vdash e : \text{logical}}{\Gamma \vdash !e : \text{logical}} \text{ T-NOT}$$

Explanation: Logical AND (&), OR (|), and NOT (!) require logical operands and produce logical results.

3.4.3.7 Vector Operations

Vector Construction

The `c()` function creates vectors from elements:

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \cdots \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash \text{c}(e_1, e_2, \dots, e_n) : \text{vector}\langle\tau\rangle} \text{ T-VECTOR}$$

Explanation: All elements must have the same type τ (after promotion). The result is a vector parameterized by that element type. For example, `c(1, 2, 3)` has type `vector<int>`.

Range Construction

The colon operator creates integer sequences:

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 : e_2 : \text{vector}\langle\text{int}\rangle} \text{ T-RANGE}$$

Explanation: The range `1:5` creates the vector `c(1, 2, 3, 4, 5)`. Both bounds must be integers.

Vector Indexing

Elements are accessed using bracket notation:

$$\frac{\Gamma \vdash e_1 : \text{vector}\langle\tau\rangle \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : \tau} \text{ T-INDEX}$$

Explanation: Indexing a vector of type `vector< τ >` with an integer index produces a value of type τ . Note that our language uses 1-based indexing, following R convention.

Vector Arithmetic

Binary operators extend component-wise to vectors:

$$\frac{\Gamma \vdash e_1 : \text{vector}\langle\tau\rangle \quad \Gamma \vdash e_2 : \text{vector}\langle\tau\rangle \quad \tau \in \{\text{int}, \text{double}\}}{\Gamma \vdash e_1 \oplus e_2 : \text{vector}\langle\tau\rangle} \text{ T-VECBINOP}$$

where $\oplus \in \{+, -, *, /, \%\}$.

Explanation: When both operands are vectors of the same numeric type, the operation applies element-wise. For example, `c(1,2) + c(3,4)` produces `c(4,6)`.

Scalar-Vector Operations

Scalars broadcast to match vector length:

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \text{vector}\langle\tau\rangle \quad \tau \in \{\text{int}, \text{double}\}}{\Gamma \vdash e_1 \oplus e_2 : \text{vector}\langle\tau\rangle} \text{ T-SCALARVEC}$$

Explanation: A scalar is implicitly replicated to match the vector length. For example, `2 * c(1,2,3)` produces `c(2,4,6)`. The symmetric case (vector \oplus scalar) follows analogously.

3.4.3.8 Functions

Function Definition

Functions are first-class values with explicit parameter and return types:

$$\frac{\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \text{function}(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau \{ e \} : (\tau_1, \dots, \tau_n) \rightarrow \tau} \text{ T-FUNC}$$

Explanation: To type-check a function, we extend the context with the parameter bindings and check that the body e has the declared return type τ . The function itself has a **function type** $(\tau_1, \dots, \tau_n) \rightarrow \tau$ indicating it takes n arguments of types τ_1, \dots, τ_n and returns type τ .

Function Application

Function calls check argument types against parameter types:

$$\frac{\Gamma \vdash e_0 : (\tau_1, \dots, \tau_n) \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_0(e_1, \dots, e_n) : \tau} \text{T-APP}$$

Explanation: To call a function, the callee e_0 must have a function type, and each argument must match the corresponding parameter type (or be promotable via subsumption). The result has the function's return type.

3.4.3.9 Control Flow

Conditional Expressions

If-else expressions require matching branch types:

$$\frac{\Gamma \vdash e_1 : \text{logical} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } (e_1) \{e_2\} \text{ else } \{e_3\} : \tau} \text{T-IF}$$

Explanation: The condition must be logical. Both branches must have the same type τ , which becomes the type of the entire expression. This enables using if-else as an expression: `x <- if (cond) { 1 } else { 2 }`.

While Loops

While loops do not produce meaningful values:

$$\frac{\Gamma \vdash e_1 : \text{logical} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{while } (e_1) \{e_2\} : \text{void}} \text{T-WHILE}$$

Explanation: The condition must be logical. The body is checked but its type is discarded; the loop itself has type `void`.

For Loops

For loops iterate over vectors:

$$\frac{\Gamma \vdash e_1 : \text{vector} \langle \tau_1 \rangle \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{for } (x \text{ in } e_1) \{e_2\} : \text{void}} \text{T-FOR}$$

Explanation: The iterable must be a vector. The loop variable x is bound to the element type within the body. Like while loops, for loops have type `void`.

3.4.3.10 Statements

Assignment

Assignment binds a value to a variable, extending the context:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (x \leftarrow e) : \tau \dashv \Gamma, x : \tau} \text{ T-ASSIGN}$$

Explanation: We use $\dashv \Gamma'$ to indicate the **output context** after the statement. Assignment evaluates e , binds the result to x , and produces an updated context where x has type τ . The assignment itself also has type τ (assignments are expressions in R).

