ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Triennale

Titolo

Tesi in Tecnologie Web

Relatore
prof. Paolo Bellavista

Laureando
Davide Crociati

Ottobre 2023

A voi

Sommario

Piacere, sommario.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Capitolo 1

Introduzione

1.1 Contestualizzazione

1.1.1 Trend di evoluzione del web

Inizialmente le applicazioni web erano costituite da semplici **pagine statiche** contenenti testo e immagini. Con l'evoluzione dell'ecosistema web,negli ultimi anni si è assistito a una trasformazione radicale delle applicazioni, guidata da una serie di tendenze e innovazioni tecnologiche che hanno portato a un'esperienza utente sempre più coinvolgente e interattiva.

L'introduzione di JavaScript e di librerie e framework correlati, hanno progressivamente arricchito l'interazione con le applicazioni web, introducendo livelli crescenti di interattività e trasformando le semplici pagine statiche in ambienti dinamici e convolgenti. Un cambiamento significativo in tal senso, è avvenuto con l'avvento delle **Single Page Application (SPA)** e di **AJAX**. Questo approccio ha rivoluzionato il tradizionale modello di navigazione e di sviluppo, consentendo alle applicazioni di essere caricate una sola volta e offrendo interazioni rapide e fluide grazie al caricamento dinamico di contenuti. Questo nuovo paradigma ha introdotto un'esperienza utente simile a quella delle applicazioni native, con transizioni scorrevoli tra le sezioni dell'applicazione e senza interruzioni visibili.

Parallelamente, la complessità delle funzionalità offerte è cresciuta in modo esponenziale, arrivando a ciò che viene riconosciuto come Web 3.0 o **Web Semantico**. Ormai è la norma spaziare da applicazioni che usano algoritmi di intelligenza artificiale, a simulatori,

alla modifica istantanea di documenti, immagini e video.
Questo enorme sviluppo di funzionalità è stato trainato dalla crescente potenza di elaborazione dei dispositivi e dalla capacità delle tecnologie web di sfruttare appieno queste risorse, facendo diventare normale interfacciarsi con siti web in grado di gestire complesse operazioni in tempi rapidi.
Tuttavia questo progresso è stato accompagnato da un aumento smisurato del **numero di richieste** effettuate in rete e dall'utilizzo intensivo di risorse computazionali, sia lato cliente, che lato servitore. È diventato cruciale bilanciare l'aggiunta di funzionalità sofisticate con la necessità di mantenere tempi di risposta rapidi e un'esperienza fluida per gli utenti. Inoltre, l'uso intensivo delle risorse ha sollevato questioni di scalabilità e di utilizzo efficiente delle risorse server.

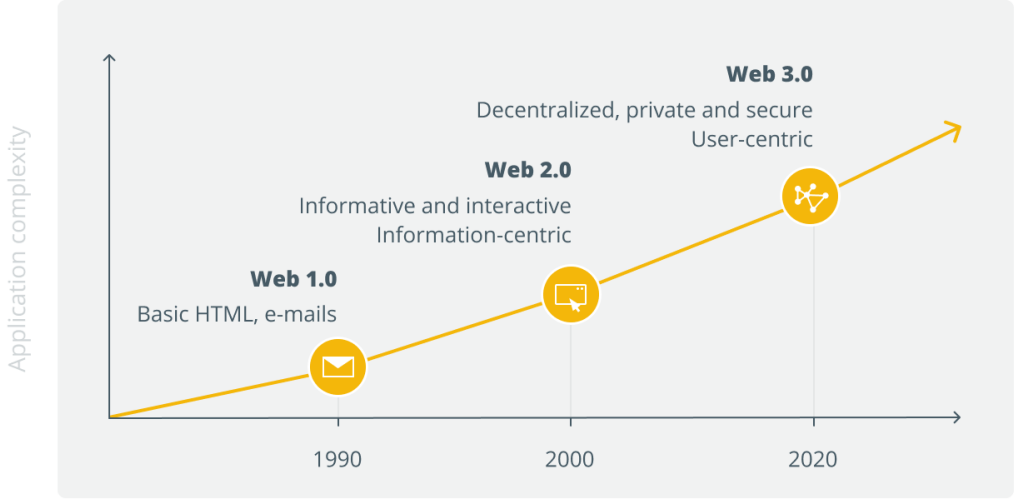

Figura 1.1: L'evoluzione del Web.

