Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica (Classe L-31)

Generatore di programmi WebAssembly

Studenti
Kacper H. Osicki, Hamza Brahmi, Lorenzo Toscano

Professore
Rosario Culmone

Matricole 109598,109429,110204

Supervisore
Mattia Paccamiccio

A.A. 23/24

Indice

Elenco dei codici

1. Introduzione

Il presente progetto si propone di implementare un generatore di programmi Webassembly che siano adatti ad essere facilmente risolvibili in termini di ground truth, adatti a testare vari edge cases di tools per analisi statica.
La struttura del progetto è concepita in modo estensibile, consentendo futuri sviluppi e iterazioni. In particolare, si mira a garantire che i programmi generati mantengano una flessibilità significativa, al fine di adattarsi e affrontare sfide emergenti nel contesto dell'analisi statica. Per lo scopo non è necessaria una particolare complessità a livello di funzionalità, è sufficiente che implementino vari tipi di indirezione di memoria.

1.1 Obiettivi

Il progetto si presta ad essere estendibile e i programmi generati hanno una struttura quanto meno rigida possibile. L'obiettivo è promuovere una flessibilità che si adatti alle esigenze specifiche dei test di analisi statica.
Gli obiettivi posti dal progetto:

1. Rappresentare le istruzioni del linguaggio;
2. direttive per generare una sintassi valida;
3. generatore di funzioni (e del programma) che utilizzi i due WP precedenti;
4. Sviluppare una Classe che rappresenti un callgraph.

1.2 Organizzazione del lavoro

Questo progetto è stato sviluppato attraverso l'implementazione di una metodologia agile. Questo approccio dinamico e collaborativo favorisce la flessibilità nelle fasi di pianificazione, esecuzione e valutazione, consentendo un adattamento rapido alle varie esigenze del progetto.
L'adozione di queste pratiche agili, facilitano la comunicazione in maniera efficace tra i membri del team, promuovendo così una gestione efficiente delle risorse e un progresso costante nel tempo con il raggiungimento degli obiettivi prefissati.
Le varie fasi dello sviluppo del progetto sono state:

1. Studio del linguaggio WebAssembly e della sua architettura a pila;
2. Studio dell'utilità di un callgraph, nel contesto dell'analisi software e delle diverse precisioni dei callgraph, che possono essere generati automaticamente o meno;
3. Revisione ed analisi di strumenti simili al progetto in questione esistente;

4. Implementazione delle istruzioni lineari e di control flow;
5. Implementazione delle direttive per generare una sintassi valida e testing;
6. Implementazione di un generatore di call graph che emuli una semplice analisi semantica che potrebbe fare un compilatore;

1.3 Tecnologie usate

Il Progetto è stato implementato utilizzando JavaScript, la scelta ricade per la sua flessibilità e capacità di sviluppo in OOP.
Sono stati utilizzati tool di verifica di webAssembly come, wat2wasm con il suo relativo simulatore online [Weba] e Graphviz online[Dre] per visualizzare i grafi delle chiamate generati e le documentazioni di webAssembly [Webb] [Webc] e GraphViz [Lab].

2. WebAssembly

Con l'aumentare della complessità delle applicazioni web e embedded, la scalabilità e la necessità di eseguire operazioni complesse in tempi brevi, sono diventate imperativi, spingendo gli sviluppatori a esplorare soluzioni avanzate per superare le limitazioni di JavaScript.
Pur essendo il linguaggio predominante nello sviluppo web, JavaScript ha dimostrato limitazioni significative, specialmente in termini di prestazioni.

JavaScript è un linguaggio di programmazione interpretato di alto livello eseguito direttamente nel browser. Nei linguaggi interpretati, non è necessario attraversare una fase preliminare di compilazione del codice. Una volta letto il codice, viene tradotto in codice macchina ed eseguito direttamente dal browser. Questa caratteristica offre un notevole vantaggio nello sviluppo web e sistemi embedded, consentendo modifiche immediate al codice senza la necessità di un processo di compilazione. D'altro canto, l'interprete dovrà convertire le singole istruzioni in codice macchina ogni volta che il codice verrà eseguito, gravando sulle prestazioni.

2.1 Il WebAssembly Text Format

Il WebAssembly, di per sé, è un linguaggio Assembly-like, che offre prestazioni simile a codice nativo. È progettato per essere un efficace target di compilazione per linguaggi come (C, C++, Rust, e altri...)
In generale supporta tutti i linguaggi compatibili con LLVM. Questo permette allo sviluppatore di scrivere il codice nel linguaggio che preferisce per poi compilarlo in WebAssembly.
Sostanzialmente, WebAssembly vuole fornire un'interfaccia portabile fra più piattaforme, che sia ad alte prestazioni e interoperabile con altri linguaggi. Il caso più comune è quello di Javascript, che essendo interpretato, non fa delle prestazioni un suo cavallo di battaglia.

2.2 La Stack Machine

Una **Stack Machine** (o macchina a pila) è un tipo di architettura di computer in cui le istruzioni vengono eseguite manipolando uno stack, ossia una struttura dati di tipo **Last-In-First-Out**(LIFO). In una macchina a pila, il "top" dello stack rappresenta la posizione corrente in cui vengono effettuate le operazioni.

Nella Stack Machine, le operazioni di manipolazione dello stack comprendono l'inserimento di dati (**push**) e la rimozione di dati (**pop**). Le istruzioni del programma possono operare direttamente sugli elementi in cima allo stack, senza la necessità di specificare operandi espliciti. Le operazioni tipiche includono l'aggiunta di elementi in cima allo stack, la loro sottrazione, moltiplicazione, divisione e altre operazioni aritmetiche.