Super-Assignment

Super-assignment modifies a variable in an enclosing scope:

$$\frac{x : \tau \in \Gamma_{outer} \quad \Gamma \vdash e : \tau'}{\Gamma \vdash (x \llleftarrow e) : \tau'} \text{ T-SUPERASSIGN}$$

where $\tau' <: \tau$ (the assigned value must be compatible with the existing binding).

Explanation: Super-assignment requires that x already exists in some enclosing function scope (Γ_{outer}). The expression type must be compatible with the existing variable's type.

Blocks

A block is a sequence of statements with an optional tail expression:

$$\frac{\Gamma \vdash s_1 : \tau_1 \dashv \Gamma_1 \quad \Gamma_1 \vdash s_2 : \tau_2 \dashv \Gamma_2 \quad \cdots \quad \Gamma_{n-1} \vdash e : \tau}{\Gamma \vdash \{s_1; s_2; \dots; e\} : \tau} \text{ T-BLOCK}$$

Explanation: Statements are checked in sequence, with each statement potentially extending the context for subsequent statements. The block's type is determined by its final expression (the tail expression).

Return Statement

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{return}(e) : \tau} \text{ T-RETURN}$$

Explanation: A return statement has the type of its argument. The type checker verifies that this matches the declared return type of the enclosing function.

3.4.4 Scoping and Closures

The language uses lexical scoping where:

- Only functions create new scopes (blocks, if-statements, and loops do not)
- Functions can capture variables from enclosing function scopes
- Captured variables are implemented using closure environments in WebAssembly
- Super-assignment (`<<-`) searches enclosing function scopes to modify variables

Example demonstrating closure:

```

1 make_counter <- function(start: int): () -> int {
2   count <- start
3   function(): int {
4     count <<- count + 1
5     return(count)
6   }
7 }
8
9 counter <- make_counter(0)
10 print(counter()) # Prints 1
11 print(counter()) # Prints 2

```

■ **Code listing 3.8** Closure example

3.5 Evaluation Model

This section describes the key runtime behaviors that affect how programs execute and how the compiler generates code.

3.5.1 Evaluation Order and Type Promotion

Expressions are evaluated left-to-right. When operands have different numeric types, the narrower type is promoted to the wider type before the operation:

- `logical` \rightarrow `int`: `TRUE` becomes 1, `FALSE` becomes 0
- `int` \rightarrow `double`: integers are converted to floating-point

For example, `TRUE + 2.5` evaluates as `1.0 + 2.5 = 3.5` (`logical` \rightarrow `int` \rightarrow `double`).

3.5.2 Vector Operations

Vector operations apply element-wise. When vectors have different element types, promotion occurs element-by-element:

$$\mathbf{c}(v_1, \dots, v_n) \oplus \mathbf{c}(w_1, \dots, w_n) = \mathbf{c}(v_1 \oplus w_1, \dots, v_n \oplus w_n)$$

Scalar Broadcasting

When a scalar and vector are combined, the scalar is implicitly broadcast (replicated) to match the vector length:

$$s \oplus \mathbf{c}(v_1, \dots, v_n) = \mathbf{c}(s \oplus v_1, \dots, s \oplus v_n)$$

For example, $2 * \mathbf{c}(1, 2, 3)$ produces $\mathbf{c}(2, 4, 6)$.

3.5.3 Function Evaluation and Closures

Functions in the typed R-like language are **closures**: they capture the environment in which they are defined. When a function is called:

1. The callee expression is evaluated to obtain a closure (function code + captured environment)
2. Arguments are evaluated left-to-right in the caller's environment
3. A new environment is created, extending the closure's captured environment with parameter bindings
4. The function body is evaluated in this new environment

This closure semantics enables patterns like factory functions and stateful callbacks, as shown in the counter example in Section 3.4.4.

3.5.4 Built-in Functions

The language provides the following built-in functions:

Function	Type	Description
$\mathbf{c}(\mathbf{e}_1, \dots, \mathbf{e}_n)$	$(\tau, \dots, \tau) \rightarrow \mathbf{vector}\langle\tau\rangle$	Creates vector from elements
$\mathbf{print}(\mathbf{e})$	$\tau \rightarrow \mathbf{void}$	Outputs value to stdout
$\mathbf{length}(\mathbf{v})$	$\mathbf{vector}\langle\tau\rangle \rightarrow \mathbf{int}$	Returns vector length
$\mathbf{sum}(\mathbf{v})$	$\mathbf{vector}\langle\tau\rangle \rightarrow \tau$	Sums numeric vector

■ **Table 3.3** Built-in functions and their types

Compiler and Runtime

All implementation related to compiling and its scripts are written in Rust language. Rust is chosen for its modern features, memory safety and performance. In coming sections, we'll describe the architecture, and design decisions about the implementation of the compiler.

4.1 Compiler Architecture

Most compilers follow pass-based implementation. To make sense of the complexity, we divide compiler into so-called front-end, middle-end, and back-end. Front-end includes everything related to actually taking the code as a string and creating AST tree, and then intermediate representation. Middle-end is used for mostly language-independent optimizations on the IR. Then, finally the backend, back-end takes that IR and generates target-specific binaries or bytecode. In our case back-end would be generating the WASM code.