1.1.2 Importanza dell'ottimizzazione

Ad oggi, l'ottimizzazione delle prestazioni è quindi diventata un aspetto cruciale nello sviluppo di applicazioni web. Gli utenti si aspettano interazioni con bassa latenza, caricamenti rapidi e risposte immediate. Questa esigenza mette in risalto l'importanza di bilanciare l'aggiunta di nuove funzionalità, con l'offerta di una *User Experience* ottimale.
I tempi di caricamento prolungati possono portare a un alto tasso di abbandono delle

pagine, riducendo l'opportunità di coinvolgere nuovi utenti. Inoltre, con l'aumentare dell'utilizzo di **dispositivi mobili** e di conseguenza, di **connessioni instabili**, l'ottimizzazione diventa ancor più critica per assicurare un'esperienza coerente su diverse piattaforme e condizioni di rete. Tutto ciò non riguarda solo il lato client, ma coinvolge anche il lato server. Un carico eccessivo sui server può influire negativamente sulla scalabilità, causando ritardi nelle risposte e possibili interruzioni del servizio. L'ottimizzazione deve quindi coinvolgere tutti gli aspetti dell'architettura delle applicazioni web.
Nell'implementare ottimizzazioni, sono nati varie soluzioni interessanti. Ad esempio, per gestire task che svolgono molte operazioni di Input/Output si è distinto il runtime environment **Node.js**[1], mentre per quanto riguardo l'esecuzione di attività che sfruttano molto la CPU è emerso **WebAssembly(Wasm)**[2]. Quest'ultimo, nato inizialmente per consentire l'esecuzione di codice ad alta efficienza all'interno del browser, è diventato portabile anche su piattaforme server side grazie allo sviluppo dell'interfaccia di sistema **WebAssembly System Interface (WASI)**[3].

1.2 Motivazioni e Obiettivi

La **crescente complessità** delle applicazioni web e l'esigenza di offrire agli utenti esperienze interattive sempre più coinvolgenti hanno portato l'ambito dello sviluppo web a una svolta significativa. Le aspettative degli utenti si sono evolute verso applicazioni che offrano prestazioni reattive, interattività immediata e funzionalità avanzate.
È proprio questo insieme di aspettative a essere alla base delle motivazioni che hanno guidato la scelta del tema di questa tesi di laurea.
In particolare, la presente ricerca, si propone di confrontare in modo dettagliato, due differenti approcci di sviluppo per un'applicazione con funzionalità **fortemente CPU-Intensive**.

1.2.1 Tipologia di applicazione

Per tale confronto, si è optato per una web-app che implementi alcune tecniche di **elaborazione di immagini**. In particolare, l'utente avrà la possibilità di eseguire l'upload di immagini su un server e indicare una serie di modifiche da apportare (ad esempio "ridimensionamento del 50%"). Il server manipolerà i file in base ai parametri

ricevuti e infine restituirà al cliente le immagini modificate.
Non verrà esplorato in modo approfondito il campo dell'elaborazione digitale di immagini, ma ci si limiterà all'implementazione di funzionalità usate spesso da utenti comuni, come ad esempio, ridimensionamento, rotazione, aumento/diminuzione di luminosità e contrasto e altre che verranno specificate nel capitolo 3.
Tale tipologia di applicazione, si sposa bene per lo scopo finale della tesi: valutare come due aprrocci (e due linguaggi) piuttosto differenti, ma sempre più diffusi al giorno d'oggi, risolvano il problema di un'applicazione web che svolga operazioni dall'alto costo computazionale.

