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 6, 2026

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Compilation of a typed R-like language to WebAssembly
Student:	Bilguudei Baljinnyam
Supervisor:	Pierre Donat-Bouillud, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science 2021
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2026/2027

Instructions

R is an open-source programming language and software environment widely used for statistical computing and data analysis. Originally developed for statisticians, it has become the de facto standard in many scientific disciplines including bioinformatics, data analysis, finance, or data mining. In R, vectors are first-class citizens, as even 1 is represented as a vector of size 1. R is a dynamic language and is notoriously difficult to compile; it is currently only JIT-compiled in experimental projects [1]. WebAssembly [2] is an open, safe and portable compilation target for programming languages, enabling deployment in the web or locally. Currently, the R Interpreter can be compiled to WebAssembly [3] but R programs themselves cannot be.

The goal of the thesis is to define and compile a statically-typed subset of R, or R-like. It should be simple enough to be able to compile it, but expressive enough to express operations on vectors and perform statistical operations. The target architecture will be WebAssembly.

1) Research the R language and WebAssembly.
2) Define and formalize a typed subset of R. It should at least support vector operations, simple control-flow instructions, and function calls.
3) Write a compiler from this subset to WebAssembly
4) Write a runtime for the subset.
5) Evaluate the compiler and the runtime.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 23 June 2025 in Prague.

[1] Flückiger, O., Chari, G., Yee, M. H., Ječmen, J., Hain, J., & Vitek, J. (2020). Contextual dispatch for function specialization. Proceedings of the ACM on Programming Languages, 4(OOPSLA), 1-24.
[2] https://webassembly.org/
[3] https://github.com/r-wasm/webr

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

In Prague on January 6, 2026

Abstract

R is a widely-used dynamic language for statistical computing, but its dynamic nature prevents efficient ahead-of-time compilation. This thesis presents the design and implementation of a compiler for a statically-typed subset of R, called Typed 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 an easily extendable runtime system written in Typed R. 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

List of Figures

List of Tables

List of Code listings

List of abbreviations

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

Chapter 1
Introduction

The R programming language is ubiquitous for statistical computing and data analysis, offering powerful abstractions for data manipulation and visualization. As an interpreted language, R requires its interpreter to execute programs: the source code is parsed into an abstract syntax tree (AST) and evaluated directly, rather than being compiled to machine code [1, 2].

With the advent of WebAssembly (WASM), a portable compilation target for the web, new possibilities have emerged for running programs in browsers. Several languages have been brought to the browser through WebAssembly, typically by compiling their interpreters to WASM. For example, Pyodide compiles the Python interpreter to WebAssembly, requiring approximately 10 to 20MB for the core runtime [3]. Similarly, WebR [4] compiles the R interpreter to WebAssembly, enabling R code to run in modern browsers.

However, to our knowledge, there has been no attempt to directly compile R programs to WebAssembly. This thesis explores that alternative: instead of compiling the interpreter and then interpreting R programs within the browser, we compile R programs themselves directly to WebAssembly bytecode. This approach requires type annotations to enable ahead-of-time compilation, but offers potential benefits such as smaller output binaries and elimination of interpreter overhead.

This work contributes to multiple communities: data scientists seeking interactive web-based tools, educators building browser-based R tutorials, and researchers exploring compilation techniques for dynamic languages. It also provides a case study in compiling a typed dialect of a dynamic language to WebAssembly's garbage-collected type system.

The structure of the thesis follows this premise. Chapter 2 introduces the concepts of R, WebAssembly, and previous works. Chapter 3 presents the rationale for creating Typed R and introduces the language itself, both formally and informally with examples. Chapter 4 describes the implementation of the compiler and runtime, and Chapter 5 discusses the evaluation of the compiler and the performance of the generated binaries in comparison to standard R and WebR.

1.1 Aim and Research Questions

This thesis aims to determine whether R's core semantics (particularly vector operations, lexical scoping, and first-class functions) can be compiled to WebAssembly through static typing, and whether such compilation is feasible as an alternative to interpreter-based approaches.

Specifically, we aim to:

1. Demonstrate that a statically-typed subset of R can be compiled directly to WebAssembly
2. Find out How can R's features (closures, superassignment, vectors) be represented in WebAssembly's type system
3. Understand ahead-of-time compilation of a typed R subset to WebAssembly feasible, and what are the resulting performance characteristics?

To explore these questions, we design **Typed R**, a statically-typed R dialect, and implement a compiler targeting WebAssembly GC. While type system design is necessary for this exploration, our primary focus is on compilation feasibility and demonstrating the approach rather than advancing type system theory. More sophisticated typing proposals for R exist [5, 6, 7]. Our goal is to define a subset of R simple enough to compile to WASM, yet expressive enough to demonstrate R's core features such as vectorized mathematical operations.

Chapter 2

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 [8], which was an experimental language developed for the same purpose by Bell Labs. 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.

R is fundamentally designed around vector operations, where everything, even scalar values, is represented as a vector. This design manifests in the language's support for **vectorization**, enabling operations to apply element-wise across data structures without explicit loops, as seen in expressions like `c(1,2,3) + c(4,5,6)` ⇒ `c(5,7,9)`. The language employs **dynamic typing** with types determined at runtime rather than through variable declarations, facilitating rapid prototyping and interactive data exploration at the cost of static verification. Combined with **lexical scoping** and closures, where functions capture their defining environment, R supports functional programming patterns that enable higher-order abstractions and metaprogramming capabilities essential for statistical computing. The language's **lazy evaluation** strategy delays argument evaluation until values are required, allowing non-standard evaluation mechanisms that support domain-specific sublanguages. To preserve referential transparency, R implements **copy-on-modify semantics**, implicitly copying objects upon modification to prevent unexpected side effects, though this approach can introduce performance overhead in memory-intensive workloads.

2.1.1 Type System

R has 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. 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"
  ```

  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"
```

R performs implicit type coercion following a hierarchy: logical → integer → double → 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.

2.2 WebAssembly

WebAssembly is a low-level assembly-like language that is made to run in modern browser [9]. 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 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 [10], Autodesk AutoCAD Web App [11], and Google Earth [12].

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. From personal experience, WAT format really eases development and debugging process by removing friction of using different tools to analyze the bytecode. Moreover, WASM's ecosystem provides useful tools like `wasm2wat, wat2wasm, wasm-tools validate` etc, which makes development better.

Let's talk about essential characteristics of WASM. First of all, WebAssembly code is organized into modules. Modules declare imports (functions, memory, tables, globals from the host environment) and exports (making internal definitions available externally) [13]:

```
(module
  ;; Import a logging function from the host
  (import "env" "log" (func $log (param i32)))

  ;; Import memory from the host
  (import "env" "memory" (memory 1))

  ;; Internal function (not exported)
  (func $internal_add (param $a i32) (param $b i32) (
     result i32)
    local.get $a
    local.get $b
    i32.add
  )

  ;; Can write memory
  ;; Can create data segment

  ;; Public function that uses imports
  (func $add_and_log (param $a i32) (param $b i32) (
     result i32)
    local.get $a
    local.get $b
    call $internal_add    ;; call internal function
    local.get 0           ;; duplicate result for logging
    call $log             ;; log the result
  )

  ;; Export the public function
  (export "addAndLog" (func $add_and_log))
)
```

Code listing 2.1 Module demonstrating import and export mechanisms

Moreover, 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:

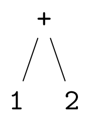


Figure 2.1 Tree representation

```
(module
  (func $compute (result i32)
    i32.const 1        ;; push constant 1 onto stack
    i32.const 2        ;; push constant 2 onto stack
    i32.add            ;; pop two values, push sum
  )
  (export "compute" (func $compute))
)
```

Code listing 2.2 A simple WebAssembly function showing stack operations

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 [13].

```
(module
  (func $max (param $a i32) (param $b i32) (result i32)
    local.get $a
    local.get $b
    i32.gt_s              ;; signed greater-than
        comparison
    (if (result i32)
      (then
        local.get $a
      )
      (else
        local.get $b
      )
    )
  )
  (export "max" (func $max))
)
```

Code listing 2.3 Conditional execution using structured control flow

WebAssembly provides a contiguous, resizable array of bytes called linear memory. Memory is accessed via load/store instructions with explicit alignment and offset [13].

```
(module
  (memory 1)              ;; declare 1 page (64KB) of
      memory
  (func $increment_at_zero
    i32.const 0           ;; memory address
    i32.const 0           ;; memory address (for load)
    i32.load              ;; load 32-bit value from
        address 0
    i32.const 1           ;; constant 1
    i32.add               ;; increment
    i32.store             ;; store result back to address
         0
  )
  (export "memory" (memory 0))
  (export "increment" (func $increment_at_zero))
)
```

Code listing 2.4 Accessing linear memory with load and store instructions

There's still more to discuss about features and characteristics of of WASM. We can take a look at instructions, traps, data segment, element segment, tables but they are not of utmost importance for reading this thesis. The reader may look explore more in details in WASM specs[13].