Conceptually, each part is composed of multiple modules shown in figure below 4.1 .

Front-end

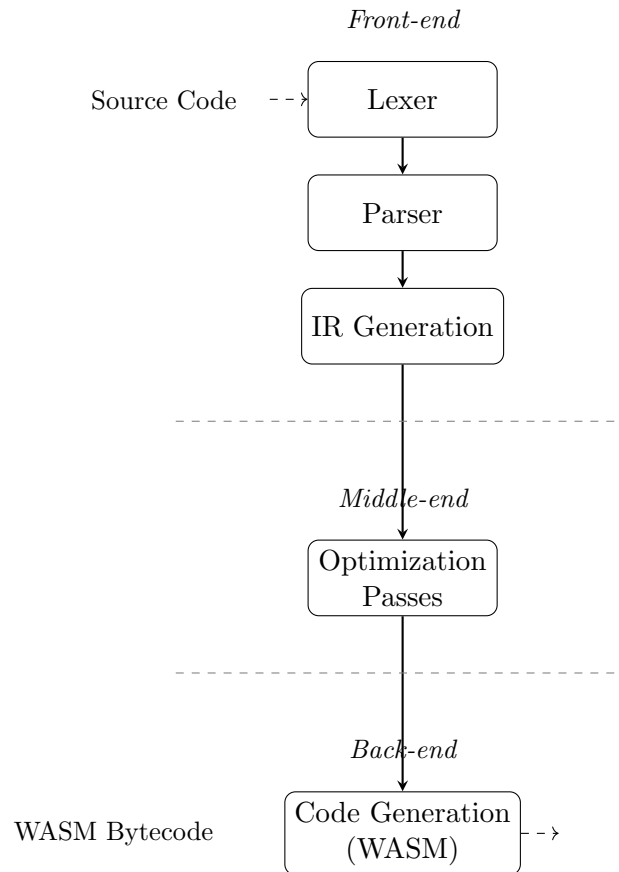
Composed of Lexer and Parser, it handles syntactic analysis and semantic analysis. Lexer takes in a source code as string, then produces so-called Tokens that programmer has defined. Those tokens are later fed into Parser, which uses the grammar to create AST tree.

Middle-end

This is where the optimizations happen. Or it could simply translate the IR into more efficient forms for back-end.

Back-end

Part that's concerned with only generating code for target architecture.



■ **Figure 4.1** Compiler pipeline architecture showing front-end, middle-end, and back-end phases

4.2 WebAssembly Code Generation

This section describes the translation of language constructs to WebAssembly instructions.

4.2.1 Type Mapping

Types are mapped to WebAssembly value types as follows:

Source Type	WebAssembly Type
int	i32
double	f64
logical	i32 (0 = false, 1 = true)
vector<T>	(ref \$vec_T) (struct with data array and length)
function	(ref \$functype) (typed function reference)

Vector Representation

Vectors are represented as WebAssembly GC structs:

```

1 (type $vec_f64 (struct
2   (field $data (ref (array (mut f64))))
3   (field $length (mut i32))
4 ))

```

■ Code listing 4.1 Vector struct type

This representation:

- Uses WebAssembly GC arrays for efficient storage
- Stores length separately for fast access
- Allows mutable arrays for in-place updates
- Type-specialized structs for different element types (i32, f64, anyref)

Function References

Functions are represented using WebAssembly typed function references:

- Simple functions: Direct (ref \$functype) where \$functype is the function signature
- Closures: Struct containing function reference and captured environment
- Function calls use `call_ref` for indirect calls through function references

Closure representation:

```

1 (type $closure (struct
2   (field $func (ref $functype))
3   (field $env (ref $env_struct))
4 ))

```

■ Code listing 4.2 Closure struct type

4.2.2 Expression Compilation

Literals

Literals compile to immediate value instructions:

- Integer literals: `i32.const n`
- Floating-point literals: `f64.const x`
- Boolean literals: `i32.const 0` (false) or `i32.const 1` (true)

Variables

Variable references compile to local or global access:

- Local variables: `local.get $var_idx`
- Captured variables: Load from closure environment struct
- Function references: `ref.func $func_idx`

Binary Operations

Binary operations compile to corresponding WebAssembly instructions:

- Integer arithmetic: `i32.add`, `i32.sub`, `i32.mul`, `i32.div_s`, `i32.rem_s`
- Float arithmetic: `f64.add`, `f64.sub`, `f64.mul`, `f64.div`
- Integer comparison: `i32.eq`, `i32.ne`, `i32.lt_s`, `i32.le_s`, `i32.gt_s`, `i32.ge_s`
- Float comparison: `f64.eq`, `f64.ne`, `f64.lt`, `f64.le`, `f64.gt`, `f64.ge`
- Logical operations: `i32.and`, `i32.or` (with boolean normalization)

Vector operations compile to loops that apply scalar operations element-wise:

1. Allocate result vector with same length as operands
2. Loop over indices 0 to length-1
3. Extract elements from both operand vectors
4. Apply scalar operation
5. Store result in result vector

Moreover, all vector operations are written in Typed R language and it's compiled and embedded into runtime. For example:

```

1  system_vector_add___vec_int__vec_int <- function(a:
    vector<int>, b: vector<int>): vector<int> {
2      n <- length(a)
3      m <- length(b)
4
5      if(n != m & n %% m != 0 & m %% n != 0) {
6          stop("Vector lengths not compatible for recycling
              ")
7      }
8
9      result_len <- max(n, m)
10     result: vector<int> <- vec(length=result_len, mode="
        int")
11
12     for(i in 1:result_len) {
13         a_idx <- ((i - 1) %% n) + 1
14         b_idx <- ((i - 1) %% m) + 1
15         result[i] <- a[a_idx] + b[b_idx]
16     }
17     return(result)
18 }

```

Such function is compiled to WASM and the operation `vec + vec` is mapped to this function name.

Function Calls

Function calls compile differently based on callee type:

- **Direct calls** (known function): `call $func_idx`
- **Indirect calls** (function variable):
 1. Load function reference from local/environment
 2. Evaluate arguments
 3. `call_ref $functype_idx`
- **Closure calls:**
 1. Load closure struct
 2. Extract environment field
 3. Extract function field
 4. Pass environment as first parameter
 5. Pass regular arguments
 6. `call_ref $closure_functype_idx`

4.2.3 Statement Compilation

Assignment

Regular assignment (`<-`):

1. Evaluate right-hand side expression
2. Store result in local variable: `local.set $var_idx`

Super-assignment (`<<-`) for captured variables:

1. Evaluate right-hand side expression
2. Load closure environment
3. Navigate to appropriate nesting level
4. Extract reference cell for variable
5. Update cell contents using `struct.set`

Control Flow

If statements compile to WebAssembly block structures:

```

1 ; Evaluate condition
2 (condition code)
3
4 ; If-else structure
5 (if (result ...)
6   (then
7     ; then-branch code
8   )
9   (else
10    ; else-branch code
11  )
12 )
```

Code listing 4.3 If-else compilation

While loops compile to loop blocks with conditional branching:

```

1 (block $loop_exit
2   (loop $loop_start
3     ; Evaluate condition
4     (condition code)
5
6     ; Exit if false
7     (i32.eqz)
8     (br_if $loop_exit)
9   )
10 )
```

```

10     ; Loop body
11     (body code)
12
13     ; Continue loop
14     (br $loop_start)
15 )
16 )

```

■ **Code listing 4.4** While loop compilation

For loops over ranges compile to indexed loops:

```

1  ; Initialize loop variable to start
2  (local.set $iter (start value))
3
4  (block $loop_exit
5    (loop $loop_start
6      ; Check condition: iter <= end
7      (local.get $iter)
8      (end value)
9      (i32.gt_s)
10     (br_if $loop_exit)
11
12     ; Loop body with iter
13     (body code)
14
15     ; Increment iter
16     (local.get $iter)
17     (i32.const 1)
18     (i32.add)
19     (local.set $iter)
20
21     (br $loop_start)
22   )
23 )

```

■ **Code listing 4.5** For loop compilation

For loops over vectors use similar structure but load vector elements by index.

4.2.4 Function Compilation

Function compilation involves multiple steps:

1. **Type registration:** Register function type in type section
2. **Function index allocation:** Assign unique function index
3. **Local variable allocation:** Determine local variable slots from `FunctionMetadata`

4. **Closure analysis:** Identify captured variables
5. **Environment struct generation:** Create struct type for captured variables (if needed)
6. **Code generation:** Emit function body

Simple function (no captures):

```

1 (func $add (param $a i32) (param $b i32) (result i32)
2   (local.get $a)
3   (local.get $b)
4   (i32.add)
5 )

```

■ **Code listing 4.6** Simple function

Closure function (with captures):

```

1 (type $env (struct (field $captured_var (mut i32))))
2
3 (func $closure_fn (param $env (ref $env)) (param $x i32)
4   (result i32)
5   ; Access captured variable
6   (local.get $env)
7   (struct.get $env $captured_var)
8
9   ; Use parameter
10  (local.get $x)
11
12  ; Computation
13  (i32.add)
14 )

```

■ **Code listing 4.7** Closure function

When returning a closure, the compiler:

1. Allocates environment struct
2. Copies captured variable values into struct fields
3. Allocates closure struct
4. Stores function reference and environment
5. Returns closure struct reference

4.3 Runtime System

4.3.1 Memory Management

The runtime uses WebAssembly GC for automatic memory management:

- Vectors, strings, and closures are GC-managed heap objects
- No explicit deallocation required
- WebAssembly GC handles reference counting and garbage collection
- Linear memory used for WASI I/O buffers and string serialization

Memory layout:

- Address 0: Reserved for null pointer checks
- Address 8+: WASI I/O buffers for `fd_write`
- Dynamic region: Managed by compiler for temporary string buffers

4.3.2 Vector Operations

Built-in vector operations are compiled to runtime function calls:

Vector Construction (`c(...)`):

1. Determine element type from arguments
2. Allocate array of appropriate size
3. Initialize array elements
4. Allocate vector struct
5. Store array reference and length
6. Return vector struct reference

Component-wise Operations: Vector arithmetic is implemented as in-line loops (described previously) rather than runtime calls for performance.