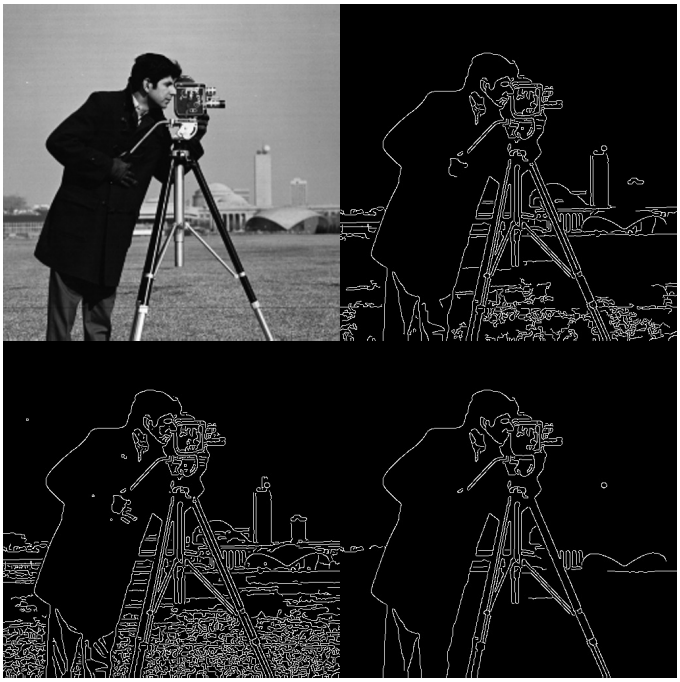

Figura 1.2: Il campo dell'elaborazione digitale di immagini è molto ampio e sarebbe possibile integrare anche operazioni avanzate (edge detection, pattern recognition etc.), che richiederebbero risorse computazionali molto elevate. Per lo scopo di questa tesi ci si limiterà a elaborazioni più semplici, ma in ogni caso rilevanti e sufficienti a mettere alla prova la CPU.

1.2.2 Metodologie confrontate

Le due modalità di sviluppo in esame riguarderanno l'utilizzo delle tecnologie JavaScript e WebAssembly lato server e quindi rispettivamente del runtime environment Node.js e

del linguaggio di programmazione **Rust** in combinazione con WebAssembly System Interface[4].
La scelta di Node.js deriva dal suo utilizzo sempre maggiore grazie all'utilizzo del linguaggio JavaScript, dall'approccio asincrono nella gestione delle richieste e dalla sua ottima scalabilità per applicazioni fortemente File-System-Intensive. Per quanto riguarda la seconda tecnologia si è invece optato per Rust, in quanto consente sia la scrittura di **moduli** che succesivamente **compilabili in WebAssembly**, sia il loro utilizzo efficiente all'interno del codice, mantenendo in questo modo un'ottima coerenza e risultando, teoricamente, una buona scelta per lo sviluppo di applicazioni computazionalmente complesse.

1.2.3 Analisi Comparativa delle Tecnologie

Un elemento iniziale di questa ricerca sarà un'analisi dettagliata delle tecnologie prese in esame. Inizialmente, nel capitolo 2, verrà svolta un'analisi comparativa delle due tecnologie, presentando i vantaggi e gli svantaggi teorici che caratterizzano entrambe le opzioni.
Sarà infatti fondamentale comprendere come ciascuna affronti la complessità legata ad operazioni I/O-intensive e CPU-intensive. Questo ci permetterà di valutare nel modo più opportuno i risultati che emergeranno succesivamente durante i test e i benchmark dell'applicazione sviluppata.
Si procederà poi ad illustrare in modo approfondito il funzionamento delle API sfruttate. Verranno analizzate le peculiarità del linguaggio Rust che lo rendono adatto ad applicazioni ad alta intensità computazionale. Allo stesso modo ci si soffermerà sull'efficacia di WebAssembly nell'esecuzione di codice di basso livello con prestazioni paragonabili a quelle dei linguaggi nativi.

Figura 1.3: Rust e WebAssembly

Nel contempo si analizzerà anche la metodologia basata su Node.js, concentrandosi sull'efficienza e la flessibilità che questo ambiente di esecuzione JavaScript può offrire in ambito web.