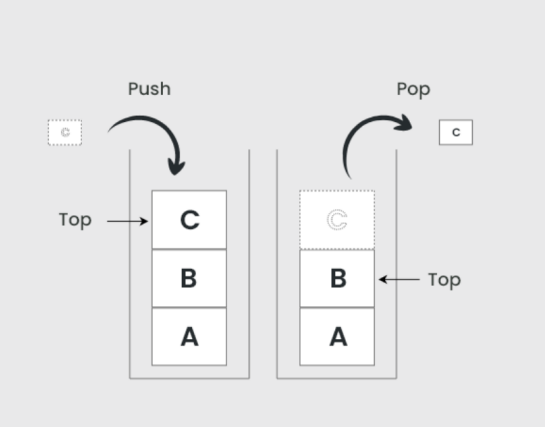

Figura 2.1: Stack Machine

Le Stack Machine sono spesso utilizzate nell'implementazione di linguaggi di programmazione interpretati o compilati in bytecode. L'uso di uno stack semplifica l'implementazione dell'interprete o del compilatore, in quanto riduce la necessità di utilizzare registri espliciti per le operazioni aritmetiche.

2.3 Il formato WebAssembly Text

Il formato binario di WebAssembly (WASM), ottimizzato per l'esecuzione veloce nei browser, è progettato per massimizzare l'efficienza e la compattezza. Tuttavia, questa ottimizzazione può rendere il codice più complesso da comprendere.
Per facilitare il processo di sviluppo e debugging da parte degli sviluppatori, è stato introdotto il **WebAssembly Text Format (WAT)**, una rappresentazione testuale del codice WebAssembly, che lo rende più accessibile durante le fasi di lavoro. Strutturato in modo simile a un linguaggio di assembly, questo formato semplifica la comprensione della logica del programma, agevolando l'analisi e la modifica del codice.
Prendiamo come esempio 2.1:

Codice 2.1: Esempio di codice Wat Somma di due valori

```
(module
  (func $add (param $a i32) (param $b i32) (result i32)
    get_local $a
    get_local $b
    i32.add)
  (export "add" (func $add)))
```

2.3.1 Elementi fondamentali del Wat

Nel wat si possono identificare diversi elementi fondamentali:

- **Module**: l'unità fondamentale del webassembly. Viene rappresentato da un S-expression. Le S-expression sono un formato di testo per rappresentare un albero; perciò, possiamo pensare a un modulo come un albero di nodi che descrivono la struttura del suo codice. Ogni nodo è racchiuso tra due parentesi tonde. Questa rappresentazione semplificata dell'albero consiste in una lista di istruzioni. (module (memory 1) (func)) Questo esempio rappresenta un semplice albero con "module" come radice e due nodi figli: "memoria", con l'attributo "1", e un nodo "func".

- **Le funzioni**: il codice webassembly è composto da funzioni con la seguente struttura: **(func *signature locals body*)** Dove la signature, o firma, dichiara il tipo e il numero dei parametri e il tipo di ritorno. Se la funzione non ritorna nulla, **è possibile omettere** il "(result)". I locals, sono variabili per cui, a differenza di Javascript, è associato un tipo esplicito. Il body, è il corpo della funziona formata da una serie di istruzioni di basso livello.

- **I locals**, dopo la firma vengono elencati le variabili locali con il loro, ad esempio (local i32). Ad esempio se la funzione è :

  Codice 2.2: Esempio di codice Wat local get indicizzati

  ```
  (func (param i32) (param f32)
      local.get 0
      local.get 1
  )
  ```

  local.get 0 assumerà il valore del primo parametro di tipo i32; local.get 1 assumerà il valore del secondo parametro di tipo f32. Inoltre, per evitare di utilizzare gli indici numerici per identificare i parametri, è possibile dargli un nome tramite il prefisso '$', ad esempio:

  Codice 2.3: Esempio di codice Wat local get e la keyword $

  ```
  (func (param $p1 i32) (param $p2 f32)
      local.get $p1
      local.get $p2
  )
  ```

2.3.2 Lo Stack

L'esecuzione di WebAssembly è definita in termini di una macchina a **stack**, in cui ogni istruzione comporta l'aggiunta (**Push**) e/o la rimozione (**POP**) di un determinato numero di elementi dalla pila, i quali potranno avere i tipi **i32, i64, f32 e f64.**

Durante la chiamata di una funzione, **inizia con uno stack vuoto** che viene gradualmente riempito e svuotato man mano che le istruzioni contenute nel body vengono eseguite.
Ad esempio:

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

Questa funzione prende due parametri i32, li somma e ne restituisce il risultato. local.get spinge il valore del local letto nello stack. i32.add consuma i due valori i32 in cima allo stack, li somma e aggiunge il valore della loro somma. Al termine dell'esecuzione, all'interno dello stack, ci sarà solo un valore i32, ovvero la somma dei due valori. Avere la corretta conformità dello stack è un parametro di conformità del programma, WebAssembly non permette errori di runtime a riguardo, il risultato della funzione deve necessariamente corrispondere allo stato dello stack e di conseguenza, se la funzione non ha tipo di ritorno, anche lo stack dovrà essere vuoto.

2.3.3 La Table

In WebAssembly, le tabelle sono vettori di riferimenti ridimensionabili a cui è possibile accedere tramite indice. Quando si chiama una funzione attraverso la keyword 'call', si passa un indice di funzione statico che fa riferimento ad una specifica funzione. Per poter rendere le chiamate dinamiche, quel valore deve essere assegnato a runtime. Per questo si utilizza la keyword 'call_indirect'.
Ad esempio:

```
(module
    (table 2 funcref)
    (elem (i32.const 0) $f1 $f2)
        (func $f1 (result i32)
            i32.const 42
        )
        (func $f2 (result i32)
        i32.const 13
        )
  ...
)
```

In **(table 2 funcref)**, Il numero 2 indica la dimensione iniziale della tabella, quindi il numero di riferimenti da memorizzare.
Mentre **funcref** rappresenta il tipo dei riferimenti a funzione.
La sezione **(elem)** rappresenta l'elenco delle funzioni a cui fa riferimento la tabella nell'ordine in cui devono essere referenziate, con il quale inizializzarla.
Le funzioni **(func)** sono l'elenco delle funzioni di cui la tabella farà riferimento.