Reduction Operations (`sum, length`):

- `length`: Extract and return length field from vector struct
- `sum`: Loop over vector elements, accumulating sum

4.3.3 Built-in Function Implementations

Print Function

The `print` function serializes values to strings and outputs via WASI:

1. Convert value to string representation
2. Write string to linear memory buffer

3. Call `fd_write` to output to stdout (file descriptor 1)

For different types:

- Integers: Convert to decimal string
- Floats: Convert to decimal string with precision
- Strings: Output directly
- Vectors: Format as `[elem1, elem2, ...]`
- Booleans: Output `TRUE` or `FALSE`

Sequence Generation

The range operator `start:end` and `gen_seq` runtime function:

1. Calculate sequence length: `end - start + 1`
2. Allocate vector of integers
3. Fill array with values from `start` to `end`
4. Return vector struct

Optimized implementation uses a simple loop without intermediate allocations.

4.3.4 WASI Integration

The runtime integrates with WASI (WebAssembly System Interface) for I/O:

- **Import:** `fd_write` from `wasi_snapshot_preview1`
- **Signature:** `(i32, i32, i32, i32) -> i32`
- **Parameters:** file descriptor, iovs pointer, iovs length, `nwritten` pointer
- **Usage:** Output strings to stdout/stderr

The `_start` function is exported as the entry point, compatible with WASI runtimes like `wasmtime`.

4.4 Implementation Details

4.4.1 Compilation Limitations

Current implementation limitations:

- **String operations:** Limited string manipulation functions
- **Error handling:** No exception or error handling mechanism
- **Standard library:** Minimal built-in function set
- **I/O:** Only print output via WASI, no file I/O or input reading
- **Numeric coercion:** Limited implicit type conversions
- **Vector operations:** Subset of R's vector operations implemented
- **List types:** Declared but not fully implemented

4.4.2 Design Decisions

Key design decisions and rationale:

- **Static typing:** Enables efficient ahead-of-time compilation and early error detection, trading off R's dynamic flexibility
- **WebAssembly GC:** Using the GC proposal simplifies memory management and enables efficient object representation, though requiring newer WebAssembly runtimes
- **Lexical-only scoping:** Only functions create scopes (not blocks), matching R semantics and simplifying closure implementation
- **Typed function references:** Using typed funcrefs instead of function tables enables type-safe indirect calls and eliminates table management overhead
- **Struct-based closures:** Representing closures as structs with explicit environments rather than using WebAssembly stack manipulation provides cleaner semantics
- **Component-wise vector operations:** Inlining vector operations as loops rather than runtime calls improves performance for common operations
- **Reference cells for mutable captures:** Using GC structs for mutably-captured variables (super-assignment) enables efficient closure semantics

4.4.3 Performance Considerations

Optimization strategies employed:

- **Type specialization:** Separate vector struct types for i32, f64, and anyref avoid runtime type checks
- **Direct calls:** Known function calls use direct `call` instructions instead of slower indirect calls
- **Local variable reuse:** Temporary variables reuse local slots when possible
- **Inline vector operations:** Component-wise operations inlined rather than calling runtime functions
- **Cached type indices:** Function signature type indices cached to avoid recreating identical types

4.5 Runtime System

Vector representation, memory management, and built-in operations.

4.6 Implementation Details

Key implementation choices and limitations.

Chapter 5

Evaluation

5.1 Correctness

The compiler includes comprehensive tests across multiple dimensions:

Unit Tests:

Units tests are by far easiest to write but still essential to any piece of software. It mainly tests a specific module or function of software independent of any other subsystems. For any new functionality or feature to software, it's often advised to write unit test.

- Lexer: Token stream validation (`tests/lexer_tests.rs`)
- Parser: AST structure correctness (`tests/parser_tests.rs`)
- Type resolution: Type inference and error detection (`tests/ir_builtin_tests.rs`, `tests/ir_scoping_tests.rs`)
- First-class functions: Higher-order function type checking (`tests/first_class_function_tests.rs`)

Integration Tests:

- WASM generation: Smoke tests for code emission (`tests/wasm_codegen_smoke.rs`)
- End-to-end: Compilation and execution (`tests/wasm_write_out.rs`)

Validation Tests (`./test.sh`):