2.2.1 WebAssembly Garbage Collection proposal

WASM Garbage collection proposal, in short GC, is a new proposal that provides 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. For example, if one wants to compile an array in WASM, one has to find a slot from the linear memory that WASM provides, bookkeep the pointer and the size, calculate the amount to allocate, then allocate the array. It's non-trivial work and the code like this for low-level environment leads to unnecessary pain, where bugs are easy to make, but hard to debug due to readability issues of low-level codes even with formats like WAT. Fortunately, they introduced high-level constructs such as structs and arrays with the proposal, which we'll heavily use in the implementation of this thesis. It's another layer of abstraction that' ll helps me to write less code and support more feature. To hint on the key features, and how it's going to help us:

- Struct Types: Define record-like structures for representing vectors and closures
- Array Types: Efficiently store vector element data
- Subtyping: Enable polymorphic closure environments (crucial for Chapter 4)

```
(type $vec_f64 (struct
  (field $data (ref (array (mut f64))))
  (field $length (mut i32))
))
```

Code listing 2.5 Example: A vector of doubles can be represented

We will dive deep into these in Chapter 4.

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 [8]. 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 [14]. Essentially, it's faster to compile R into some intermediate bytecode and then have it executed by VM [15]. Thus, standard R environment execution became either with AST interpreter or with Bytecode Compiler [2]. From then on all standard functions and packages in base R ws pre-compiled into bytecode. Lastly, JIT bytecode compiler became the default from R.3.4.1 [16].

On top of standard implementations, execution on different environments have also been explored. WebR [4], 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.

Although there's not much work other than WebR that tries to exactly compile R to WASM, there are some interesting works that tries to compile other dynamic languages to WASM. Rasm,[17] a prototype compiler for a subset of Racket, demonstrates that programs from a lexically scoped, functional language can be translated directly to WebAssembly without embedding an interpreter, though only for a restricted language fragment and without any GC, so memory used is limited. Research on ahead-of-time compilation of JavaScript[18] to WebAssembly similarly explores alternatives to interpreter compilation, often relying on partial evaluation or specialization of an interpreter to preserve dynamic semantics. These approaches tend to retain significant runtime complexity in the generated code, limiting their practicality. The present thesis differs from interpreter-based and specialization-based techniques by adopting a statically typed subset of R, using type annotations to enable more direct compilation. Rather than aiming for full compatibility, this work focuses on feasibility and expressiveness, particularly with respect to R's vectorized operations and lexical scoping. As such, it positions itself as a complementary exploration of direct compilation, rather than a replacement for existing R execution environments.

Chapter 3

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. The first reason is typing.

Let's take the following code.

```
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

```
(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.

```
if x is int && y is int:
  add_int(x, y)
if x is double && y is double:
  add_double(x, y)
...
```

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 inferrable at compile time.

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

```r
x <- 5          # numeric
x <- "hello"    # now a string
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.

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

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

This requires complex bookkeeping that's hard to optimize. Then, we have one of the main challenging feature with any attempts of ahead-of-compilation. The reflection. R code can inspect and modify itself at runtime:

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

Lastly, many R functions evaluate arguments in non-standard ways:

```r
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.

Table 3.1 compares the data types supported by R and Typed R, showing which primitive and composite types are available in each language.

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

Table 3.2 presents a broader comparison of language features, including type system properties, functional programming capabilities, and advanced features that distinguish Typed R from standard 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. It's also worth to mention that implementation of those features take a lot of engineering time and effort. Therefore, I have only implemented what I see as essential and most used for demonstrating the point of this thesis.

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.

Let's start with examples before diving into formalities of the language. Starting with variables, they are declared using the left-assignment operator with optional type annotations:

```
# Simple assignment with type inference
x <- 10

# Assignment with explicit type annotation
y: int <- 20

# Vector assignment
vec <- c(1, 2, 3)
```

Code listing 3.1 Variable assignment examples

After variable assignment, we have very interesting syntax for super assignment. R distinguishes between assigning a variable in the scope or outside the scope with different syntax, unlike other popular dynamic languages like Python or Javascript. In Typed R, we'll do the same.

```
outer <- function() {
    x <- 0
    inner <- function() {
        x <<- 10  # Modifies x in outer scope
    }
    inner()
    return(x)  # Returns 10
}
```

Code listing 3.2 Super-assignment example

I should also note that on Code fragment 3.2 above, Typed R looks exactly same as R. For trivial cases like this, type inference can help a lot, which we will talk in next chapter.

Moving onto functions, they are first-class values defined using the `function` keyword:

```
# Simple function with type annotations
add <- function(a: int, b: int): int {
    return(a + b)
}

# Function returning a vector
create_vector <- function(): vector<double> {
    return(c(1.0, 2.0, 3.0))
}
```

```

# Higher-order function
apply_twice <- function(f: int -> int, x: int): int {
    return(f(f(x)))
}
```

Code listing 3.3 Function definition examples in typed R

First-class value mean they are essentially same as objects like vectors. They can be passed as parameter, saved to variable, can be returned from function.

In terms of control-flow, I attempted to support everything similar as R. Starting with if-else:

```
# If statement
if (x > 0) {
    print(x)
}

# If-else statement
if (x > 0) {
    y <- 1
} else {
    y <- -1
}

# If expression (returns value)
result <- if (x > 0) { 1 } else { -1 }
```

Code listing 3.4 Conditional examples in typed R

However, in R there's syntactic sugar where `if` that return expression can be written as `if (expr) 1 else 0`, without brackets. It's not implemented in our compiler.

Moreover, like in R, two loop constructs are provided: `for` and `while`.

```
# For loop iterating over range
for (i in 1:10) {
    print(i)
}

# For loop iterating over vector
vec <- c(1, 2, 3, 4, 5)
for (x in vec) {
    print(x)
}

# While loop
i <- 1
sum <- 0
while (i <= 5) {
    sum <- sum + i
    i <- i + 1
}
```

Code listing 3.5 Loop examples

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:

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

Code listing 3.6 Block with tail expression

Only functions create new scopes; blocks, if-statements, and loops share their enclosing function's scope. A **closure** is a function that captures variables from its defining environment. When a function references a variable from an outer function scope, that variable remains accessible even after the outer function returns. Super-assignment (`<<-`) modifies captured variables by searching enclosing function scopes.

Example demonstrating closure:

```
make_counter <- function(start: int): () -> int {
    count <- start
    function(): int {
        count <<- count + 1
        return(count)
    }
}

counter <- make_counter(0)
print(counter())  # Prints 1
print(counter())  # Prints 2
```

Code listing 3.7 Closure example

3.3.1 Abstract Grammar

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

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. Type inference fails for programs lacking sufficient type information, requiring explicit annotations in those cases. The type grammar defines three major categories:

$$
\begin{array}{rll}
\tau ::= & \mathit{num} \mid \texttt{void} & \text{(base type)} \\
 & \mid \tau\,\texttt{[]} & \text{(vector type)} \\
 & \mid (\tau_1, \ldots, \tau_n) \to \tau & \text{(function type)} \\
\mathit{num} ::= & \texttt{int} \mid \texttt{double} \mid \texttt{bool} & \text{(numeric types)}
\end{array}
$$

Typed R builds upon three core numeric types that align naturally with WebAssembly's type system. The `int` type represents 32-bit signed integers, mapping directly to WebAssembly's `i32` primitive, with values ranging from -2^{31} to $2^{31} - 1$. The `double` type provides 64-bit IEEE 754 floating-point numbers, corresponding to WebAssembly's `f64`, matching R's default numeric type for statistical computations. Boolean logic uses the `logical` type with R-style literals `TRUE` and `FALSE`, represented as `i32` in WebAssembly where zero means false and non-zero means true. Finally, `void` denotes the absence of a value for functions that perform side effects without returning data.

