

Capitolo 1

Generazione di WebAssembly

La ricerca di soluzioni efficienti e portabili per eseguire codice in ambienti diversi è diventata una priorità fondamentale nell'informatica moderna. In questo contesto, WebAssembly (Wasm) emerge come una tecnologia chiave, fornendo un formato binario sicuro, veloce e indipendente dalla piattaforma. Nel corso di questo capitolo, esploriamo il processo di generazione di codice WebAssembly attraverso un compilatore dedicato a un linguaggio personalizzato. Il nostro linguaggio, creato per soddisfare specifiche esigenze o paradigmi di programmazione unici, si propone di offrire una flessibilità senza precedenti agli sviluppatori. Attraverso un compilatore appositamente progettato, saremo in grado di tradurre il codice sorgente del nostro linguaggio in istruzioni Wasm, consentendo così l'esecuzione di programmi in un ambiente virtuale altamente performante e sicuro. Nel corso di questo capitolo, esamineremo in dettaglio il processo di compilazione, passando attraverso le fasi cruciali che trasformano il nostro codice sorgente in un modulo WebAssembly. Dalla rappresentazione intermedia alla gestione delle dipendenze, esploreremo come il compilatore si adatta alle specificità del nostro linguaggio per garantire una corretta esecuzione e ottimizzazione delle risorse. Il capitolo si propone inoltre di approfondire le sfide e le opportunità che emergono durante il processo di generazione di WebAssembly. Analizzeremo le scelte di progettazione del compilatore, l'ottimizzazione del codice e la gestione delle risorse, fornendo un quadro completo delle considerazioni che guidano il nostro approccio alla generazione di codice Wasm.

1.1 Introduzione WebAssembly

WebAssembly (Wasm) è un formato di istruzioni altamente performante, portabile e sicuro, progettato per essere eseguito in ambienti virtuali. Orig-

nariamente concepito da un gruppo di lavoro congiunto tra Google, Mozilla, Microsoft e Apple, il principale obiettivo di WebAssembly è fornire un formato binario indipendente dalla piattaforma, garantendo al contempo sicurezza e velocità. Sebbene sia stato ideato principalmente per essere eseguito all'interno dei browser web, Wasm trova applicazioni anche in contesti diversi come server, dispositivi IoT e applicazioni desktop.

Le istruzioni Wasm si distinguono dalle istruzioni di un processore reale in quanto sono progettate per l'esecuzione in un ambiente virtuale. Ciò implica che le istruzioni Wasm non possono essere eseguite direttamente da un processore fisico; richiedono invece una traduzione preliminare in istruzioni native. Questo processo di traduzione è gestito da un motore di runtime, il quale interpreta le istruzioni Wasm e le traduce in istruzioni native del sistema ospite. Oltre alla traduzione delle istruzioni, il motore di runtime si occupa anche della gestione della memoria e delle risorse del sistema, fornendo un'astrazione sicura e indipendente dalla piattaforma.

In sintesi, WebAssembly rappresenta una tecnologia versatile e potente che offre un'alternativa efficace alle tradizionali soluzioni di esecuzione di codice, consentendo l'esecuzione di applicazioni web e non solo in modo efficiente, sicuro e indipendente dalla piattaforma di destinazione.[2]

1.2 WebAssembly Text Format

Il formato di testo WebAssembly (WAT) [**WebAssemblyTextFormat**] è un formato di testo leggibile dall'uomo per la rappresentazione di moduli WebAssembly. Il formato è stato progettato per essere utilizzato come rappresentazione intermedia durante il processo di compilazione, fornendo un'astrazione leggibile dall'uomo per il codice Wasm. Il compilatore che abbiamo sviluppato generiamo un file .wat ed in seguito il tool wat2wasm [1] genera il file .wasm, che a sua volta potrà essere eseguito da un motore di runtime.