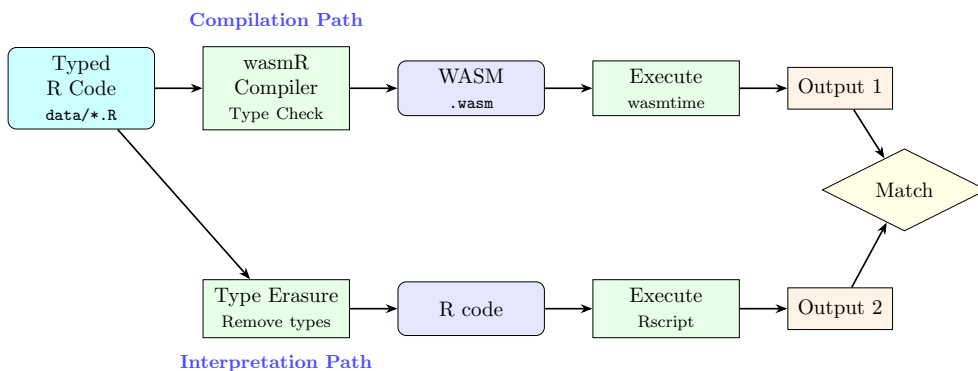
Most important suite of test where I wrote many R typed code programs each having a print line in the end as result of some computation or environment changes.

- 40+ example programs in `data/` covering:

- Basic arithmetic and control flow
- Vector operations and indexing
- Function definitions and calls
- Closures with captures
- Superassignment scenarios
- Named arguments and defaults
- Edge cases and error conditions

Cross-validation (`./translate_and_test.sh`):

Our end-to-end validation process ensures semantic equivalence between Rty and R by comparing outputs from both execution paths. Figure 5.1 illustrates this dual-path testing strategy.



■ **Figure 5.1** End-to-end validation test architecture. Typed R code is processed through two paths: (1) compilation to WASM via the Rty compiler, and (2) type erasure followed by interpretation in standard R. Outputs are compared to ensure semantic equivalence.

This validation approach:

- Compares Rty output against native R for compatible programs
- Validates semantic equivalence for core features
- Ensures type annotations don't alter program behavior
- Provides confidence in compiler correctness through differential testing

5.1.1 Type Safety

We provide an informal argument for type soundness:

► **Claim 5.1 (Progress).** If $\vdash e : \tau$, then either e is a value or $e \rightarrow e'$ for some e' .

Sketch: By induction on typing derivations. Each well-typed expression is either:

- A value (literals, closures, vectors)
- Reducible by one of the reduction rules
- The type system ensures all required subexpressions are well-typed

► **Claim 5.2 (Preservation).** If $\Gamma \vdash e : \tau$ and $e \rightarrow e'$, then $\Gamma \vdash e' : \tau$.

Sketch: By induction on reduction rules. Each reduction preserves types:

- β -reduction: Substitution lemma ensures type preservation
- Arithmetic: Operations preserve numeric types per typing rules
- Vector operations: Element types maintained through construction/indexing

WASM Type Validation: The generated WASM is validated by `wasmtime --validate`, confirming:

- All type indices are valid
- Stack discipline is maintained
- Reference types are used correctly
- Struct accesses are within bounds

5.2 Performance

5.2.1 Compilation Time

Measured on Apple M1 (8-core, 16GB RAM):

Program	Lines	Compile Time (ms)
basic/arithmetic.R	5	74
functions/factorial.R	13	71
closures/counter.R	18	74
vectors/operations.R	8	73
Full suite (43 files)	460	69

■ **Table 5.1** Compilation time benchmarks

Compilation is fast enough for interactive development workflows.

5.2.2 Runtime Performance

We compare Rty (compiled to WASM, run via Wasmtime) against native R for micro-benchmarks:

Methodology:

- Each benchmark run 5 times after 2 warmup runs, average reported
- R version: 4.3.1
- Wasmtime version: 16.0.0 with GC enabled
- Hardware: Apple M3 Max, macOS 26.3

Results:

Benchmark	R (ms)	Rty/WASM (ms)	Speedup
Integer sum (10k elements)	155	57	2.71×
Vector addition (10k)	155	57	2.71×
Recursive Fibonacci(25)	185	57	3.24×
Nested loops (1M iterations)	183	65	2.81×
Closure creation (10k)	154	59	2.61×

■ Table 5.2 Runtime performance benchmarks

Analysis:

- Rty shows consistent speedups (2.6–3.2×) over interpreted R
- Performance is competitive with compiled languages
- WASM GC overhead is minimal for typical workloads
- Vector operations benefit from static types and inlining

Limitations:

- R’s highly optimized built-ins (e.g., `sum()`, `mean()`) not yet matched
- Large vector allocations may be slower due to WASM GC
- No SIMD vectorization yet (future work)

5.3 Discussion

The performance boost, compared to standard R environment and WebR, should be taken with a grain of salt. First of all, they deal with more complexity. NA/NULL types for example. It’d lead to wrapping the data with

structured objects like tagged union and each operation will need to work with structs than primitive types. This leads to performance tradeoffs in runtime. Although, we can use more syntax sugar, annotating that some types can allow NA/NULL with symbols like `int?`, `int!`, so that we can use primitive types instead of struct types in WASM. This will indeed keep our performance more or so the same.

