Capitolo 1

Generazione di WebAssembly

La ricerca di soluzioni efficienti e portabili per eseguire codice in ambienti diversi è diventata una priorità fondamentale. 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, esploreremo 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) [2] è un formato di istruzioni portabile, sicuro e ad alte prestazioni, progettato per essere eseguito in ambienti virtuali. Il formato

è stato sviluppato da un gruppo di lavoro congiunto tra Google, Mozilla, Microsoft e Apple, con l'obiettivo di fornire un formato binario sicuro, veloce e indipendente dalla piattaforma. Il formato è stato progettato per essere eseguito in ambienti virtuali, come browser web, ma può essere utilizzato anche in altri contesti, come ad esempio server, dispositivi IoT e applicazioni desktop. Le istruzioni Wasm si differiscono dalle istruzioni di un processore reale, in quanto sono progettate per essere eseguite in un ambiente virtuale. Questo significa che le istruzioni Wasm non sono direttamente eseguibili da un processore fisico, ma devono essere prima tradotte in istruzioni native. Questo processo di traduzione è gestito da un motore di runtime, che si occupa di interpretare le istruzioni Wasm e di tradurle in istruzioni native. Il motore di runtime è responsabile anche di gestire la memoria e le risorse del sistema, fornendo un'astrazione sicura e indipendente dalla piattaforma.

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 potrá 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.

```
(func $fun (param i32) (param i32) (result f64)
    (local $par1 i32)
    (local $par2 i32)
    (local $par3 f64)
    ...
    (local.get $par1)
    (local.get $par2)
    (local.set $par3)
    ...
)
```

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:

```
(func $somma(param $p1 i32)(param $p2 i32)
    (result i32)
    local.get $p1
    local.get $p2
    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:

```
(func $main
    (result i32)
    i32.const 10
    i32.const 5
    call $somma)
```

La prima istruzione *i32.const* inserisce il valore constante 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).

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

La prima stringa "main" é il nome della funzione che vogliamo esportare e che sara 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 andrá poi convertito in un file .wasm, che a sua volta potrá essere eseguito da un motore di runtime. Questa conversione viene fatta tramite il tool wat2wasm [1]. Attraverso il comando:

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

Il tool wat2wasm prende in input un file .wat e genera un file .wasm. Andando piú 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 verrá 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.

```
    @Override
    public String codeGeneration() {
        return "(local $res i32)\n" +
                "(if"+exp.codeGeneration()+
                "(then\n"+stmT.codeGeneration()+
                "(local.set $res)" +
                "\n)" +
                "(else\n"+stmF.codeGeneration()+
                "(local.set $res)" +
                "\n)" +
                "\n)" +
                "(local.get $res)\n";
```

```
13      }
```

Listing 1.3: codeGeneration() per l'if Node

Andiamo a descrivere 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 servirá 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. Verrá valutata l'espressione e se il risultato é 1 allora verrá eseguito il ramo then, altrimenti verrá 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 cosi 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 ció 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                   exp.codeGeneration() +
6                   "(local.set $"+id+"_max)\n" +
7                   "(local.get $"+id+"_max)\n" +
8                   "(loop $for"+line+"\n"+
9                   "(if (i32.lt_u (local.get $"+id+"_max) (local
    .get $"+id+"))\n"+
10                  "(then"
11                  + stm.codeGeneration()
12                  +"(local.get $"+id+"\n)" +
13                  "(i32.const 1)\n" +
14                  "(i32.add)\n" +
15                  "(local.set $"+id+")\n" +
16                  "(br $for"+line +")\n)"
17                  + "(else\n" +
18                  "(local.get $"+id+"_max)\n" +
19                  "(local.set $"+id+"))\n))" ;
```

```
20      }
```

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 servirá da iteratore, e la seconda istruzione *(local $id_max i32)* dichiara una variabile locale di nome $id_max di tipo i32, che servirá da limite superiore per l'iteratore. 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, verrá 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 é minore del valore della variabile locale $id. Questo é 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 é 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 verra preso il contatore $id, verrá incrementato di 1, e verrá salvato nella variabile locale $id, con l'istruzione *(local.set $id)* e salta alla label definita in precedenza*(br $for+line)*.