Beyond primitive types, vectors form the core composite structure. Vectors are homogeneous arrays parameterized by element type:

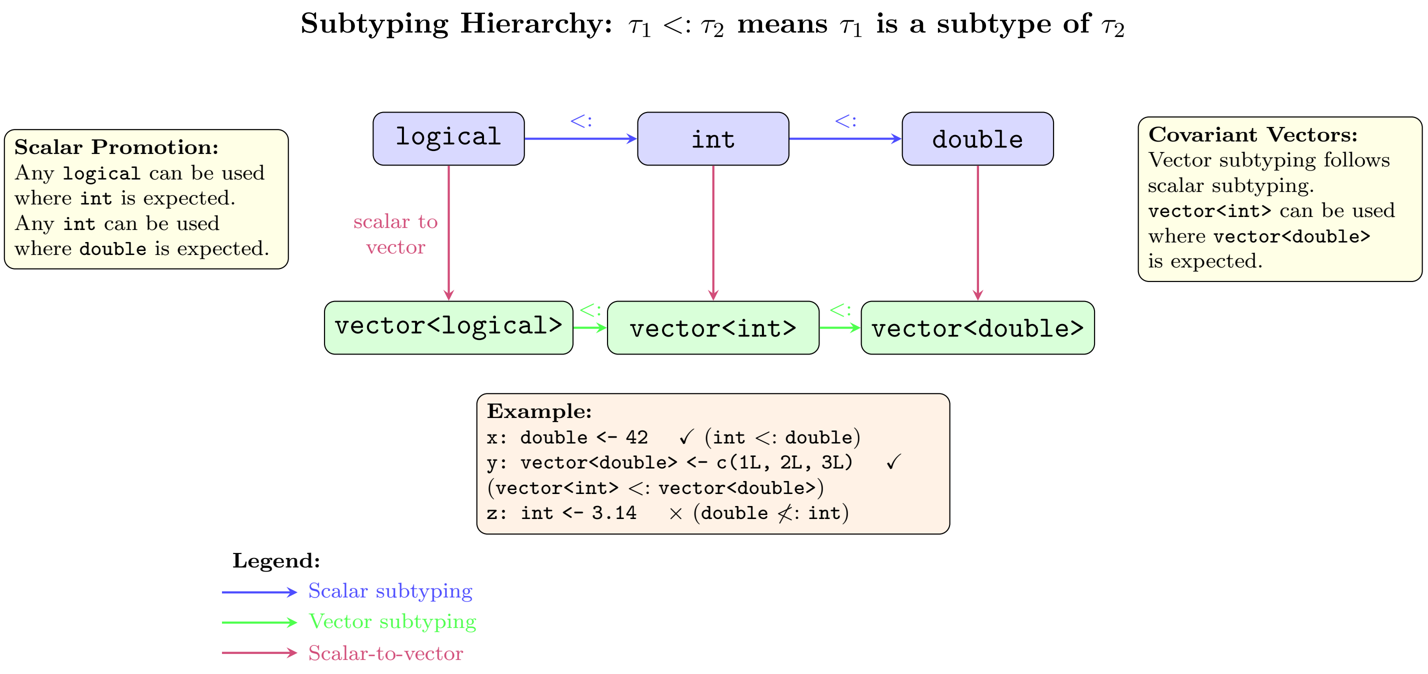
```
# Vector of integers
v1: vector<int> <- c(1, 2, 3)

# Vector of doubles
v2: vector<double> <- c(1.5, 2.5, 3.5)

# Vector operations (component-wise)
v3: vector<double> <- v2 + c(1.0, 2.0, 3.0)
```

Code listing 3.8 Vector type examples

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

Function types complete the type system by encoding callable values with their parameter and return types. The simplest form, `int -> int`, describes a function accepting a single integer and producing an integer result. Multi-parameter functions extend this: `int, int -> double` takes two integers and returns a double. Higher-order functions naturally fit this notation, such as `(int -> int) -> int`, which accepts a function mapping integers to integers and returns an integer. Complex signatures like `int, (int, int -> int) -> int` combine these patterns, taking both a plain integer and a binary function as parameters. To avoid ambiguity in expressions like `int -> int -> int`, we require explicit parentheses in type annotations, ensuring each signature has exactly one interpretation.

Type promotion follows a subtyping hierarchy shown in Figure 3.1. Logical values can be used wherever integers are expected, and integers can be used wherever doubles are expected, creating the chain `logical` $<:$ `int` $<:$ `double`. Vectors are covariant with respect to their element types, meaning `vector<int>` can be used where `vector<double>` is expected, preserving the subtyping relationship across composite types.


Subtyping Hierarchy: $\tau_1 <: \tau_2$ means τ_1 is a subtype of τ_2
Scalar Promotion: Any logical can be used where int is expected. Any int can be used where double is expected.
logical
<:
int
<:
double
scalar to vector
vector<logical>
<:
vector<int>
<:
vector<double>
Covariant Vectors: Vector subtyping follows scalar subtyping. vector<int> can be used where vector<double> is expected.
Legend:
Scalar subtyping
Vector subtyping
Scalar-to-vector

Example:
`x: double <- 42` ✓ (`int` $<:$ `double`)
`y: vector<double> <- c(1L, 2L, 3L)` ✓ (`vector<int>` $<:$ `vector<double>`)
`z: int <- 3.14` × (`double` $\not<:$ `int`)

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.

The language provides no user-creatable subtyping; we don't support R's

S3 system or OOP features in Typed R.

Type checking follows a bidirectional approach[19] that combines bottom-up inference and top-down checking. Bottom-up inference derives expression types from literal values and operator signatures, while top-down checking uses function return types and variable annotations to provide expected types. Unification solves type constraints to determine concrete types, incorporating the subtyping rules to enable implicit type promotion.

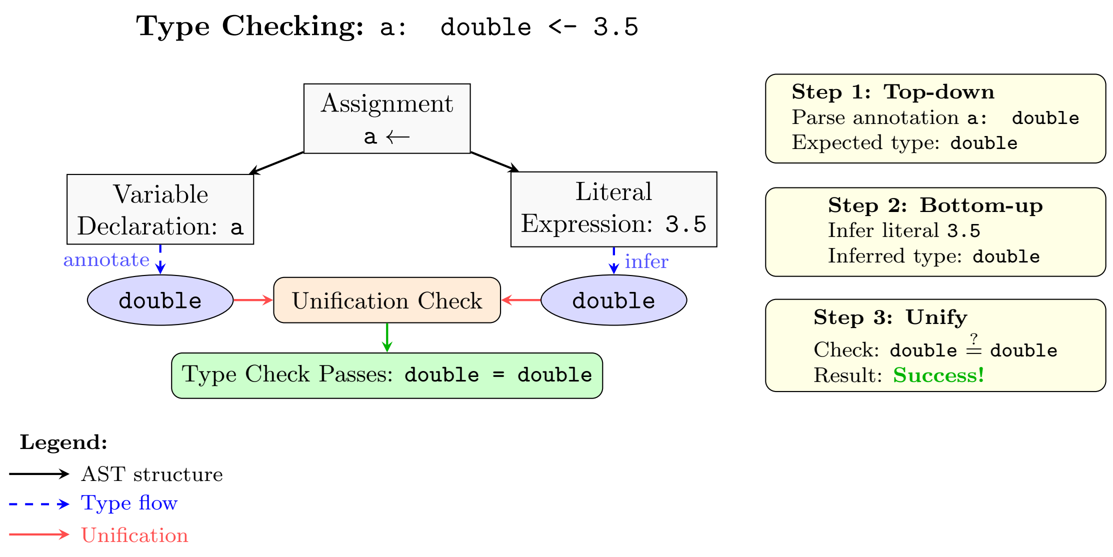


Figure 3.2 Bidirectional type checking for variable assignment. The type annotation `double` provides the expected type (top-down), while the literal `3.5` has its type inferred as `double` (bottom-up). The unification step verifies that both types match, allowing the assignment to proceed.

The basic algorithm works as follows: when type-checking an expression, if the type can be inferred from the expression itself (bottom-up), that inferred type is used. If an explicit type annotation is provided (top-down), the annotation is used as the expected type. When both are present, unification checks that the inferred type is compatible with (a subtype of) the annotated type. If neither inference nor annotation provides type information, type checking fails.