L'ordine in questa sezione, a differenza della sezione *elem*, non ha importanza. Il valore (i32.const 0) all'interno della sezione elem **è un offset** che specifica da quale indice della tabella iniziano i riferimenti a funzione. Nell'esempio precedente, abbiamo l'offset a 0 e la dimensione della tabella pari a 2, quindi possiamo inserire i riferimenti negli indici 0 e 1.

Una volta definita la tabella è possibile utilizzarla in questo modo:

```
(type $return_i32 (func (result i32)))
(func (export "callByIndex") (param $i i32) (result i32)
    local.get $i
    call_indirect (type $return_i32)
)
```

Il blocco **(type $return_i32 (func (result i32)))** specifica un type con un nome di rifermento. Questo tipo viene utilizzato durante la fase di **type-checking** delle chiamate di riferimento delle funzioni nella tabella. In questo caso i riferimenti devono essere funzioni che restituiscono come risultato un i32. Successivamente, viene definita una funzione che verrà esportata con il nome di "callByIndex". Questa richiederà un parametro i32, a cui verrà assegnato il nome $i. La funzione, aggiunge il valore di $i nello stack. Infine, si utilizza call_indirect per chiamare una funzione dalla tabella, la quale estrarrà implicitamente il valore di $i dallo stack. La funzione chiamata "*callByIndex*" chiamerà la i-esima funzione della tabella.

2.4 Esempi di utilizzo del linguaggio

WebAssembly ha rapidamente guadagnato terreno nel mondo del web develpoment e dei sistemi embedded, emergendo come una tecnologia fondamentale per migliorare le prestazioni delle applicazioni. Un esempio tangibile di questo successo è evidente nell'ampia adozione di WebAssembly da parte di importanti attori dell'industria, come dimostra il caso di Amazon Prime Video.

2.4.1 Il caso Amazon Prime Video

In questo articolo di Amazon Science, viene evidenziato come l'adozione di **WebAssembly** *(Wasm)* abbia contribuito significativamente al miglioramento delle prestazioni dell'applicazione Prime Video su una vasta gamma di oltre 8.000 dispositivi. Utilizzando WebAssembly in combinazione con Rust, Amazon è riuscita a ottimizzare il frame rate, la stabilità e l'efficienza della gestione delle risorse dell'app. Il passaggio a Wasm ha consentito di ridurre i tempi di frame medi e peggiori su dispositivi come le TV, migliorando l'esperienza utente complessiva. L'articolo sottolinea anche come l'uso di Rust abbia semplificato il processo di sviluppo, migliorando la manutenibilità del codice e permettendo di risparmiare risorse di memoria significative [Ene].
La scelta di utilizzare WebAssembly e Rust ha dimostrato di essere un investimento proficuo per Amazon Prime Video, portando a miglioramenti sostanziali nelle prestazioni e nell'efficienza dell'applicazione.

3. Analisi Statica

Il controllo o analisi statica, è una tecnica che consente di analizzare il codice di un programma attraverso un processo di valutazione basato sulla struttura, sulla forma e sul contenuto. Non richiede dunque l'esecuzione del programma che si sta esaminando. Linguaggi diversi presentano problematiche differenti.
Si concentra su diversi aspetti:

- Flusso di controllo: si analizzano le sequenze di esecuzione possibili al fine di accertare che il codice sia ben strutturato e localizzare blocchi di codice raggiungibile;
- Flusso dei dati: si analizza l'uso e l'evoluzione di variabili e costanti, rileva anomalie quali l'utilizzo di variabili non inizializzate o la presenza di scritture successive senza letture intermedie;
- Esecuzione simbolica: è un metodo di analisi in cui si assume di avere degli input simbolici e si procede con l'esecuzione del programma in cui non vengono elaborati valori ma formule, ottenendo un execution tree che contiene le informazioni su tutti i possibili esiti;
- Verifica formale del codice: la prova della correttezza del codice rispetto alla specifica con l'ausilio di strumenti matematici, per avere una prova di correttezza totale bisogna provare anche la terminazione;

È possibile eseguire una verifica statica usando strumenti automatici, che permettono di ridurre tempi e costi di testing.

3.1 CallGraph

Un grafo delle chiamate(**CallGraph**) è un grafico del flusso di controllo, che rappresenta le relazioni di chiamata tra subroutine in un programma per computer. Ogni nodo rappresenta una procedura e ogni arco indica che la procedura (f) chiami la procedura (g). Pertanto, un ciclo nel grafico, indica chiamate di procedura ricorsive.
Lo studio dell'utilità di un Callgraph nel contesto dell'analisi del software, riveste un ruolo fondamentale nell'evidenziare le relazioni tra le diverse funzioni e procedure presenti nel codice sorgente di un programma. La generazione automatica di callgraph può variare in termini di precisione, e tale variazione ha implicazioni significative nell'efficacia dell'analisi software.
Il callgraph, rappresentando le dipendenze tra le chiamate di funzioni, fornisce un quadro completo della struttura del software, agevolando l'identificazione di flussi di

esecuzione e la comprensione delle interazioni tra componenti. La precisione del callgraph è cruciale per garantire risultati affidabili nelle analisi, e questa precisione può variare a seconda del metodo di generazione adottato.

3.2 L'utilità di un CallGraph

I grafi delle chiamate, sono strumenti essenziali nell'analisi del software poiché forniscono una rappresentazione visuale delle relazioni tra le diverse funzioni o procedure presenti nel codice sorgente di un programma.
La principale utilità dei callgraph include:

- **Comprensione della Struttura del Software**:
  I callgraph consentono di ottenere una visione completa e chiara della struttura del software, mostrando le dipendenze e le interazioni tra le diverse parti del codice. Questo aiuta gli sviluppatori e gli analisti, a comprendere come le funzioni sono collegate e come il flusso di esecuzione attraversa il programma.

- **Analisi dei Flussi di Esecuzione**:
  Attraverso il callgraph, è possibile identificare i diversi flussi di esecuzione all'interno di un programma. Questo è particolarmente utile per individuare percorsi critici, analizzare il comportamento del software in scenari specifici e ottimizzare le prestazioni.