Figura 1.4: Node.js

Si proseguirà affrontando un'approfondimento sulle prestazioni di ciascuna tecnologia, evidenziando scenari in cui una risulti più vantaggiosa. Questo approfondimento sarà alla base delle successive valutazioni sulle prestazioni delle applicazioni sviluppate con i due metodi presi in esame.

1.2.4 Valutazione dell'Impatto di Wasm

WebAssembly, in particolare, è emerso come **un'innovazione** cruciale nel mondo dello sviluppo web. Consentendo l'esecuzione di codice a basso livello con prestazioni paragonabili a quelle dei linguaggi nativi, Wasm offre la possibilità di ottenere prestazioni elevate all'interno di un ambiente browser-based.
Parallelamente, WebAssembly System Interface (WASI) gioca un ruolo chiave nell'estendere il potenziale di WebAssembly. WASI fornisce un'interfaccia standardizzata per l'**accesso a risorse di sistema**, consentendo alle applicazioni di interagire con l'ambiente circostante in modo controllato e sicuro. Tale capacità è particolarmente rilevante nell'ambito delle applicazioni web CPU-intensive, in quanto consente di accedere, in maniera efficiente, alle risorse necessarie per eseguire complesse operazioni di calcolo e manipolazione dei dati.
Sarà quindi fondamentale valutare e cercare di misurare l'impatto di WASI, sia per quanto riguarda le prestazioni sia per quanto riguarda lo sviluppo e l'integrazione dello stesso all'interno di un'applicazione web.

1.2.5 Obiettivi

Si intende esplorare le opportunità offerte dalle tecnologie enunciate sopra, nell'ottica di un'ottimizzazione delle prestazioni. In questo contesto l'obiettivo centrale è quello di comprendere i **benefici specifici** legati a ciascun approccio, individuando le circostanze in cui uno dei due possa risultare vantaggioso in termini di **efficienza computazionale** e reattività. Inoltre si intende sottolineare il fatto che le decisioni intraprese sin dalla fase di progettazione, possono avere un impatto significativo sulle prestazioni e sull'esperienza utente di un'applicazione.
Si vuole mettere in evidenza l'importanza di una valutazione attenta e accurata dell'architettura e delle esigenze dell'applicazione stessa. Aspetto da affrontare attraverso un'analisi dettagliata che considera le funzionalità dell'applicazione e valuta se esse siano orientate al calcolo intensivo o ad un'ampia manipolazione del File System.
Per raggiungere in maniera **efficace e quantificabile** gli obiettivi, verrà fatta un'analisi approfondita delle performance e delle prestazioni delle due applicazioni sviluppate. Questo consentirà di valutare in modo empirico e misurabile l'impatto di ciascuna tecnologia nel contesto di applicazioni CPU-Intensive.
Sarà inoltre trattato anche il tema della **scalabilità**. Esso diventa di primaria importanza quando i volumi di traffico e i carichi di lavoro aumentano. Valutare come diverse tecnologie gestiscano richieste concorrenti è fondamentale per determinare quale sia più adatta alle esigenze di un progetto.
Infine si considererà anche la **facilità di sviluppo e l'espandibilità**. Questi temi avanzano spesso di pari passo ed è importante, in un ambiente web in rapida evoluzione, che un progetto si adatti facilmente, sia all'aggiunta di nuovi requisiti e funzionalità, sia all'integrazione con altri sistemi e servizi. Tale flessibilità può infatti avere un impatto significativo sulla vita e sulla sostenibilità di un progetto.

1.3 Struttura della tesi

La tesi seguirà una struttura articolata in modo da affrontare in maniera approfondita gli aspetti chiave delle tecnologie e delle analisi proposte.
Nel Capitolo 2, sarà presentata una panoramica esaustiva delle tecnologie coinvolte in questa ricerca. Questo capitolo fornirà un'analisi dettagliata di WebAssembly/WASI in combinazione con Rust e di Node.js, esaminandone le caratteristiche, i vantaggi e le **API** utili allo sviluppo dell'applicazione. Verranno esplorati i contesti in cui ciascuna tecnologia si dimostra più adeguata, fornendo le basi per una comprensione approfondita delle successive misurazioni e valutazioni.
Nel Capitolo 3, il focus sarà sul prototipo dell'applicazione sviluppata. Saranno illustrate le scelte progettuali, l'architettura complessiva e le funzionalità implementate. Successivamente, verranno presentati i dettagli delle misurazioni condotte, insieme ai criteri di valutazione delle prestazioni delle tecnologie in esame. Saranno discussi i risultati ottenuti e i confronti tra le diverse implementazioni, evidenziando gli aspetti in cui ognuno degli approcci studiati, si distingue in termini di ottimizzazione delle prestazioni.