Type annotations are required for function parameters, function return types (when not inferrable from return statements), and ambiguous variable declarations. For example, in `a: double <- 3.5`, the literal `3.5` infers as `double` (bottom-up), the annotation expects `double` (top-down), and unification confirms they match (see Figure 3.2). The expression `3 + 2.5` demonstrates type promotion: `3` infers as `int` but is promoted to `double` through subtyping to match `2.5`, evaluating as `3.0 + 2.5 = 5.5`.

We do not formalize the complete type inference algorithm here, as bidirectional type inference is well-studied in programming language literature[19]. Our implementation follows standard techniques, extended with the subtyping hierarchy described in Section 3.1.

The formal type system uses inference rules with typing judgments of the form $\Gamma \vdash e : \tau$, read as "under typing context Γ, expression e has type τ." A typing context maps variable names to their types, defined as $\Gamma ::= \emptyset \mid \Gamma, x : \tau$, where $\emptyset$ is the empty context and $\Gamma, x : \tau$ extends context Γ with a binding of variable x to type τ. We write $x : \tau \in \Gamma$ when Γ contains the binding. Rules appear in natural deduction style with premises above the horizontal line and conclusions below; axioms have no premises. Our type system follows standard approaches from programming language theory [20], extended with subtyping for numeric promotion and vector operations.

Subtyping defines when one type can safely substitute for another. 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). Every type is a subtype of itself, and the numeric promotion hierarchy establishes base cases:

$$\frac{}{\tau <: \tau}\ \textsc{S-Refl} \qquad\qquad \frac{}{\texttt{logical} <: \texttt{int}}\ \textsc{S-LogicalInt}$$

$$\frac{}{\texttt{int} <: \texttt{double}}\ \textsc{S-IntDouble}$$

S-Refl states that any type is a subtype of itself (reflexivity). S-LogicalInt allows logical values to be used as integers. S-IntDouble allows integers to be promoted to doubles, enabling mixed arithmetic like `3 + 2.5`. Subtyping is transitive, allowing chains like `logical` $<:$ `int` $<:$ `double`:

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ \textsc{S-Trans}$$

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, giving us `logical` $<:$ `double` automatically. Vectors are covariant in their element type:

$$\frac{\tau_1 <: \tau_2}{\texttt{vector}\langle\tau_1\rangle <: \texttt{vector}\langle\tau_2\rangle}\ \textsc{S-Vector}$$

When element type τ_1 is a subtype of τ_2, 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. The subsumption rule connects subtyping to the type system, enabling implicit type promotion:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{T-Sub}$$

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, enabling all implicit conversions. For example, if `x` has type `int`, we can use it where `double` is expected.

Literals and variables form the basic building blocks of expressions:

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{int}}\ \textsc{T-Int} \qquad \frac{d \in \mathbb{R}}{\Gamma \vdash d : \texttt{double}}\ \textsc{T-Double}$$

$$\frac{b \in \{\texttt{TRUE}, \texttt{FALSE}\}}{\Gamma \vdash b : \texttt{logical}}\ \textsc{T-Bool} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{T-Var}$$

Integer literals like `42` have type `int` (T-Int), floating-point literals like `3.14` have type `double` (T-Double), and `TRUE`/`FALSE` have type `logical` (T-Bool). Variables have whatever type the context Γ assigns them (T-Var).

Arithmetic operators (`+`, `-`, `*`, `/`, `%%`) follow the same typing pattern, shown here for addition:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\ \textsc{T-AddInt}$$

$$\frac{\Gamma \vdash e_1 : \texttt{double} \qquad \Gamma \vdash e_2 : \texttt{double}}{\Gamma \vdash e_1 + e_2 : \texttt{double}}\ \textsc{T-AddDouble}$$

When both operands have the same numeric type, the result has that type. Mixed-type arithmetic like `3 + 2.5` applies subsumption to promote `3` from `int` to `double` before applying T-AddDouble. Comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) require both operands to have the same type after potential promotion and always produce `logical` results:

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\texttt{int}, \texttt{double}, \texttt{logical}\}}{\Gamma \vdash e_1 \odot e_2 : \texttt{logical}}\ \textsc{T-Compare}$$

where $\odot \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}$. Logical AND (`&`), OR (`|`), and NOT (`!`) operate on boolean values:

$$\frac{\Gamma \vdash e_1 : \texttt{logical} \qquad \Gamma \vdash e_2 : \texttt{logical}}{\Gamma \vdash e_1 \ \texttt{\&}\ e_2 : \texttt{logical}} \ \textsc{T-And}$$

$$\frac{\Gamma \vdash e_1 : \texttt{logical} \qquad \Gamma \vdash e_2 : \texttt{logical}}{\Gamma \vdash e_1 \ \texttt{|}\ e_2 : \texttt{logical}} \ \textsc{T-Or}$$

$$\frac{\Gamma \vdash e : \texttt{logical}}{\Gamma \vdash \texttt{!}e : \texttt{logical}} \ \textsc{T-Not}$$

The `c()` function constructs vectors from elements, requiring all elements to have the same type τ after promotion:

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \cdots \qquad \Gamma \vdash e_n : \tau}{\Gamma \vdash \texttt{c}(e_1, e_2, \ldots, e_n) : \texttt{vector}\langle\tau\rangle} \ \textsc{T-Vector}$$

For example, `c(1, 2, 3)` has type `vector<double>`. The colon operator creates sequences:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1, \tau_2 \in \{\texttt{int}, \texttt{double}\}}{\Gamma \vdash e_1\texttt{:}e_2 : \texttt{vector}\langle\tau_1 \sqcup \tau_2\rangle} \ \textsc{T-Range}$$

where $\tau_1 \sqcup \tau_2$ denotes the **least upper bound** (join) of the two types under the subtyping relation:

$$\texttt{int} \sqcup \texttt{int} = \texttt{int} \qquad \texttt{double} \sqcup \texttt{double} = \texttt{double} \qquad \texttt{int} \sqcup \texttt{double} = \texttt{double}$$

The range `1:5` creates a sequence from 1 to 5, where the element type is the wider of the two bounds: `1L:5L` produces `vector<int>`, while `1:5` or `1L:5.0` produces `vector<double>`. Vector indexing uses bracket notation, producing a value of the element type:

$$\frac{\Gamma \vdash e_1 : \texttt{vector}\langle\tau\rangle \qquad \Gamma \vdash e_2 : \tau' \qquad \tau' \in \{\texttt{int}, \texttt{double}\}}{\Gamma \vdash e_1[e_2] : \tau} \ \textsc{T-Index}$$

The index can be an integer or double (truncated to integer at runtime), and follows R's 1-based indexing convention. Binary operators extend component-wise to vectors when both operands have the same numeric vector type:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \texttt{vector}\langle\tau\rangle \\ \Gamma \vdash e_2 : \texttt{vector}\langle\tau\rangle \qquad \tau \in \{\texttt{int}, \texttt{double}\}\end{array}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{vector}\langle\tau\rangle} \ \textsc{T-VecBinOp}$$

where $\oplus \in \{+, -, *, /, \%\%\}$. For example, `c(1,2) + c(3,4)` produces `c(4,6)`. Scalars broadcast to match vector length:

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

A scalar is implicitly replicated, so `2 * c(1,2,3)` produces `c(2,4,6)`. The symmetric case (vector $\oplus$ scalar) follows analogously.