4. Implementazione

Questo capitolo offre una panoramica approfondita del nostro generatore di programmi WebAssembly (Wasm). Il nostro obiettivo principale è la creazione dinamica di moduli Wasm, composti da un insieme di funzioni generate in modo casuale, con il rispettivo grafico delle chiamate.

Durante il processo di implementazione, abbiamo posto una particolare attenzione sull'architettura modulare, per garantire flessibilità ed estensibilità al sistema.
Ogni aspetto, dalla generazione casuale del corpo delle funzioni alla creazione del grafo delle chiamate, è stato affrontato con cura al fine di assicurare coerenza sintattica e un'ampia diversità nei programmi generati.

Questo capitolo fornisce una guida dettagliata attraverso le componenti chiave del nostro software e illustra le decisioni implementative fondamentali che ne guidano il funzionamento.

4.1 Architettura del Progetto

Il progetto è organizzato in modo da garantire una modularità ed estensione delle varie parti che costituiscono la logica, alla base della generazione del codice e del suo rispettivo grafo delle chiamate. Così da facilitare la comprensione, l'aggiunta di nuove funzionalità e la manutenzione del codice.
Le classi principali del progetto seguono uno schema di progettazione basato sulla divisione delle varie responsabilità che sono emerse durante la fase iniziale dello sviluppo. Ogni classe svolge un ruolo cruciale nel processo complessivo di generazione, contribuendo al fine di garantire una coerenza strutturale e funzionale.
Esploreremo in dettaglio il ruolo e le responsabilità di ciascuna classe, delineando come ognuna contribuisca in modo unico all'obiettivo globale di generazione dinamica del codice WebAssembly nel formato .wat, e del suo rispettivo grafico delle chiamate nel formato .dot.
Le principali classi sono:

- **WasmGenerator:**
  Responsabile della generazione del codice WebAssembly, la classe 'WasmGenerator', guida l'intero processo di creazione dinamica del codice. Gestisce la logica di generazione casuale delle funzioni, la costruzione del modulo WebAssembly, e la personalizzazione dei parametri chiave del generatore, come la probabilità di chiamate e istruzioni di controllo del flusso.

- **Module:**
  La classe 'Module' rappresenta il modulo del file WebAssembly text (wat). Questa classe si occupa della struttura generale del modulo, inclusi i tipi di funzioni, le

tabelle delle funzioni, l'esportazione delle funzioni, e la scrittura del modulo su un file .wat.

- **Instruction:**
  La classe 'Instruction' è la classe base per le istruzioni, fornendo una struttura comune per tutte le operazioni possibili. Essa potrà essere estesa da classi specifiche. Ogni istruzione ha tre attributi: il nome, quanti elementi consuma dallo stack e quanti ne produce. Questo poi permetterà al programma di poter scegliere, in base allo stato attuale dello stack, quali istruzioni poter generare.

- **CallInstruction:**
  Specializzazione di 'Instruction', la classe 'CallInstruction' gestisce le istruzioni call e call_indirect, permettendo la corretta generazione di chiamate di funzioni durante la creazione del corpo delle funzioni WebAssembly.

- **Instructions:**
  La classe `Instructions` rappresenta una collezione di istruzioni di flusso lineare. Essa offre un modo strutturato per organizzare e manipolare un insieme di istruzioni sequenziali, semplificando la gestione di flusso nel corpo delle funzioni.

- **ControlFlowInstructions:**
  La classe `ControlFlowInstructions` rappresenta una collezione di istruzioni di flusso di controllo, come "if", "loop" e "return". Essa è utilizzata per gestire la generazione di istruzioni che controllano il flusso di esecuzione del programma WebAssembly.

- **Stack:**
  La classe 'Stack' è responsabile della rappresentazione dello stack del programma WebAssembly. Gestisce lo stato corrente dello stack, garantendo che le operazioni siano eseguite in modo coerente rispetto alla struttura dello stack durante la generazione del codice WebAssembly.

- **GraphManager:**
  La classe 'GraphManager' rappresenta un gestore del grafo delle chiamate. Questa classe gestisce la creazione e la manipolazione del grafo, consentendo l'aggiunta di nodi e la definizione di archi tra di essi. È responsabile di generare i dati necessari per creare una rappresentazione grafica del grafo delle chiamate attraverso il formato DOT.

4.2 Logica dietro la Generazione Casuale delle Funzioni

Una volta definita la struttura del progetto, è il momento di illustrate le varie fasi coinvolte nella generazione casuale del modulo WebAssembly.

Questo processo può essere suddiviso in tre parti fondamentali: la generazione del modulo, la generazione delle funzioni e la generazione delle singole istruzioni. Ognuna di queste sottosezioni, costituiscono un tassello cruciale nella creazione casuale di codice WebAssembly.

In particolare, affrontiamo la gestione dello stack durante l'esecuzione delle funzioni, i dettagli sulla selezione delle istruzioni, la gestione delle probabilità e altri aspetti collegati alla casualità della generazione. Discuteremo anche della possibilità di personalizzare il generatore per adattarsi alle varie esigenze specifiche. Inoltre, forniremo

esempi concreti di codice WebAssembly generato casualmente, con il suo rispettivo grafico delle chiamate.

4.2.1 Generazione del Modulo

La generazione del modulo costituisce la fase iniziale del processo di creazione del codice WebAssembly, rivestendo un ruolo cruciale nella definizione della struttura complessiva. La scrittura effettiva del modulo viene delegata alla classe Module, la quale rappresenta il modulo.

Le principali tappe di questo processo includono:

1. **Generazione di Tipi di Funzioni Casuali:**
   Il metodo `generateFunctionTypes` della classe `WasmGenerator` è responsabile della creazione e memorizzazione di tipi di funzioni unici per un modulo WebAssembly.

   Ogni tipo di funzione è composto da un insieme di tipi di parametri e risultati. Il numero di parametri e risultati, così come i loro tipi, sono generati casualmente entro limiti specificati.

   Il concetto di "tipo di funzione" in WebAssembly si riferisce alla firma di una funzione, che include il numero e i tipi di parametri che accetta e il numero e i tipi di risultati che restituisce. Questi tipi sono fondamentali per la corretta interpretazione e esecuzione delle funzioni all'interno di un modulo WebAssembly.