Capitolo 2

Tecnologie utilizzate

2.1 Introduzione alle tecnologie

Come già introdotto brevemente nel capitolo 1, per lo scopo di questa tesi si è scelto di confrontare due approcci differenti, ma sempre più utilizzati nelle applicazioni web moderne. In particolare si partirà da WebAssembly e dall'interfaccia di sistema WASI, per finire con un'introduzione anche su Node.js.

2.2 WebAssembly

WebAssembly (Wasm) è uno standard che definisce un formato binario (.wasm) e un relativo formato testuale (.wat) per la scrittura di codice eseguibile nelle pagine web. Wasm è attualmente sviluppato in maniera open-source, da un *Community Group* del W3C che comprende membri di tutti i browser più utilizzati.
Esso è nato principalmente come integrazione a JavaScript, per consentire l'esecuzione di codice ad una velocità paragonabile a quella del codice nativo, grazie alla dimensione ridotta dei file binari generati e alla loro efficienza.
Inoltre grazie all'esecuzione all'interno di una sandbox è garantita un ottima sicurezza, anche sotto il punto di vista della memoria.
Grazie al formato testuale (.wat), è possbile eseguire debug, test, ottimizzazioni, scrivere a mano programmi di basso livello, ma anche visualizzare i sorgenti dei moduli wasm, quando questi sono utilizzati una pagina web. [5]

Al giorno d'oggi è in costante crescita, sia il numero di linguaggi che possono avere come *compilation target* WebAssembly (Rust, C, Go, Kotiln, etc), sia il numero di siti che sfruttano tale tecnologia, grazie anche al fatto che praticamente ogni browser attualmente utilizzato, supporta Wasm.

2.2.1 Storia e Origini

I predecessori

Nel 2011 Google rilasciò un progetto open source chiamato **Native Client (NaCl)**. L'obiettivo era quello di consentire l'esecuzione di codice nativo nel browser all'interno di una sandbox con privilegi limitati. In particolare si stava cercando di supportare software che richiedevano grande sforzo computazionale (simulazioni, elaborazioni di audio e video e giochi).
Il progetto si rivelò un successo sotto il punto di vista prestazionale. Vennero infatti rilasciati diversi software e giochi che presentavano prestazioni simili alla rispettiva versione Desktop. Non mancavano però diversi problemi. Il codice ottenuto era eseguibile solo nel browser Chrome ed era impossibile l'interazione con JavaScript o con altre API sul web.
Un tentativo di evoluzione fu presentato nel 2013 dal team di Mozilla. Si trattava di **asm.js**, un sottoinsieme di JavaScript che consentiva l'invocazione di funzioni scritte in linguaggi come C, C++ o Rust, in diversi browser e direttamente da JavaScript. A discapito di una maggior portabilità ci fu una significativa diminuizione delle prestazioni, dovuta alla lentezza dell'interprete JavaScript, che era stato caricato di un notevole overhead.
Queste due soluzioni dimostrarono la possbilità di eseguire codice in una sandbox, o con ottime prestazioni, ma solo all'interno di Chrome (NaCl), oppure in diversi browser ma con prestazioni decisamente inferiori. Si voleva quindi trovare un modo per unificare gli enormi vantaggi offerti da ognuno dei due approcci.[6]