Functions are first-class values with explicit parameter and return types. To type-check a function, we extend the context with parameter bindings and verify the body has the declared return type:

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \texttt{function}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \ \{\ e\ \} : (\tau_1, \ldots, \tau_n) \to \tau} \text{ T-Func}$$

The function itself has function type $(\tau_1, \ldots, \tau_n) \to \tau$. Function application checks that arguments match parameter types (with promotion via subsumption):

$$\frac{\Gamma \vdash e_0 : (\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_0(e_1, \ldots, e_n) : \tau} \text{ T-App}$$

The result has the function's return type. Conditional expressions require a logical condition and matching branch types:

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

Both branches must have type τ, which becomes the expression's type, enabling constructs like `x <- if (cond) { 1 } else { 2 }`. Assignment binds a value to a variable, extending the context:

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

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. Super-assignment modifies a variable in an enclosing scope:

$$\frac{x : \tau \in \Gamma_{outer} \qquad \Gamma \vdash e : \tau'}{\Gamma \vdash (x \ll\!\leftarrow e) : \tau'} \text{ T-SuperAssign}$$

where $\tau' <: \tau$ (the assigned value must be compatible with the existing binding). Super-assignment requires that x already exists in some enclosing function scope (Γ_{outer}), and the expression type must be compatible with the existing variable's type. Blocks are sequences of statements with an optional tail expression:

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

Statements are checked in sequence, with each potentially extending the context for subsequent statements. The block's type is determined by its final expression. Return statements have the type of their argument, which the type checker verifies matches the enclosing function's declared return type:

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

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

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

3.5.2 Vector Operations

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

$$\texttt{c}(v_1, \ldots, v_n) \oplus \texttt{c}(w_1, \ldots, w_n) = \texttt{c}(v_1 \oplus w_1, \ldots, v_n \oplus w_n)$$

When the length of the vector differs, we use recycling like in R. Essentially the vector of lower length is permutated until it has same or more length as the vector with higher length and operation is executed. For R, when the length of one vector is not divisible by the other, meaning the permutated vector will have additional elements, that won't be included in the operation, standard R environment shows a warning message. However, we don't have warning message in our compiler yet, it won't be provided.

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

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

For example, `2 * c(1, 2, 3)` produces `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

It'll be discussed more in details in compiler and runtime environment in Chapter 4.

3.5.4 Built-in Functions

The language provides the following built-in functions:

Function	Type	Description
`c(e1, ..., en)`	$(\tau, \dots, \tau) \to \texttt{vector}\langle\tau\rangle$	Creates vector from elements
`print(e)`	$\tau \to \texttt{void}$	Outputs value to stdout
`length(v)`	$\texttt{vector}\langle\tau\rangle \to \texttt{int}$	Returns vector length
`sum(v)`	$\texttt{vector}\langle\tau\rangle \to \tau$	Sums numeric vector

Table 3.3 Built-in functions and their types

Chapter 4
Compiler and Runtime

This chapter describes the compiler implementation, highlighting challenges and design decisions made while compiling Typed R to WebAssembly. All implementation is written in Rust for its memory safety, modern features, and performance.

4.1 Compiler Architecture

The compiler follows a multi-pass architecture divided into front-end, middle-end, and back-end stages. The front-end transforms source code into an intermediate representation through lexical analysis, parsing, and type resolution. The middle-end performs language-independent optimizations and transformations on the IR, preparing it for efficient code generation. Finally, the back-end generates WebAssembly bytecode from the transformed IR. Compilation begins with the lexer (`src/lexer.rs`), which tokenizes source text into a stream of tokens, recognizing R-specific operators like `<-`, `<<-`, and `:`, while tagging built-in type names as `Token::Type`. These tokens feed into the parser (`src/parser/`), which constructs an untyped Abstract Syntax Tree using recursive descent parsing[21] with operator precedence. The parser handles function definitions as expressions and accepts type annotations without validating them, producing `Stmt` and `Expr` nodes defined in `src/ast.rs`.

The `TypeResolver` (`src/ir/type_resolver.rs`) then performs scope analysis, building a scope stack where only functions create new scopes. It infers types for untyped expressions, validates type annotations and operation compatibility, and produces typed IR nodes with concrete types. The resulting IR (`src/ir/types.rs`) consists of `IRExpr` with associated `ty: Type` fields, `IRStmt` with type information, and built-in call resolution to `BuiltinKind` enum variants.

A pass manager (`src/ir/passes/manager.rs`) coordinates three transformation passes that prepare the IR for code generation. The Variable Collection Pass assigns WebAssembly local indices to all variables, tracking parameters, user variables, and compiler-generated temporaries while populating `FunctionMetadata` for each function. The Captured Variables Analysis identifies variables captured from parent scopes, computes transitive captures through nested functions, and marks variables requiring reference cells for superassignment. The Function Flattening Pass lifts nested functions to top level, replacing them with closure construction expressions and maintaining capture lists for environment building.

Finally, the backend (`src/backend/`) generates WebAssembly bytecode. The `WasmGenerator` constructs WASM module sections, registers type sections for structs, arrays, and functions, emits function code with local context tracking, and initializes memory for runtime data.

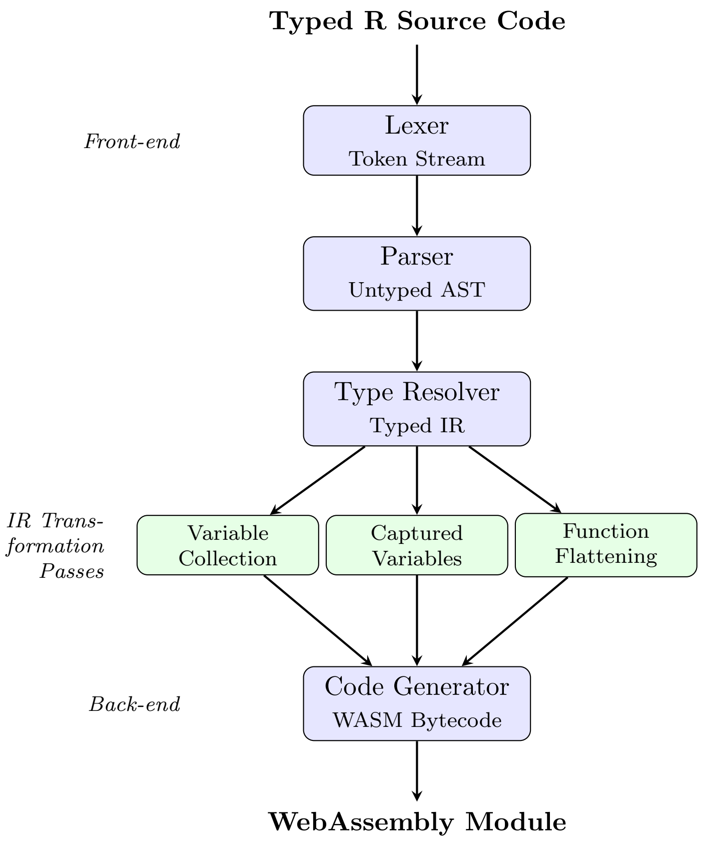


Figure 4.1 Typed R compiler pipeline showing the multi-pass architecture from source code through lexing, parsing, type resolution, IR transformation passes, to WebAssembly code generation

4.2 WebAssembly Code Generation

Most languages compile to WebAssembly through intermediate layers like LLVM[22] or JVM rather than generating WebAssembly directly. This approach avoids the N×M explosion problem where every language would need to support every architecture. LLVM provides a common intermediate representation that many static languages use to reach WebAssembly. Similarly, JVM-based languages gained WebAssembly support when the JVM added a WebAssembly backend. Our goal is compiling Typed R directly to WebAssembly, requiring careful mapping of R semantics to WebAssembly's type system and execution model.

Type mapping follows WebAssembly's value types: `int` maps to `i32`, `double` to `f64`, and `logical` to `i32` where 0 represents false and 1 represents true. Vectors map to `(ref $vec_T)`, implemented as WebAssembly GC structs containing a data array and length field. Functions map to `(ref $functype)`, using WebAssembly's typed function references.

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)

Vectors are represented as WebAssembly GC structs with two fields: a mutable array for element data and a mutable integer for length:

```
(type $vec_f64 (struct
  (field $data (ref (array (mut f64))))
  (field $length (mut i32))
))
```

Code listing 4.1 Vector struct type

This design uses WebAssembly GC arrays for efficient storage, stores length separately for fast access, supports mutable arrays for in-place updates, and provides type-specialized structs for different element types (i32, f64, anyref). WebAssembly GC handles memory management automatically. Vectors are constructed through runtime functions and built-ins like `c`, `seq`, and `rep`.

Functions use WebAssembly typed function references. Simple functions compile to direct `(ref $functype)` references where `$functype` encodes the function signature. Closures require a struct containing both the function reference and the captured environment:

```
(type $closure (struct
  (field $func (ref $functype))
  (field $env (ref $env_struct))
))
```

Code listing 4.2 Closure struct type

Function calls use `call_ref` for indirect calls through function references.