I tipi di dati che troviamo in wat sono:

- **i32** 32-bit integer
- **i64** 64-bit integer
- **f32** 32-bit float
- **f64** 64-bit float

Un singolo parametro (*param i32*) e il tipo di ritorno (*result i32*).

```
1 (func (param i32) (param i32) (result f64) ...)
```

Listing 1.1: Esempio di funzione in wat

I parametri locali possono essere dichiarati all'interno di una funzione, e sono accessibili solo all'interno della funzione stessa. I comandi *local.get* e *local.set* vengono utilizzati per accedere agli indici dei parametri locali. Possiamo usare anche l'operatore *\$* per accedere ai parametri locali, in maniera più human-readable.

```
1 (func $fun (param i32) (param i32) (result f64)
2   (local $par1 i32)
3   (local $par2 i32)
4   (local $par3 f64)
5   ...
6   (local.get $par1)
7   (local.get $par2)
8   (local.set $par3)
9   ...
10 )
```

Listing 1.2: Esempio di funzione in wat

Come vediamo in questo esempio la funzione *\$fun* prende in input due parametri di tipo intero e ritorna un valore di tipo float. Inoltre all'interno della funzione vengono dichiarati tre parametri locali, due di tipo intero e uno di tipo float.

Stack Machine

L'esecuzione del WebAssembly é definita in termini di Stack-Machine, dove l'idea generale é che ogni tipo di istruzione esegue operazioni di tipo *push/pop* dallo stack. Quando viene chiamata una funzione, inizia con uno stack vuoto che viene gradualmente riempito e svuotato man mano che le istruzioni del corpo vengono eseguite. Quindi, ad esempio, dopo aver eseguito la seguente funzione:

```
1 (func $somma (param $p1 i32) (param $p2 i32)
2   (result i32)
3   local.get $p1
4   local.get $p2
5   i32.add)
```

Quando viene chiamata la funzione *\$somma*, viene passato il precedente valore nella pila come parametro *\$p1*. La prima istruzione *local.get* copia il valore di *\$p1* nello stack, e la seconda istruzione *local.get* copia il valore di *\$p2* nello stack. Infine, l'istruzione *i32.add* rimuove i due valori superiori dello stack, li somma e inserisce il risultato nello stack. Alla fine dell'esecuzione

della funzione, lo stack contiene il risultato della somma dei due valori passati come parametro. Per eseguire la **chiamata della funzione** precedente, vediamo il codice seguente:

```
1 (func $main
2   (result i32)
3   i32.const 10
4   i32.const 5
5   call $somma)
```

La prima istruzione *i32.const* inserisce il valore costante 10 nello stack, e la seconda istruzione *call* chiama la funzione *\$somma*. La funzione *\$somma* viene eseguita, e il risultato viene inserito nello stack. Alla fine dell'esecuzione della funzione, lo stack contiene il risultato della somma dei due valori passati come parametro. Dobbiamo inoltre aggiungere una dichiarazione di esportazione per fare in modo che la funzione sia visibile all'esterno del modulo (per esempio anche dal codice javascript).

```
1 (export "main"(func $main))
```

La prima stringa “main” é il nome della funzione che vogliamo esportare e che sarà visibile anche all'esterno del modulo, mentre la seconda é l'identificativo della funzione a cui fa riferimento.

1.3 CostCompiler to WAT

Il compilatore che abbiamo sviluppato genera un file .wat, questo file .wat andrà poi convertito in un file .wasm, che a sua volta potrà essere eseguito da un motore di runtime. Questa conversione viene fatta tramite il tool *wat2wasm* [1]. Attraverso il comando:

```
1 wat2wasm file.wat -o file.wasm
```

Il tool *wat2wasm* prende in input un file .wat e genera un file .wasm. Andando più nel dettaglio di come viene generato il file .wat dal compilatore, vediamo che per ogni nodo dell'ast che abbiamo parlato nei capitoli precedenti viene creata un'ulteriore funzione “codeGeneration()” che ritorna una stringa. Ricorsivamente andremo a chiamare la funzione “codeGeneration()” per ogni nodo dell'ast che ritorna una stringa che mano a mano verrà concatenata con la precedente andando a ottenere il codice wat. Andremo a vedere nello specifico due implementazioni della funzione “codeGeneration()” durante la generazione del codice wat, la *codeGeneration* per l'if Node e la *codeGeneration* per il for Node.

```
1 @Override
2 public String codeGeneration() {
```

```

3         return "(local $res i32)\n" +
4             "(if"+exp.codeGeneration()+
5             "(then\n"+stmT.codeGeneration()+
6             "(local.set $res)" +
7             "\n)" +
8             "(else\n"+stmF.codeGeneration()+
9             "(local.set $res)" +
10            "\n)" +
11            "\n)" +
12            "(local.get $res)\n";
13    }