   Il metodo garantisce che nessun due tipi di funzioni siano identici. Se un tipo di funzione appena generato non è duplicato rispetto a uno già esistente, viene aggiunto al modulo e ne viene memorizzato l'indice.

2. **Assegnazione di Tipi alle Funzioni:**
   Successivamente, i tipi di funzioni generati vengono assegnati casualmente alle funzioni generate nel modulo. Questo compito è assegnato al metodo `assignRandomFunctionTyp` nella classe `WasmGenerator`.

   Questo avviene selezionando un tipo di funzione casuale da un elenco di tipi di funzioni precedentemente generati per ogni funzione. La selezione è casuale ma all'interno del range dei tipi di funzioni disponibili.

```
1   assignRandomFunctionTypeToEachFunction() {
2     this.functionTypesByIndex = [];
3     for (let i = 0; i < this.max_number_of_functions; i++) {
4       // Assign a random function type index to the current function
5       this.functionTypesByIndex[i] =
6           this.getRandomInt(0, this.functionTypes.length - 1);
7     }
8   }
```

Codice 4.1: Metodo per l'assegnazione casuale dei tipi alle funzioni

3. **Creazione di una Tabella delle Funzioni:**

   Il metodo `generateFunctionTable` è responsabile della creazione di una tabella WebAssembly che contiene riferimenti alle funzioni (`funcref`).

```
1     generateFunctionTable(max_number_of_functions) {
2       this.watCode += `(table ${max_number_of_functions} funcref)\n`;
3       let elem = "(elem␣(i32.const␣0)";
4       for (let i = 0; i < max_number_of_functions; i++) {
5         elem += ` ${i}`;
```

```
6         }
7         elem += ")\n";
8         this.watCode += elem;
9       }
```

Codice 4.2: Metodo per la Creazione della Tabella delle Funzioni

Questa tabella è un'organizzazione strutturata degli indici delle funzioni disponibili nel modulo e può essere utilizzata per gestire e accedere alle diverse funzioni presenti nel modulo WebAssembly.

4. **Identificazione della Funzione di Avvio:**

   Il metodo `addStartExport` svolge un ruolo chiave nel processo di avvio del modulo WebAssembly. Esso permette di esportare una specifica funzione, indicata da un indice (`export_id`), come "start".

```
1       addStartExport(export_id) {
2          this.watCode += `(export "start" (func ${export_id}))\n`;
3       }
```

Codice 4.3: Metodo per l'Esportazione della Funzione di Avvio

La funzione esportata in questo modo verrà eseguita automaticamente all'avvio del modulo, fornendo un punto di inizio per l'esecuzione del codice WebAssembly.

5. **Aggiunta dei Corpi di Funzione:**
   Il metodo `addFunction` nella classe `Module` è utilizzato per aggiungere una funzione al modulo WebAssembly. Prende l'indice della funzione, l'indice del tipo, i parametri, i risultati e il corpo della funzione come argomenti e costruisce una rappresentazione in stringa della funzione nel formato WebAssembly Text (WAT).

```
1    addFunction(funcIndex, typeIndex, params, result, funcBody) {
2       const paramsString = params.length ? `(param ${params.join(" ")})` : "";
3       const resultString = result.length ? `(result ${result.join(" ")})` : "";
4       const functionString =
5       `(func $${funcIndex} (type ${typeIndex}) ${paramsString} ${resultString}\n
6          ${funcBody})\n`;
7       this.watCode += functionString;
8    }
```

Codice 4.4: Metodo per l'Aggiunta di Funzioni al Modulo

6. **Chiusura del Modulo:**
   Il metodo `closeModule` è utilizzato per finalizzare il modulo WebAssembly. Aggiunge una parentesi di chiusura al codice WAT.

```
1    closeModule() {
2       this.watCode += ")\n";
3    }
```

Codice 4.5: Metodo per la Chiusura del Modulo

7. **Creazione del File .wat:**
   Il metodo `saveToFileWat` salva il codice WAT in un file .wat. Prende il nome del file come argomento e scrive il codice WAT in un file con quel nome nella directory `./output_files`.

```
saveToFileWat(fileName) {
   // Save the watCode to a .wat file
   fs.writeFileSync(`./output_files/${fileName}.wat`, this.watCode);
}
```

Codice 4.6: Metodo per la Creazione del File .wat

4.2.2 Generazione casuale del corpo delle Funzioni

La fase di generazione delle istruzioni all'interno del corpo delle funzioni è un processo intricato che coinvolge la manipolazione dello stack e la scelta dinamica di istruzioni in base allo stato corrente. Il seguente procedimento sottolinea le fasi chiave di questa operazione:

1. **Generazione delle Istruzioni del Corpo della Funzione:**
   Prima di procedere con la generazione delle istruzioni, uno stack viene inizializzato per tenere traccia dello stato corrente durante il processo. Le istruzioni per il corpo della funzione vengono generate dinamicamente sulla base di variabili quali lo stato dello stack e il tipo di funzione.

   Questo processo inizia con la generazione casuale di istruzioni fino al raggiungimento del numero minimo richiesto, o fino all'incontro di un'istruzione di ritorno.

   Un aspetto cruciale di questa generazione, è la considerazione dei concetti di 'consume' e 'produce' delle istanze della classe Instruction.

```
   class Instruction {
   constructor(name, type, consumes, produces) {
     this.name = name;
     this.type = type;
     this.consumes = consumes;
     this.produces = produces;
   }
   ...
}
```

Codice 4.7: Classe Instruction

   In WebAssembly, ogni istruzione può consumare e/o produrre elementi nello stack. Ad esempio, alcune istruzioni aggiungono valori allo stack ('produce'), mentre altre, ne consumano alcuni ('consume'). La corretta manipolazione di queste interazioni è fondamentale per garantire l'integrità dello stack durante la generazione delle istruzioni.