Literals compile to immediate value instructions: `i32.const n` for integers, `f64.const x` for floating-point, and `i32.const 0` or `i32.const 1` for booleans. Unlike R, which uses vectors even for scalar values, Typed R keeps scalar values for better performance. Variable references compile to `local.get $var_idx` for local variables, environment struct loads for captured variables, and `ref.func $func_idx` for function references.

Binary operations map to corresponding WebAssembly instructions: `i32.add`, `i32.sub`, `i32.mul` for integer arithmetic; `f64.add`, `f64.sub`, `f64.mul` for floating-point; comparison operators like `i32.eq`, `i32.lt_s` for integers and `f64.eq`, `f64.lt` for floats; and `i32.and`, `i32.or` for logical operations with boolean normalization. Vector operations compile to loops applying scalar operations element-wise: allocate a result vector matching operand length, loop over indices, extract elements from operands, apply the scalar operation, and store results. All vector operations are written in Typed R and compiled into the runtime. For example:

```
system_vector_add___vec_int__vec_int <- function(a:
    vector<int>, b: vector<int>): vector<int> {
     n <- length(a)
     m <- length(b)

     if(n != m & n %% m != 0 & m %% n != 0) {
          stop("Vector lengths not compatible for recycling
              ")
     }

     result_len <- max(n, m)
     result: vector<int> <- vec(length=result_len, mode="
         int")

     for(i in 1:result_len) {
          a_idx <- ((i - 1) %% n) + 1
          b_idx <- ((i - 1) %% m) + 1
          result[i] <- a[a_idx] + b[b_idx]
     }
     return(result)
}
```

Such function is compiled to WASM and the operation vec + vec is mapped to this function name. Function calls compile differently based on callee type. Direct calls to known functions use `call $func_idx`. Indirect calls through function variables load the function reference from a local or environment, evaluate arguments, then use `call_ref $functype_idx`. Closure calls are more complex: load the closure struct, extract the environment and function fields, pass the environment as the first parameter followed by regular arguments, and invoke via `call_ref $closure_functype_idx`.

Conditional expressions compile to WebAssembly block structures. The `if` construct is represented as an expression that can return a value, containing a condition, then-branch, and optional else-branch. The compiler verifies at compile-time that both branches return matching types when the if-expression produces a value:

```
; Evaluate condition
(condition code)

; If-else structure
(if (result ...)
  (then
    ; then-branch code
    ; when expr, just leave the expr on stack
  )
  (else
    ; else-branch code
    ; when expr, just leave the expr on stack
  )
)
```

Code listing 4.3 If-else compilation

When the if-expression returns a value, it remains on the stack for the next consumer.

Assignment statements come in two forms. Regular assignment (`<-`) evaluates the right-hand side expression and stores the result using `local.set $var_idx`. Super-assignment (`<<-`) for captured variables evaluates the expression, loads the closure environment, navigates to the appropriate nesting level, extracts the reference cell for the variable, and updates cell contents using `struct.set`.

Loops compile straightforwardly. While loops become loop blocks with conditional branching:

```
(block $loop_exit
  (loop $loop_start
    ; Evaluate condition
    (condition code)

    ; Exit if false
    (i32.eqz)
    (br_if $loop_exit)

    ; Loop body
    (body code)

    ; Continue loop
    (br $loop_start)
  )
)
```

Code listing 4.4 While loop compilation

For loops over ranges compile to indexed loops:

```
; Initialize loop variable to start
(local.set $iter (start value))

(block $loop_exit
  (loop $loop_start
    ; Check condition: iter <= end
    (local.get $iter)
    (end value)
    (i32.gt_s)
    (br_if $loop_exit)

    ; Loop body with iter
    (body code)

    ; Increment iter
    (local.get $iter)
    (i32.const 1)
    (i32.add)
    (local.set $iter)

    (br $loop_start)
  )
)
```

Code listing 4.5 For loop compilation

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

4.2.1 Function Compilation

Function compilation is one of the most important. We are taking an environment where functions are first-class citizens to static-like function environment in WASM.

For example, how should one compile this?

```
        x <- 5L
        f <- function(): int {
            g <- function(): int {
                x + 1L
            }
            g()
        }
```

In WASM functions cannot be nested. So we need to propogate the environment information somehow through the function parameters or maybe in the `Memory`. But playing with Memory, will make the implementation much more complex. Therefore, the better is to use parameters with flattened functions.

```
        (function $g (param $x i32) result(i32)
          local.get 0
          i32.const 1
          i32.add
          return
        )

        (function $f (param $x i32) result(i32)
          ;; call g with param x
        )

        (function $main
          (local $x i32)
          i32.const 5
          local.set 0
          local.get 0 ;; put x on stack
          call $f ;; call f with param x
        )
```

This is the basis for scoping and functions. Now, let's take the case, where we mutate the variable that's outside the scope of function:

```
        x <- 5L
        f <- function(): int {
            g <- function(): int {
                x <<- x + 1L
                x
            }
            g()
        }
```

The variable is not only referenced but changed. However, this is not closure, the variable doesn't outlive its scope. To solve this, I use WASM's construct `struct` from WASM GC. Then we can have a `struct` reference to that struct and pass it around, and then finally change it.

```
        (struct $cell (field (mut i32)))

        (function $g (param $x (ref $cell)) result(ref
           $cell)
          local.get 0
          struct.get $cell 0 ; get first field of the
              struct cell.
          i32.const 1
          i32.add ; add and push the result
          struct.set $cell 0 ; pop the result and set in
             the cell
          struct.get $cell 0 ; get updated element from
             the struct
          return
        )

        (function $f (param $x (ref $cell)) result(ref
           $cell)
          ;; call g with param x
        )

        (function $main
          (local $x (ref $cell))
          i32.const 5
          struct.new $cell ; construct cell struct with
             expression on the stack.
          local.set 0 ; set the struct
          local.get 0 ;; put the struct on stack
          call $f ;; call f with param struct
        )
```

From here, we generalize this mechanism to handle true closures where functions outlive their lexical scope. The key is combining the reference cell approach with function pointers in a unified `env` struct. This structure contains both the captured variables we want to pass and modify, as well as a function pointer that allows us to call the closure later. By packaging these together, we create a closure representation that WebAssembly can work with efficiently.

The closure compilation strategy leverages structural subtyping to maintain type safety. We define a base environment type that all closures share, containing only the function pointer field. Each specific closure then defines a concrete environment type that extends this base with its particular captured variables. This approach allows closures with different capture sets to share a common interface while preserving their unique environments.

```
(type $env_base (struct
  (field $func_ptr (ref $closure_func_ty))))

(type $env_concrete (sub $env_base (struct
  (field $func_ptr (ref $closure_func_ty))
  (field $captured_x i32)
  (field $captured_y f64))))
```

Code listing 4.6 Base and concrete environment types

Consider a typical scenario where an outer function returns an inner function that captures local variables. The compiler analyzes which variables the inner function references, then generates a concrete environment struct type with fields for each captured variable. When the closure is created at runtime, the compiler emits code to allocate the environment struct, initialize its fields with current values of captured variables, and store the function pointer in the first field. The result is a reference to this struct, which serves as the closure value itself.

When a closure is invoked, accessing the captured variables requires a downcast. The function receives the environment as its first parameter typed as the base environment reference, but to access specific captured variable fields, the compiler must downcast it to the concrete environment type. WebAssembly validates this downcast at runtime, ensuring type safety. The function pointer stored in the environment allows indirect calls via `call_ref`, passing the environment as the first argument so the function can access its captures.

```
(type $env_concrete (struct
  (field $func_ptr (ref $closure_func_ty))
  (field $captured_x i32)))

(func $closure_body (param $env (ref $env_base)) (param
   $y i32) (result i32)
  (local $concrete_env (ref $env_concrete))

  ; Downcast to concrete type
  (local.set $concrete_env
    (ref.cast (local.get $env) (ref $env_concrete)))

  ; Access captured variable
  (struct.get $env_concrete $captured_x (local.get
     $concrete_env))
  (local.get $y)
  (i32.add)
)
```

Code listing 4.7 Closure accessing captured variables

For simple functions that capture no variables, compilation is more straightforward. The compiler registers the function type in the type section, allocates a unique function index, determines local variable slots from the function's metadata, and emits the function body directly. These functions require no environment parameter and compile to efficient direct calls.

```
(func $add (param $a i32) (param $b i32) (result i32)
  (local.get $a)
  (local.get $b)
  (i32.add)
)
```

Code listing 4.8 Simple function without captures

The complete function compilation process integrates type registration, closure analysis, environment struct generation, and code emission into a cohesive pipeline. For each function, the compiler identifies whether it captures variables from outer scopes through static analysis. If captures exist, it generates an appropriate environment struct type and modifies the function signature to accept the environment as its first parameter. The function body is then emitted with additional instructions to downcast the environment and access captured variables through struct field accesses. When a function is returned as a closure value, the compiler generates code to construct the environment struct at the return site, populate it with the current values of captured variables, store the function pointer in the first field, and return the struct reference representing the closure.