La nascita di Wasm

Fu nel Giugno 2015 che Brendan Eich (creatore di JavaScript), insieme ad altri sviluppatori di Mozilla, annunciarono che lo sviluppo di WebAssembly era cominciato.[7]
Wasm venne presentato come "un nuovo standard open source che definiva un formato e

un modello di esecuzione portabile, efficiente in termini di dimensioni e tempo di caricamento, specificamente progettato per essere un target di compilazione per il web". WebAssembly prometteva prestazioni fino a 20 volte superiori rispetto ad asm.js, grazie alla maggiore velocità di decodifica dei file binari rispetto a quella di parsing da parte dell'interprete JavaScript per i file di asm.js. Nel 2017 venne lanciato il *minimum viable product* (MVP), che conteneva pressochè le stesse funzionalità presenti in asm.js e venne dichiarata conclusa la fase di preview. Capendo le potenzialità di ciò che stava venendo sviluppato hanno dato il loro contributo al progetto, aziende del calibro di Google, Microsoft, Apple, Unity.

2.2.2 Concetti chiave

WebAssembly codifica un linguaggio di programmazione di basso livello, simile ad Assembly. Tale linguaggio è strutturato attorno ai seguenti concetti:[8]

Valori

In WebAssembly sono presenti:

- Un tipo *byte* per la rappresentazione di byte non interpretati
- Quattro tipi di valori numerici: interi e numeri a virgola mobile, ognuno da 32 o 64 bit (*i32, i64, f32, f64*)
- Un tipo vector a 128 bit(*i128*) contenente anch'esso valori numerici (ad esempio da 2 f64, oppure da 4 i32 etc.)
- Un tipo riferimento per puntatori a differenti entità.

Quest'ultimo è definito "opaco", in quanto non è visibile né la loro dimensione, né la loro rappresentazione in bit. Al contrario i primi due tipi si dicono "trasparenti".
Infine è anche possibile .

Istruzioni

Il modello computazionale di WebAssembly è basato su uno stack. Il codice è costituito da una sequenza di istruzioni eseguite in ordine. Le istruzioni sfruttano una struttura dati detta *operand stack* e possono essere di tipo semplice, o di controllo.

Le operazioni semplici svolgono manipolazioni basilari sui dati, prelevando parametri dallo stack (*pop*) e inserendo il risultato nello stesso (*push*). Le operazioni di controllo, si occupano invece di alterare il flusso di esecuzione grazie a costrutti condizionali, blocchi e cicli.

Traps

Alcune istruzioni possono fallire e generare degli errori (traps) che non è possibile gestire all'interno di WebAssembly. Tali errori vengono infatti lanciati nell'ambiente di esecuzione dell'host dove, al contrario, vengono normalmente gestiti.

Funzioni

Il codice è diviso in funzioni. Ognuna di queste ha una certe sequenza di valori sia come parametri che come tipo di ritorno. In una funzione viene eseguita una serie di istruzioni, possono inoltre essere chiamate altre funzioni (anche ricorsivamente) e create variabili locali.

Tabelle

Una tabella è un array di valori "opachi" di un particolare tipo. Tale struttura consente al programma di ottenere i valori indirettamente attraverso un indice dinamico. Tramite le tabelle è possibile, per esempio invocare funzioni indirettamente, emulando in questo modo i puntatori a funzione.

Memoria lineare

La memoria lineare è un array continuo di byte. Ha una dimensione iniziale che può crescere dinamicamente al bisogno. Un programma può leggere e scrivere valore in memoria (operazioni di *load/store*) in un qualsiasi indirizzo al suo interno (anche in maniera non allineata).

Moduli

Un modulo è l'unità di deployment per un programma WebAssembly. Esso conterrà le definizioni di funzioni, tabelle, memoria etc. In un modulo è anche possibile esportare o

importare definizioni, inizializzare tabelle o memoria lineare e anche definire una funzione *start* che verrà eseguita automaticamente.

Embedder

Solitamente un modulo WebAssembly sarà integrato in un host che ne definirà l'inizializzazione, la risoluzione delle funzioni importate e le modalità di accesso di quelle esportate. L'Embedder è l'entità che implementa la connessione tra l'ambiente host e il modulo Wasm. Ci si aspetta che l'embedder interagisca con la semantica di WebAssembly in un modo ben definito nelle specifiche del formato Wasm.