   Durante la scelta delle istruzioni, il metodo `getPossibleInstructions`, tiene conto dello stato attuale dello stack e delle istruzioni disponibili, selezionando quelle che sono compatibili con lo stato corrente. Se lo stack è vuoto, vengono scelte solo le istruzioni che producono almeno un elemento ('produce' 0 e 'consume' = 0). Se lo stack ha elementi, vengono scelte solo le istruzioni che consumano un numero di elementi inferiore o uguale alla lunghezza dello stack ('consume' ¡= stack.length). Le istruzioni risultanti sono poi sottoposte a un processo di filtraggio in base alle probabilità definite.

```
getPossibleInstructions(stackState, instructions){
  let possibleInstructions;
  if (stackState.length === 0) {
    possibleInstructions =
       instructions.filter(instruction =>
```

```
6               instruction.produces > 0 && instruction.consumes === 0);
7       } else {
8         possibleInstructions =
9           instructions.filter(instruction =>
10              instruction.consumes <= stackState.length);
11      }
12      // Apply the probabilities to the possible instructions
13      possibleInstructions = this.applyProbabilities(possibleInstructions);
14      return possibleInstructions;
15    }
```

Codice 4.8: Metodo per la selezione delle possibili istruzioni

Il processo prosegue generando ulteriori istruzioni fino a quando lo stato dello stack contiene un numero di elementi pari al numero di risultati attesi per la funzione, o fino a quando viene scelta un'istruzione di ritorno. In questo caso, la generazione continua a produrre istruzioni corrette, sebbene queste potrebbero essere ignorate in quanto si trovano dopo l'istruzione di ritorno.

2. **Generazione dei Locals:**
   Le variabili locali necessarie per la funzione sono generate in base alla presenza di specifiche istruzioni che ne richiedono, ad esempio local.get e local.set. Durante questo processo, vengono adoperate due principali metodi:

   - `handleLocalInstruction(stackState, instruction, funcType)`: Questo metodo gestisce l'istruzione relativa alle variabili locali. Se non sono presenti parametri né variabili locali, viene aggiunta una nuova variabile locale attraverso il metodo `addLocalVariable()`. Se invece sono presenti parametri o variabili locali, ne viene selezionata casualmente una. L'istruzione relativa viene quindi aggiunta al corpo della funzione, contribuendo alla corretta gestione delle variabili locali durante la generazione delle istruzioni.
   - `addLocalVariable()`: Questo metodo aggiunge una variabile locale al generatore. Assegna un nome univoco alla variabile locale (es. "local0", "local1") e le attribuisce un tipo casuale tra quelli consentiti. La nuova variabile viene quindi inclusa nella lista delle variabili locali, contribuendo alla diversità delle variabili generate.

Infine, il corpo della funzione viene assemblato incorporando sia le istruzioni generate che le variabili locali. Questo costituisce il risultato finale del processo di generazione del corpo della funzione.

Questo approccio alla generazione delle istruzioni e alla gestione dello stack assicura la diversità e la coerenza del codice WebAssembly generato, garantendo il corretto funzionamento delle funzioni generate.

4.2.3 Personalizzazione del Generatore

La personalizzazione del generatore è un aspetto significativo che consente agli sviluppatori di adattare il comportamento della generazione in base alle esigenze specifiche. Gli argomenti chiave da affrontare includono:

1. **Parametri di Personalizzazione:**
   Questo codice definisce un costruttore per un generatore di codice WebAssembly (Wasm). Il generatore è personalizzato impostando vari parametri che controllano le caratteristiche del codice generato.

   I parametri includono:

   - I tipi di valori consentiti nel codice generato (`ALLOWED_TYPES`).
   - Il numero massimo di funzioni, parametri e risultati nel codice generato.
   - Il numero minimo e massimo di istruzioni in ciascuna funzione.
   - Le probabilità di includere diversi tipi di istruzioni (chiamata, chiamata indiretta, if, loop) in ciascuna funzione.
   - Il numero massimo di if annidati e loop in ciascuna funzione.
   - Il numero massimo di iterazioni per ogni loop.

   Questi parametri vengono impostati come proprietà dell'oggetto generatore nel costruttore, i quali verranno poi utilizzati per controllare la randomicità del codice che genera.

```
1       // Personalization of the logic of the generator
2       // Config of the function generator
3       const ALLOWED_TYPES = ["i32"];
4       const MAX_NUMBER_OF_FUNCTIONS_TO_GENERATE = 5;
5       const MAX_NUMBER_OF_PARAMS_FOR_A_FUNCTION = 5;
6       const MAX_NUMBER_OF_RESULTS_FOR_A_FUNCTION = 2;
7
8       // Min and max number of instructions to generate every time
9       const MIN_NUMBER_OF_INSTRUCTIONS = 10;
10      const MAX_NUMBER_OF_INSTRUCTIONS = 50;
11
12      // Config of the probabilities of having those instructions
13      const PROBABILITY_OF_CALL = 1;
14      const PROBABILITY_OF_CALL_INDIRECT = 1;
15      const PROBABILITY_OF_IF = 0.5;
16      const PROBABILITY_OF_LOOP = 0.4;
17
18      // Config of the if instructions
19      const MAX_NESTED_IFS = 1;
20
21      // Config of the loop instructions
22      const MAX_NESTED_LOOPS = 0;
23      const MAX_LOOP_ITERATIONS = 10;
24      }
```

Codice 4.9: Esempi di Parametri di Personalizzazione del Generatore

4.3 GraphManager

La classe `GraphManager` è responsabile della gestione delle relazioni di chiamata tra le funzioni. Durante la generazione delle funzioni casuali, questa classe tiene traccia dei nodi (funzioni) e degli archi (relazioni di chiamata) nel grafo delle chiamate. Ecco un'overview delle funzionalità chiave:

- **Costruzione del Grafo:**
  La classe mantiene una lista di nodi (funzioni) e archi (chiamate di funzioni) attraverso i metodi `addNode` e `addEdge`.