```

Listing 1.3: codeGeneration() per l'if Node

Descriviamo il funzionamento della codeGeneration per l'if Node. Come vediamo nel listato 1.3 la funzione ritorna una stringa che contiene il codice wat per l'if Node. La prima istruzione *(local \$res i32)* dichiara una variabile locale di nome \$res di tipo i32, che serve da accumulatore per il risultato del ramo then e il risultato del ramo else. La seconda istruzione *if* richiama la funzione codeGeneration() dell'exp Node, che ritorna una stringa che contiene il codice wat per l'exp Node. Viene valutata l'espressione e se il risultato é 1 allora viene eseguito il ramo then, altrimenti viene eseguito il ramo else. La terza istruzione *then* richiama la funzione codeGeneration() del ramo then, che ritorna una stringa che contiene il codice wat per il ramo then. Al termine di quella codeGeneration ci aspettiamo di avere un elemento della pila che contenga il risultato di quella espressione, con il local.set \$res andiamo a salvare il risultato nella variabile locale \$res, e togliendolo da quella pila, mantenendo così l'invariante. La quarta istruzione *else* richiama la funzione codeGeneration() del ramo else, che ritorna una stringa che contiene il codice wat per il ramo else, in maniera simmetrica a ciò che abbiamo fatto per il ramo then. La quinta istruzione *local.get* prende il valore della variabile locale \$res e lo inserisce nello stack, e lo ritorna.

Andremo di seguito a vedere lo stesso ragionamento per la codeGeneration del for Node:

```

1    @Override
2    public String codeGeneration() {
3        return "(local $" + id + " i32)\n" +
4            "(local $" + id + "_max i32)\n" +
5            "(local $res i32)\n" +
6            exp.codeGeneration() +
7            "(local.set $" + id + "_max)\n" +
8            "(loop $for"+line+"\n" +
9            "(if (i32.lt_u (local.get $" + id + ") (local.get
    $" + id + "_max) )\n"+

```

```

10         "(then"
11         + stm.codeGeneration()
12         +"(local.set $res)\n"
13         +"(local.get $" + id + "\n)" +
14         "(i32.const 1)\n" +
15         "(i32.add)\n" +
16         "(local.set $" + id + "\n)" +
17         "(br $for" + line + ")\n)" +
18         "(else\n" +
19         "(local.get $" + id + "_max)\n" +
20         "(local.set $" + id + "))\n)\n" +
21         "(local.get $res)" ;
22     }
23 }

```

Listing 1.4: codeGeneration() per il for Node

La prima istruzione (*local \$id i32*) dichiara una variabile locale di nome \$id di tipo i32, che servirà da iteratore, e la seconda istruzione (*local \$id_max i32*) dichiara una variabile locale di nome \$id_max di tipo i32, che serve da limite superiore per l'iteratore, infine viene dichiarata una variabile \$res per memorizzare il risultato del corpo del ciclo, in caso di ritorno. La terza istruzione *exp.codeGeneration()* richiama la funzione codeGeneration() dell'exp Node, che ritorna una stringa che contiene il codice wat per l'exp Node. Questo valore appena valutato, verrà salvato nella variabile locale \$id_max, con la quarta istruzione (*local.set \$id_max*). La quinta istruzione (*local.get \$id_max*) prende il valore della variabile locale \$id_max e lo inserisce nello stack, e lo ritorna. successivamente viene eseguito un loop, che viene eseguito finché il valore della variabile locale \$id_max è minore del valore della variabile locale \$id. Questo è reso possibile attraverso la definizione della label (*loop \$for+line*) che ci permette di definire l'inizio del loop. L'istruzione (*if (i32.lt_u (local.get \$id_max) (local.get \$id))*) prende i due valori \$id_max e \$id e li confronta, se il secondo è minore del primo allora esegue il ramo then, eseguendo il corpo del ciclo, altrimenti esce dal loop e passa al nodo successivo. Dentro il ramo then viene eseguito il corpo del ciclo, richiamando la funzione codeGeneration() del corpo del ciclo, che ritorna una stringa che contiene il codice wat per il corpo del ciclo. Inoltre verrà preso il contatore \$id, verrà incrementato di 1, e verrà salvato nella variabile locale \$id, con l'istruzione (*local.set \$id*) e salta alla label definita in precedenza (*br \$for+line*).