4.3 Runtime System

The runtime leverages WebAssembly GC for automatic memory management, treating vectors, and closures as GC-managed heap objects. This eliminates the need for explicit deallocation, as WebAssembly GC handles reference counting and garbage collection automatically. Linear memory is reserved for WASI I/O buffers and string serialization. The memory layout reserves address 0 for null pointer checks, allocates addresses from 8 onward for WASI I/O buffers used by `fd_write`, and maintains a dynamic region managed by the compiler for temporary string buffers.

Vector construction through `c(...)` determines the element type from arguments, allocates an array of appropriate size, initializes array elements, allocates the vector struct, stores the array reference and length, and returns the vector struct reference. Component-wise vector arithmetic is implemented as inline loops for performance rather than runtime calls, as described earlier. Reduction operations are simpler: `length` extracts and returns the length field from the vector struct, while `sum` loops over vector elements to accumulate their sum.

The `print` function handles output by serializing values to strings and calling WASI. It converts the value to its string representation, writes the string to a linear memory buffer, then calls `fd_write` to output to stdout (file descriptor 1). Different types require different serialization: integers and floats convert to decimal strings (with precision for floats), strings output directly, vectors format as `[elem1, elem2, ...]`, and booleans output as `TRUE` or `FALSE`.

Sequence generation through the range operator `start:end` or `gen_seq` function calculates the sequence length as `end - start + 1`, allocates a vector of integers, fills the array with values from start to end, and returns the vector struct. The implementation uses a simple loop without intermediate allocations for efficiency.

WASI integration provides I/O capabilities through the WebAssembly System Interface. The runtime imports `fd_write` from `wasi_snapshot_preview1` with signature `(i32, i32, i32, i32) -> i32`, accepting parameters for file descriptor, iovs pointer, iovs length, and nwritten pointer. This enables output to stdout and stderr. The exported `_start` function serves as the entry point, maintaining compatibility with WASI runtimes like Wasmtime.

Moreover, we have two ways to extend our runtime. Either add inline WASM code in the source code, like how `print` works, or write Typed R program, which will be compiled and embedded into the top of the generated WASM code. The latter simplifies the implementation and makes it easy to extend our runtime. For mathematicals, and library functions, we can just write Typed R program and they'll be compiled to WASM as runtime.

4.4 Implementation Details

4.4.1 Compilation Limitations

The current implementation represents a proof-of-concept compiler with several deliberate limitations. Most notably, the compiler lacks any exception or error handling mechanism, meaning runtime errors simply terminate execution without graceful recovery. The standard library remains minimal, providing only essential built-in functions like `print`, `length`, `sum`, and vector construction. I/O capabilities are restricted to print output via WASI; file operations and user input are not supported. Type coercion is limited compared to R's flexible type system, supporting only the basic numeric promotion hierarchy (logical <: int <: double). Vector operations cover common arithmetic and indexing but lack many of R's specialized functions like `apply`, `sapply`, or statistical operations. List types, while declared in the type system, remain largely unimplemented in the code generator.

4.4.2 Design Decisions

Several architectural decisions shaped the compiler's implementation, each involving trade-offs between performance, simplicity, and compatibility with R semantics. The choice of static typing fundamentally enables efficient ahead-of-time compilation and early error detection, though at the cost of R's dynamic flexibility. This trade-off aligns with the thesis goal of exploring compilation rather than interpretation.

Memory management leverages the WebAssembly GC proposal, which simplifies object lifecycle management and enables efficient heap object representation without manual deallocation. This choice requires newer WebAssembly runtimes but eliminates the complexity of implementing a custom garbage collector or reference counting system. The scoping model follows R's semantics where only functions create scopes while blocks, if-statements, and loops share their enclosing function's scope. This design simplifies closure implementation by avoiding the need to track block-level environments.

For function calls, the compiler uses typed function references rather than function tables. Typed funcrefs enable type-safe indirect calls and eliminate table management overhead, though this feature requires WebAssembly reference types support. Closures are represented as structs containing a function reference and captured environment, providing cleaner semantics than stack manipulation approaches. Variables captured with super-assignment semantics use reference cells (GC structs wrapping mutable values), enabling efficient updates through multiple closure layers.

Vector operations are inlined as loops rather than compiled to runtime function calls. While this increases code size, it improves performance for common element-wise operations by avoiding function call overhead and enabling better optimization by the WebAssembly runtime.

The implementation relies on two key tools: wasm-encoder for code generation and Wasmtime for execution. The wasm-encoder library (part of the wasm-tools project) provides a programmatic API for constructing WebAssembly modules in Rust. Unlike alternatives such as writing WebAssembly Text Format (WAT) and assembling with wat2wasm, or using the older parity-wasm library, wasm-encoder offers type-safe module construction with strong guarantees about validity, making it easier to generate correct bytecode. The library's design matches naturally with Rust's type system, reducing the likelihood of encoding errors.

For runtime execution and testing, Wasmtime serves as the WebAssembly runtime engine. Wasmtime, maintained by the Bytecode Alliance, implements the latest WebAssembly specifications including the GC proposal, WASI, and component model. Alternatives like Wasmer or browser-based runtimes were considered, but Wasmtime provides the most mature support for WebAssembly GC, which is essential for this compiler's memory management strategy. Its command-line interface simplifies testing during development, and its Rust embedding API enables potential integration with higher-level tooling. Wasmtime's focus on security, standards compliance, and performance makes it well-suited for both development and production use.

4.4.3 Performance Considerations

The compiler employs several optimization strategies to generate efficient WebAssembly code. Type specialization creates separate vector struct types for i32, f64, and anyref element types, avoiding runtime type checks and enabling more efficient memory layouts. When function call targets are known at compile time, the compiler generates direct `call` instructions rather than slower indirect `call_ref` instructions, reducing call overhead significantly.

Local variable allocation reuses slots for temporary variables when their lifetimes don't overlap, minimizing the function's local declaration section and potentially improving WebAssembly JIT compilation. As discussed earlier, component-wise vector operations are inlined as loops rather than calling runtime functions, trading code size for execution speed. The compiler also caches function signature type indices to avoid recreating identical types in the WebAssembly type section, reducing module size and improving instantiation time. These optimizations focus on low-hanging fruit that provide measurable benefits without requiring complex analysis passes.

Chapter 5
Evaluation

This chapter evaluates the Typed R compiler implementation along two critical dimensions: correctness and performance. We first present the testing methodology and validation approach used to ensure the compiler produces correct WebAssembly code that faithfully implements R semantics. We then analyze compilation time characteristics and runtime performance through benchmarking against standard R implementations.

5.1 Correctness

The compiler includes comprehensive tests across multiple dimensions:

5.1.1 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 adviced 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.r`

5.1.2 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`)

5.1.3 Validation Tests

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.

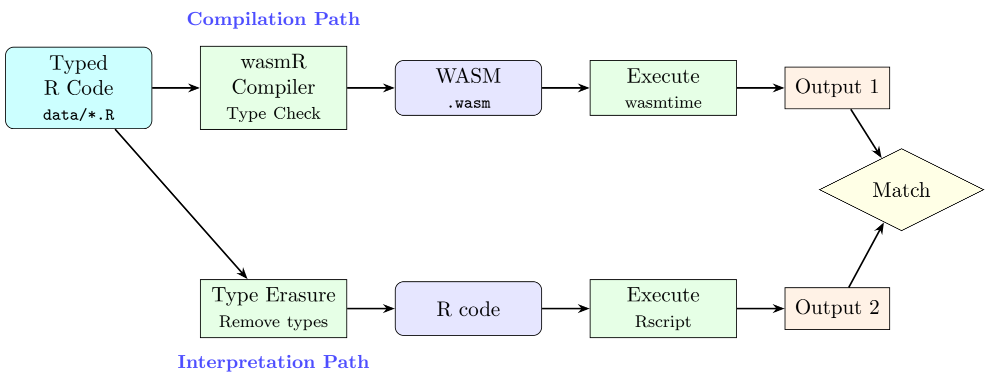


Figure 5.1 End-to-end validation test architecture. Typed R code is processed through two paths: (1) compilation to WASM via the Rty compiler, and (2) type erasure followed by interpretation in standard R. Outputs are compared to ensure semantic equivalence.