- **Rappresentazione dei Nodi:**
  Utilizzando il metodo `addNode`, è possibile aggiungere un nodo al grafo insieme a una descrizione della funzione associata.
- **Creazione degli Archi:**
  Il metodo `addEdge` consente di definire una relazione di chiamata tra due funzioni, aggiungendo un arco al grafo.
- **Generazione di Dati Dot:**
  Per supportare la visualizzazione del grafo, la classe fornisce il metodo `generateDotData`, che genera dati nel formato Dot per la creazione di grafici.

La struttura modulare di `GraphManager` facilita la gestione e l'estensione del grafo delle chiamate, fornendo una base solida per esplorare le relazioni tra le funzioni generate durante il processo di generazione casuale del modulo WebAssembly.

4.3.1 Generazione del grafo .dot

Dopo aver introdotto la classe `GraphManager`, esploriamo ora come questa classe viene utilizzata all'interno di `WasmGenerator` per generare il grafo delle chiamate. Il processo coinvolge una serie di passaggi che riflettono la dinamica di chiamata delle funzioni durante la generazione del modulo WebAssembly.

- Il metodo `createNode`, è parte integrante della creazione del grafo. Esso genera un nodo nel grafo per rappresentare una funzione. Durante questa fase, la classe `GraphManager` tiene traccia dell'identificatore della funzione, aggiungendo informazioni aggiuntive in base alla sua esportazione o meno.
- Il metodo `generateDotData`, si occupa di collegare i nodi del grafo attraverso archi, riflettendo le relazioni di chiamata tra le funzioni. Utilizzando le informazioni raccolte durante la generazione delle funzioni, vengono create connessioni tra i nodi. In particolare, quando una chiamata di funzione diretta viene gestita, viene aggiunta anche una connessione nel grafo tra il nodo della funzione corrente e il nodo della funzione chiamata. La stessa logica è applicata quando viene gestita una chiamata di funzione indiretta.

La rappresentazione grafica delle relazioni di chiamata tra le funzioni offre una panoramica chiara della struttura generata durante il processo di generazione casuale del modulo WebAssembly. Questa visualizzazione è fondamentale per comprendere le interazioni tra le funzioni generate e fornisce una risorsa visiva utile per l'analisi.

Utilizzo di `compiler_cg` Un aspetto cruciale in entrambi i metodi è l'utilizzo di `compiler_cg`, il compilatore del grafo delle chiamate. Questo componente è responsabile di tracciare le relazioni tra le funzioni chiamate, creando una struttura dati che riflette le dipendenze tra di esse.

Questa rappresentazione grafica delle relazioni tra le funzioni è fondamentale per comprendere la struttura generata e fornisce una risorsa visiva utile per analisi. Inoltre, il grafo delle chiamate contribuisce a una comprensione più approfondita del flusso di esecuzione e delle dipendenze tra le funzioni del modulo WebAssembly generato casualmente.

4.3.2 Scrittura del file .dot

Una volta che il grafo delle chiamate è stato costruito e aggiornato attraverso l'utilizzo di `compiler_cg`, il metodo `generateDotData` del `GraphManager` viene chiamato per generare i dati nel formato Dot necessari per la visualizzazione grafica. Questi dati sono successivamente scritti in un file .dot utilizzando il modulo `fs` in Node.js, creando così una rappresentazione visiva del grafo delle chiamate. Questo file .dot può essere convertito in un formato grafico più elaborato utilizzando strumenti esterni, consentendo una visualizzazione più chiara e comprensibile delle relazioni tra le funzioni.

```
1  /**
2      * Generates dot data for a graph and writes it to dot file.
3      * @param {string} graphName - The name of the graph.
4      */
5     generateDotData(graphName) {
6        this.compiler_cg.forEach((pair) => {
7           const [sourceNode, targetNode] = pair;
8           this.graphManager.addEdge(sourceNode, targetNode);
9        });
10       let dotGraph = this.graphManager.generateDotData(graphName);
11       fs.writeFileSync(`./output_files/${graphName}.dot`, dotGraph);
12    }
```

Codice 4.10: Metodo per la scrittura del file.dot

4.4 Test e Validazione

Testare del codice generato casualmente introduce una sfida significativa nel processo di validazione. Poiché il modulo WebAssembly viene creato in modo casuale, è difficile prevedere esattamente quali istruzioni saranno presenti e in quale sequenza. Ci siamo limitati a scrivere dei test per alcune parti cruciali del software e utilizzare tools esterni per la verifica della correttezza del codice generato.

Il generatore è progettato per fornire una diversità di funzioni, tipi e strutture di controllo del flusso, e la loro combinazione può variare notevolmente da un'esecuzione all'altra. Durante i test, è stato cruciale verificare la conformità del modulo alle specifiche del linguaggio e garantire che le configurazioni di probabilità e limiti vengano rispettate.

Nonostante la complessità dei test in un contesto di generazione casuale, l'utilizzo di strumenti esterni come `wat2wasm` si è dimostrato di essere una strategia efficace per la validazione della correttezza del codice generato.

4.4.1 Tools per la Validazione

Per la validazione del codice generato, è stato utilizzato principalmente il tool `wat2wasm`. Inizialmente, abbiamo considerato l'utilizzo della libreria JavaScript di questo tool per verificare la correttezza del codice generato dal nostro progetto, ma abbiamo riscontrato limitazioni legate alla gestione di funzioni con più di un tipo, motivo per cui abbiamo optato per l'utilizzo della demo online del tool.

Durante i test, i file `.wat` generati sono stati copiati ed eseguiti sulla demo per verificare la correttezza e la conformità del modulo WebAssembly generato.

Questo approccio ha garantito una validazione efficace e ha contribuito a identificare potenziali problemi nel codice generato casualmente.