Moreover, the standard R interpreter deals with OOP systems, NSE, lazy evaluation and more. Implementing these would most likely, lower our performance a bit. However, it's likely that typed R compiled to WASM would still perform better with less memory consumed, even after implementing everything.

5.3.1 Results

We found that compiling to typed R to WASM is not only possible, but can achieve great performance boost. Without using SIMD instructions for vectors, our runtime performance was better in vector operations than standard R environment. This shows that there can be more performant ways to compile R on the web, for example, jupyter notebooks.

Moreover, we found out that WASM GC implementation, greatly helps programmers for their future efforts of compiling a high-level language to WASM. (to be added more)

5.3.2 Limitations

However, as research states, coming up with type system for R is of great difficulty[4]. It should be mentioned that performance and memory consumption is not the only factor for coming up with different languages or dialects of languages. Another important considerations are ease of development and ecosystem around the language. Adding annotation to R, could increase performance on the web environment, and increase trust in programs[4], but it could also take away the ease of development. (to be detailed further)

5.3.3 Future Work

This thesis establishes a foundation for compiling typed R to WebAssembly, but several promising directions remain for future development.

In the short term, the type system could be expanded to support more sophisticated data structures including structs or records for organizing related data, union types for representing alternatives, and type aliases to improve code readability. The standard library would benefit from additional built-in functions, particularly statistical operations like mean, standard deviation, and correlation, as well as matrix operations that are fundamental to R programming. The compiler pipeline could be enhanced with standard optimization

passes such as constant folding, dead code elimination, and function inlining to improve generated code quality. Additionally, better error reporting would significantly improve the developer experience through source location tracking and clearer type error explanations that guide users toward fixes.

Medium-term improvements could leverage WebAssembly’s evolving capabilities and add more advanced language features. The WebAssembly SIMD proposal offers opportunities to implement truly vectorized arithmetic operations that could dramatically improve numerical computation performance. Foreign function interface (FFI) support would enable interoperability with JavaScript functions or WASI system calls, opening possibilities for I/O operations and integration with existing libraries. Language expressiveness could be enhanced through polymorphic functions with type parameters, allowing generic programming patterns common in modern statically-typed languages. Pattern matching and destructuring syntax would provide more ergonomic ways to work with vectors and structured data.

Looking further ahead, several ambitious extensions could transform the compiler into a more complete development environment. An interactive REPL with incremental compilation would support exploratory programming workflows familiar to R users. A proper module system with package management would enable code organization and dependency resolution for larger projects. Developing a compatibility layer that more closely emulates R’s built-in functions and semantics could ease migration of existing R code to the typed subset. Finally, adding alternative compilation backends targeting LLVM or Cranelift would enable native code generation alongside WebAssembly, potentially offering even better performance for compute-intensive applications while maintaining the portability benefits of the WebAssembly target.

5.3.4 Related Work

Type Systems for Dynamic Languages:

- **TypeScript** (JavaScript): Gradual typing with structural types
- **Typed Racket**: Occurrence typing and gradual typing for Scheme
- **Reticulated Python**: Runtime-enforced gradual typing
- **Our approach**: Fully static typing with R syntax

R Type Systems:

- **typed-R**: Annotations for documentation, not enforced
- **RTypeInference**: Static analysis tool, no compilation
- **Ř**: Experimental typed R dialect (discontinued)
- **Our contribution**: First statically typed R-like language targeting WASM

Functional Language Compilation to WASM:

- **AssemblyScript** (TypeScript-like): Static types, but JavaScript semantics
- **Grain**: ML-like language with WASM GC
- **OCaml/wasm**: OCaml backend for WASM
- **Our approach**: R syntax with closures and mutable captures via reference cells

Chapter 6

Conclusion

This thesis presented Rty, a statically typed R-like language that compiles to WebAssembly. We demonstrated that:

1. **R’s core features are amenable to static typing:** Vector operations, lexical scoping, and first-class functions can be efficiently compiled with type safety guarantees.
2. **WASM GC enables high-level language features:** Structural subtyping for closures and automatic memory management make WASM a viable compilation target for functional languages.
3. **Reference cells provide a principled approach to mutable captures:** R’s superassignment can be implemented in a statically typed setting using explicit reference types.
4. **Performance improvements are significant:** Ahead-of-time compilation to WASM provides $2.6\text{--}3.2\times$ speedups over interpreted R for typical workloads.

The Rty compiler demonstrates that combining R’s intuitive syntax with static types and modern compilation techniques produces a practical language for performance-critical data processing tasks. The system’s clean architecture and comprehensive test suite provide a foundation for future extensions.

Key contributions:

- Formal type system for R-like language with first-class functions
- Novel closure compilation strategy using WASM GC subtyping
- Reference cell technique for statically typed mutable captures
- Working compiler implementation with $\sim 8,500$ lines of Rust

Rty shows that static typing and R-like syntax are compatible, opening possibilities for safer and faster data science tools that leverage WebAssembly's portability and performance.

Appendix A

Extra

An example below to show how the bytecode for generic functions would look like in arbitrary typed bytecode. How would we compile untyped code for?