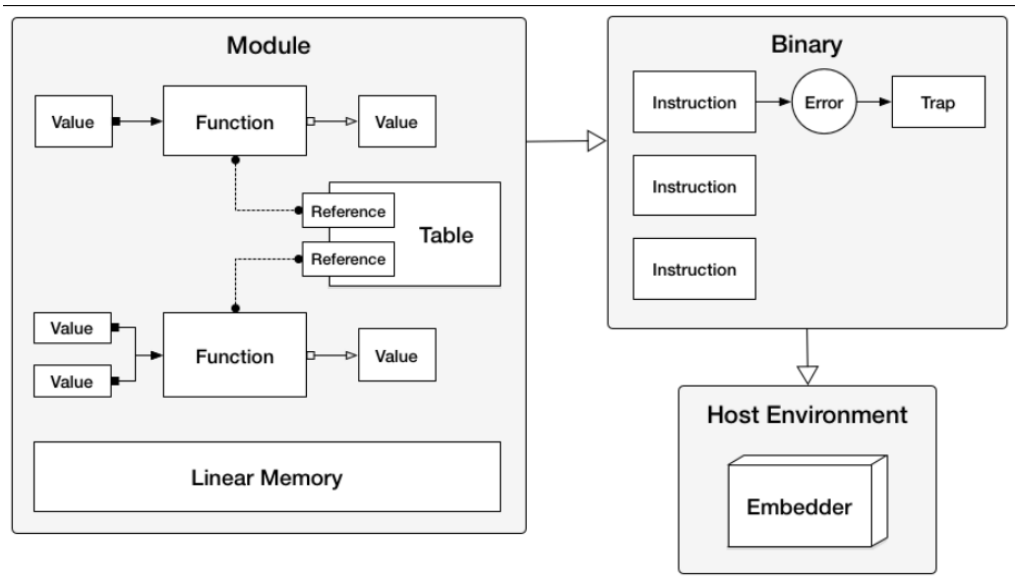

Figura 2.1: Architettura di WebAssembly

2.2.3 Fasi semantiche

Nella semantica di WebAssembly è possibile individuare tre fasi principali: decodifica, validazione ed esecuzione.[8]

Decodifica

I moduli WebAssembly sono distribuiti in formato binario (.wasm) e per questo è necessario decodificarli in modo da ottenere una rappresentazione interna del modulo, con cui il web browser o il runtime environment potrà lavorare.

Validazione

Dopo aver decodificato i moduli binari, per poterli istanziare, è necessario controllare che questi siano validi. La validità è verificata grazie ad un sistema di tipi basato sulla sintassi astratta di un modulo e sul suo contenuto. In particolare per ogni componente della sintassi è presente una regola che specifica le condizioni da rispettare perchè il modulo risulti valido.

(value types) $t ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$
(packed types) $tp ::= \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32}$
(function types) $tf ::= t^* \to t^*$
(global types) $tg ::= \mathsf{mut}^? \; t$

$unop_{iN} ::= \textbf{clz} \mid \textbf{ctz} \mid \textbf{popcnt}$
$unop_{fN} ::= \textbf{neg} \mid \textbf{abs} \mid \textbf{ceil} \mid \textbf{floor} \mid \textbf{trunc} \mid \textbf{nearest} \mid \textbf{sqrt}$
$binop_{iN} ::= \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div\_}sx \mid \textbf{rem\_}sx \mid$
$\quad \textbf{and} \mid \textbf{or} \mid \textbf{xor} \mid \textbf{shl} \mid \textbf{shr\_}sx \mid \textbf{rotl} \mid \textbf{rotr}$
$binop_{fN} ::= \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div} \mid \textbf{min} \mid \textbf{max} \mid \textbf{copysign}$
$testop_{iN} ::= \textbf{eqz}$
$relop_{iN} ::= \textbf{eq} \mid \textbf{ne} \mid \textbf{lt\_}sx \mid \textbf{gt\_}sx \mid \textbf{le\_}sx \mid \textbf{ge\_}sx$
$relop_{fN} ::= \textbf{eq} \mid \textbf{ne} \mid \textbf{lt} \mid \textbf{gt} \mid \textbf{le} \mid \textbf{ge}$
$cvtop ::= \textbf{convert} \mid \textbf{reinterpret}$
$sx ::= \textbf{s} \mid \textbf{u}$