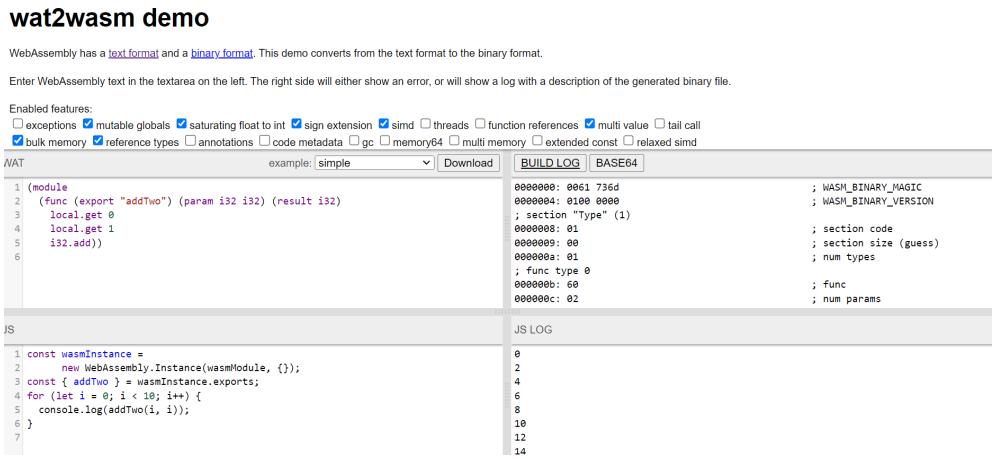

Figura 4.1: Demo online del tool wat2wasm

4.5 Esempio di codice generato dal software e rispettivo callgraph

Nell'esempio di codice generato dal software, il modulo WebAssembly è rappresentato con le sue funzioni e istruzioni. Si noti che questo è solo un frammento di codice e che il software può generare moduli molto più complessi e articolati, mantenendo comunque la sua correttezza.

```
(module
(type (func (result i32 i32)))
(type (func (param i32 i32) (result i32 i32)))
(type (func (param i32) (result i32)))
(table 3 funcref)
(elem (i32.const 0) 0 1 2)
(export "start" (func 0))
(func $0 (type 2) (param i32) (result i32)
i32.const 95
drop
local.get 0
call 2
local.set 0
local.get 0
)
(func $1 (type 2) (param i32) (result i32)
local.get 0
i32.const 23
i32.lt_s
i32.const 0
call_indirect (type 2)
call 1
)
(func $2 (type 2) (param i32) (result i32)
(local $loopCounter0 i32)
local.get 0
(if (result i32 )
(then
local.get 0
drop
local.get 0
drop
local.get 0
)
(else
i32.const 82
call 2
(loop $loop0
```

```
local.get 0
drop
local.get 0
drop
i32.const 68
drop
local.get $loopCounter0
i32.const 1
i32.add
local.set $loopCounter0
local.get $loopCounter0
i32.const 10
i32.lt_s
br_if $loop0
)
i32.const 0
i32.const 2
call_indirect (type 2)
i32.div_u
)
)
call 2
call 1
local.set 0
i32.const 24
)
)
```

Codice 4.11: Esempio di codice generato dal software

Il codice sopra rappresenta un modulo WebAssembly, dove sono presenti tre tipi di funzioni e una tabella di funzioni. L'export "start" punta alla funzione principale ($0) del tipo 2. In questo codice sono presenti diversi tipi di istruzione, come operazioni, if e loop.

Successivamente, è fornito un esempio di file .dot per il grafo delle chiamate associate al codice generato.

```
digraph example
{
 node0 [label="node0" style = "filled" color="gray"]
node1 [label="node1"];
node2 [label="node2"];
node0 -> node2;
node1 -> node0;
node1 -> node1;
node2 -> node2;
node2 -> node1;
}
```

Codice 4.12: Esempio di .dot file del grafo delle chiamate

Il grafo delle chiamate rappresenta le relazioni tra le funzioni del modulo WebAssembly. Ogni nodo corrisponde a una funzione, e gli archi indicano le chiamate tra di esse. Questo file .dot generato è utile in fase di analisi statiche dell'esecuzione del codice e può essere anche modificato all'occorrenza per altri tipi di analisi.

Inoltre è possibile visualizzare questi file graficamente come un vero e proprio grafo attraverso tools appositi.

5. Conclusioni e Sviluppi Futuri

Analizziamo in breve le fasi principali del project group che hanno richiesto maggior tempo per essere affrontate:

1. **Esplorazione del Linguaggio:**
   La prima fase è stata dedicata all'apprendimento del linguaggio, attraverso la consultazione della documentazione e l'esplorazione degli strumenti disponibili.

2. **Sviluppo del Software:**
   La seconda fase, ha visto la realizzazione del software per la generazione di codice .Wat funzionante dal punto di vista sintattico.

3. **Creazione della Libreria di Callgraph:**
   Nella terza fase, abbiamo sviluppato una libreria per la creazione del callgraph, offrendo una visualizzazione delle relazioni tra le funzioni del modulo WebAssembly generato casualmente.

5.1 Sviluppi Futuri

Per quanto riguardano gli sviluppi futuri, il progetto è stato ideato con un'architettura modulare ed estendibile, aprendo la strada a possibili miglioramenti e aggiunte:

- **Esplorazione di Nuove Istruzioni:**
  È possibile arricchire la diversità del codice generato, aggiungendo nuove tipologie di istruzioni.

- **Ampliamento del Grafico delle Chiamate:**
  Sfruttando la modularità ed estendibilità del software, si potrebbero ampliare i dettagli all'interno del grafico delle chiamate, consentendo una comprensione più approfondita delle relazioni tra le funzioni.

Bibliografia

[Dre] Dreampuf. *GraphViz Online*. URL: https://shorturl.at/xLVWY.

[Ene] Alexandru Ene. *How Prime Video updates its app for more than 8,000 device types*. URL: https://www.amazon.science/blog/how-prime-video-updates-its-app-for-more-than-8-000-device-types.

[Lab] ATT Research Labs. *GraphViz*. URL: https://graphviz.org.

[Weba] WebAssembly. *Wat2Wasm online*. URL: https://webassembly.github.io/wabt/demo/wat2wasm/.

[Webb] WebAssembly. *WebAssembly documentation*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly.

[Webc] WebAssembly. *WebAssemblyMath*. URL: https://webassembly.github.io/spec/core/intro/introduction.html,Bdsk-Url-1={https://webassembly.github.io/spec/core/intro/introduction.html}.