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

5.2 Performance

5.2.1 Compilation Time

Table 5.1 presents compilation time measurements for representative programs of varying complexity. All measurements were performed on Apple M1 hardware (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

Table 5.2 presents the runtime performance comparison between native R and Typed R compiled to WebAssembly. The benchmarks cover various computational patterns including vector operations, recursion, loops, and closure creation.

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:

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

- 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[6]. 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[6], 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.

Chapter 6

Conclusion

The primary objective of this thesis was to explore the feasibility of compiling a statically-typed subset of R directly to WebAssembly. Specifically, we aimed to:

1. Design a statically-typed language that retains R's core characteristics, including vector operations, lexical scoping, and first-class functions
2. Develop a compiler that efficiently maps these high-level features to WebAssembly bytecode
3. Evaluate both the correctness and performance of the compiled code
4. Demonstrate that R's dynamic features can be expressed with static types without sacrificing usability

These objectives have been successfully fulfilled through the design and implementation of Rty, a statically typed R-like language with a complete compiler targeting WebAssembly. The key achievements include:

- **Language design**: Defined a type system that preserves R's vector-oriented programming model while enabling static type checking. The language supports first-class functions, closures, lexical scoping, and type promotion, maintaining familiar R syntax with explicit type annotations.
- **Compilation strategy**: Implemented a multi-pass compiler architecture that translates Rty programs to WebAssembly. The compiler leverages the WebAssembly GC proposal for automatic memory management and uses reference cells to implement R's superassignment semantics in a statically-typed setting.
- **Correctness validation**: Developed comprehensive test suites including unit tests, integration tests, and cross-validation against standard R implementations, ensuring semantic equivalence between Rty and R for the supported feature set.

- **Performance evaluation**: Demonstrated that ahead-of-time compilation to WebAssembly provides significant performance improvements, achieving 2.6–3.2× speedups over interpreted R for typical computational workloads including vector operations, recursion, loops, and closures.

The thesis contributions include a formal type system for an R-like language with first-class functions, a novel closure compilation strategy using WASM GC structural typing, a reference cell technique for statically typed mutable captures, and a working compiler implementation comprising approximately 8,500 lines of Rust code.

Rty demonstrates 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. The system's architecture and comprehensive test suite provide a foundation for future extensions and research in compiling dynamic languages to WebAssembly.

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 combinated 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. HENDERSON, Mike. *R Bytecode Book* [[Online]]. 2020. Available also from: `https://coolbutuseless.github.io/book/rbytecodebook/`. [Accessed: 2025-12-28].
3. DROETTBOOM, Michael. *Pyodide: Bringing the scientific Python stack to the browser*. Mozilla Hacks, 2019. Available also from: `https://hacks.mozilla.org/2019/04/pyodide-bringing-the-scientific-python-stack-to-the-browser/`.
4. RSTUDIO, PBC. *webR* [[Online]]. 2023. Version 1.0. Available also from: `https://webr.rstudio.com`. [Accessed: 2025-12-28].
5. WEDATA. *TypR: R's Types for Data Sciences* [[Online]]. 2025. Available also from: `https://we-data-ch.github.io/typr.github.io/`. [Accessed: 2025-12-30].
6. 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. ICOOOLPS '19. ISBN 9781450368629. Available from DOI: `10.1145/3340670.3342426`.
7. TURCOTTE, Alexi; GOEL, Aviral; KŘIKAVA, Filip; VITEK, Jan. Designing types for R, empirically. *Proc. ACM Program. Lang.* 2020, vol. 4, no. OOPSLA. Available from DOI: `10.1145/3428249`.
8. 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. ISSN 1061-8600.

9. MDN WEB DOCS. *WebAssembly* [[Online]]. 2025. Available also from: https://developer.mozilla.org/en-US/docs/WebAssembly. [Accessed: 2025-12-28].

10. WALLACE, Evan. *WebAssembly cut Figma's load time by 3x* [[Online]]. 2017. Available also from: https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/. [Accessed: 2025-12-28].

11. TURNER, Aaron. *AutoCAD Web App* [[Online]]. 2019. Available also from: https://madewithwebassembly.com/showcase/autocad/. [Accessed: 2025-12-28].

12. MEARS, Jordon. *How we're bringing Google Earth to the web* [[Online]]. 2019. Available also from: https://web.dev/case-studies/earth-webassembly/. [Accessed: 2025-12-28].

13. ROSSBERG, Andreas. *WebAssembly Core Specification* [[Online]]. 2019. W3C. Available also from: https://www.w3.org/TR/wasm-core-1/. [Accessed: 2025-12-28].

14. EDDELBUETTEL, Dirk. *The new R compiler package in R 2.13.0: Some first experiments* [[Online]]. 2011. 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].

15. 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.

16. 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.

17. MATEJKA, Grant. *Rasm: Compiling Racket to WebAssembly*. 2022. MA thesis. California Polytechnic State University.

18. FALLIN, Chris. Ahead-of-Time Compilation of JavaScript via Interpreter Specialization. 2024. Unpublished research notes and talks.

19. DUNFIELD, Jana; KRISHNASWAMI, Neel. Bidirectional Typing. *ACM Comput. Surv.* 2021, vol. 54, no. 5. ISSN 0360-0300. Available from DOI: 10.1145/3450952.

20. PIERCE, Benjamin C. *Types and Programming Languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

21. SNEPSCHEUT, Jan L. A. van de. Recursive Descent Parsing. In: *What Computing Is All About*. New York, NY: Springer New York, 1993, pp. 101–120. ISBN 978-1-4612-2710-6. Available from DOI: 10.1007/978-1-4612-2710-6_6.

22. LATTNER, Chris; ADVE, Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. Palo Alto, California: IEEE Computer Society, 2004, p. 75. CGO '04. ISBN 0769521029.

Contents of the attachment

The attached media contains the complete Rty compiler implementation, test suite, and thesis sources. The directory structure is organized as follows:

```
/
├── README.md ............ project overview and build instructions
├── Cargo.toml ........... Rust package manifest and dependencies
├── src .................. compiler source code (∼6,000 lines)
│   ├── lib.rs ........... library entry point and compilation orchestration
│   ├── lexer.rs ......... lexical analysis (350 lines)
│   ├── ast.rs ........... abstract syntax tree definitions
│   ├── types/ ........... cross-cutting type system
│   ├── parser/ .......... syntactic analysis (1,200 lines)
│   ├── ir/ .............. intermediate representation (2,000 lines)
│   │   └── passes ....... IR transformation passes
│   ├── backend .......... WebAssembly code generation (2,500 lines)
│   │   ├── mod.rs ....... WASM generator and module construction
│   │   ├── context.rs ... local context for code emission
│   │   ├── emit/ ........ code emission by construct
│   │   └── wasm/ ........ WASM-specific utilities
│   └── driver/ .......... compilation orchestration (500 lines)
├── runtime_embed/ ....... runtime library in R
│   ├── numeric_ops.R .... numeric operations (min, max, abs)
│   ├── utility_ops.R .... utility functions (seq, sum, rep)
│   ├── vector_ops.R ..... vector operations
│   └── ..
├── tests ................ Rust unit and integration tests (∼2,500 lines)
│   └── ..
├── data ................. example R programs for testing
│   ├── basic ............ arithmetic and basic operations
│   ├── types ............ type inference and casting
│   └── ..
└── benchmarks ........... performance benchmarks
```

```
│
├── test.sh ............................... end-to-end validation script
├── translate_and_test.sh ............... cross-validation with native R
├── measure_compile_time.sh ........... compilation time measurement
└── out ............................ compiler output directory (generated)
    ├── *.wasm ........................ compiled WebAssembly modules
    └── *.wat .............................. WebAssembly text format
```