1.4 Esecuzione del modulo WebAssembly

Una volta generato il file `.wasm`, possiamo eseguirlo attraverso un motore di runtime. Nel nostro specifico caso di debug utilizziamo un file html che contiene il seguente codice javascript:

```
1   const memory = new WebAssembly.Memory({ initial: 1 });
2
3   const importObject = {
4     js: { mem: memory }
5   };
6
7   fetch("output.wasm")
8     .then(response => response.arrayBuffer())
9     .then(bytes => WebAssembly.instantiate(bytes,
10    importObject))
11     .then(obj => {
12       console.log(obj.instance.exports.main(10));
13     })
14     .catch(error => console.error(error));
```

Listing 1.5: Esecuzione del modulo WebAssembly

La prima istruzione `const memory = new WebAssembly.Memory(initial: 1);` definisce un oggetto `memory` che rappresenta la memoria del modulo WebAssembly. In questo caso, la memoria è inizializzata con una dimensione iniziale di 1 pagina. La seconda istruzione `const importObject = { js: { mem: memory } }` definisce un oggetto di importazione che contiene la memoria del modulo WebAssembly. Questo oggetto di importazione verrà utilizzato per fornire l'accesso alla memoria del modulo WebAssembly. La terza istruzione `fetch("output.wasm")` carica il file `.wasm` e restituisce una Promise che contiene i byte del file `.wasm`. Andando nella console di chrome é possibile vedere il file `.wasm` caricato, e debuggarlo come vediamo in questo snippet:

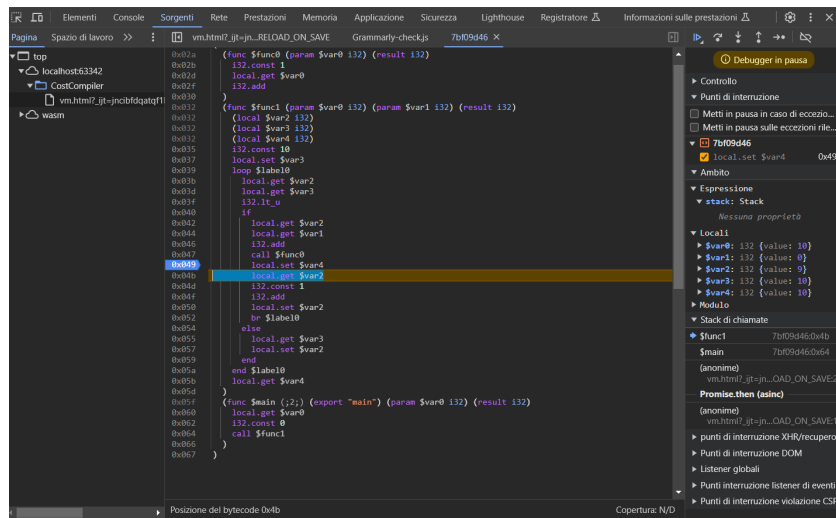


Figura 1.1: Debug del file output.wasm

Come notiamo il file .wasm é stato caricato correttamente, e possiamo andare a vedere il suo contenuto, inserire breakpoint e andare ad analizzare il runtime del nostro codice. Notiamo inoltre che il formato WAT a confronto é molto piú leggibile, soprattutto per quanto riguarda il nome delle variabili, Wasm contiene il programma mappando le variabili in var_0, \dots, var_n . A destra troviamo sia lo stack, mentre altre di sotto le variabili che interagiscono in un determinato scope locale, e lo stack di chiamate delle funzioni.