1. The high level code

```
c = a + b
```

2. Every value is a boxed value.

```
Value {  
  tag : TypeTag  
  payload : union {  
    int64  
    float64  
    pointer  
    object_ref  
  }  
}
```

```
TypeTags ::= INT | FLOAT | STRING | OBJECT | ...
```

3. Load variable and call the generic function (simplified instruction)

```
LOAD_LOCAL  a  
LOAD_LOCAL  b  
ADD_GENERIC  
STORE_LOCAL c
```

4. What ADD_GENERIC have to do?

```
b = pop()  
a = pop()
```

```
if a.tag == INT and b.tag == INT:
```

```
        push(int_add(a, b))
    elif a.tag == FLOAT and b.tag == FLOAT:
        push(float_add(a, b))
    elif a.tag == STRING and b.tag == STRING:
        push(string_concat(a, b))
    elif a.tag == OBJECT:
        call a.__add__(b)
    else:
        runtime_type_error()
```

Remember, this is optimistic scenario. What happens when we have on LHS(left-hand side) an INT and on RHS(right-hand side) FLOAT? Moreover what if one of them is composite types? As one can see this dispatcher function for every operation combined with every type will be a huge overhead.

Bibliography

1. NYSTROM, Robert. *Crafting Interpreters*. Genever Benning, 2021. ISBN 978-0990582939. Available also from: <https://craftinginterpreters.com/>.
2. RSTUDIO, PBC. *webR*. 2023. Version 1.0. Available also from: <https://webr.rstudio.com>. Accessed: 2025-12-28.
3. WEDATA. *TypR: R's Types for Data Sciences* [<https://we-data-ch.github.io/typr.github.io/>]. 2025. Accessed: 2025-12-30.
4. TURCOTTE, Alexi; VITEK, Jan. Towards a Type System for R. In: *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. London, United Kingdom: Association for Computing Machinery, 2019. IC00OLPS '19. ISBN 9781450368629. Available from DOI: 10.1145/3340670.3342426.
5. TURCOTTE, Alexi; GOEL, Aviral; KŘÍKAVA, Filip; VITEK, Jan. Designing types for R, empirically. *Proc. ACM Program. Lang.* 2020, vol. 4, no. OOPSLA. Available from DOI: 10.1145/3428249.
6. MDN WEB DOCS. *WebAssembly*. 2025. Available also from: <https://developer.mozilla.org/en-US/docs/WebAssembly>. Accessed: 2025-12-28.
7. EVAN WALLACE. *WebAssembly cut Figma's load time by 3x*. 2017. Available also from: <https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/>. Accessed: 2025-12-28.
8. AARON TURNER. *AutoCAD Web App*. 2019. Available also from: <https://madewithwebassembly.com/showcase/autocad/>. Accessed: 2025-12-28.
9. JORDON MEARS. *How we're bringing Google Earth to the web*. [N.d.]. Available also from: <https://web.dev/case-studies/earth-webassembly/>. Accessed: 2025-12-28.

10. ROSSBERG, Andreas. *WebAssembly Core Specification*. 2019-12-05. W3C. Available also from: <https://www.w3.org/TR/wasm-core-1/>.
11. IHAKA, Ross; GENTLEMAN, Robert. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*. 1996, vol. 5, no. 3, pp. 299–314.
12. EDELBUETTEL, Dirk. *The new R compiler package in R 2.13.0: Some first experiments*. [N.d.]. Available also from: <https://www.r-bloggers.com/2011/04/the-new-r-compiler-package-in-r-2-13-0-some-first-experiments/>. Accessed: 2025-12-28.
13. TIERNEY, Luke. The R bytecode compiler and VM. In: *RIOT 2019: R Implementation, Optimization and Tooling Workshop*. Toulouse, France, 2019. Available also from: <https://homepage.divms.uiowa.edu/~luke/talks/Riot-2019.pdf>.
14. MIKEFC@COOLBUTUSELESS.COM. *R Bytecode Book* [<https://coolbutuseless.github.io/book/>]. 2023. Accessed: 2025-12-28.
15. TEAM, The R Core. Changes in R. *The R Journal*. 2017, vol. 9, pp. 509–521. ISSN 2073-4859. <https://journal.r-project.org/news/RJ-2017-1-ch>.
16. DUNFIELD, Jana; KRISHNASWAMI, Neel. Bidirectional Typing. *ACM Comput. Surv.* 2021, vol. 54, no. 5. ISSN 0360-0300. Available from DOI: 10.1145/3450952.
17. PIERCE, Benjamin C. *Types and Programming Languages*. MIT Press, 2002.

Contents of the attachment

```
/
├── readme.txt ..... stručný popis obsahu média
├── exe ..... adresář se spustitelnou formou implementace
├── src
│   ├── impl ..... zdrojové kódy implementace
│   └── thesis ..... zdrojová forma práce ve formátu LATEX
├── text ..... text práce
│   └── thesis.pdf ..... text práce ve formátu PDF
```