(instructions) $e ::= \textbf{unreachable} \mid \textbf{nop} \mid \textbf{drop} \mid \textbf{select} \mid$
$\quad \textbf{block}\ tf\ e^*\ \textbf{end} \mid \textbf{loop}\ tf\ e^*\ \textbf{end} \mid \textbf{if}\ tf\ e^*\ \textbf{else}\ e^*\ \textbf{end} \mid$
$\quad \textbf{br}\ i \mid \textbf{br\_if}\ i \mid \textbf{br\_table}\ i^+ \mid \textbf{return} \mid \textbf{call}\ i \mid \textbf{call\_indirect}\ tf \mid$
$\quad \textbf{get\_local}\ i \mid \textbf{set\_local}\ i \mid \textbf{tee\_local}\ i \mid \textbf{get\_global}\ i \mid$
$\quad \textbf{set\_global}\ i \mid t.\textbf{load}\ (tp\_sx)^?\ a\ o \mid t.\textbf{store}\ tp^?\ a\ o \mid$
$\quad \textbf{current\_memory} \mid \textbf{grow\_memory} \mid t.\textbf{const}\ c \mid$
$\quad t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid t.cvtop\ t\_sx^?$

(functions) $f ::= ex^*\ \textbf{func}\ tf\ \textbf{local}\ t^*\ e^* \mid ex^*\ \textbf{func}\ tf\ im$
(globals) $glob ::= ex^*\ \textbf{global}\ tg\ e^* \mid ex^*\ \textbf{global}\ tg\ im$
(tables) $tab ::= ex^*\ \textbf{table}\ n\ i^* \mid ex^*\ \textbf{table}\ n\ im$
(memories) $mem ::= ex^*\ \textbf{memory}\ n \mid ex^*\ \textbf{memory}\ n\ im$
(imports) $im ::= \textbf{import}$ "$name$" "$name$"
(exports) $ex ::= \textbf{export}$ "$name$"
(modules) $m ::= \textbf{module}\ f^*\ glob^*\ tab^?\ mem^?$

Figura 2.2: La sintassi astratta di WebAssembly

Ad esempio viene controllato l'ordine delle istruzioni nel corpo di una funzione assicurandosi che lo stack sia utilizzato nel modo corretto. Oppure se nelle specifiche esaminiamo un'operazione binaria tra tipi numerici (i32.add, f64.xor etc.), notiamo che essa è del tipo $t.binop$ e dato un contesto C essa è valida solamente con tipo $[t\ t] \to [t]$.

Tale regola si può anche rappresentare con la seguente notazione formale:

$$\frac{}{C \vdash t.binop : [t\ t] \rightarrow [t]}$$

Esaminando invece le regole di validazione di un'istruzione per l'aumento di memoria lineare *memory.grow* (tale istruzione si aspetta l'offset che indica di quanto la memoria è da espandere e ritorna la dimensione della memoria precedente all'espansione) notiamo che:

- La memoria $C.mems[0]$ deve essere definita nel contesto.
- Allora l'istruzione risulta valida con tipo $[i32] \rightarrow [i32]$

In notazione formale:

$$\frac{C.mems[0] = memtype}{C \vdash memory.grow : [i32] \rightarrow [i32]}$$

Esecuzione

Terminata la validazione di ogni istruzione il modulo può finalmente essere istanziato. L'istanza di un modulo ne è la rappresentazione dinamica. Esso comprende lo stack, sul quale operano le istruzioni WebAssembly e uno ***store*** astratto, contenente lo stato globale (è la rappresentazione a runtime di tutte le istanze di funzioni, tabelle, memorie etc. che sono state allocate dall'istanziazione). Terminata l'istanziazione, diventa effettivamente possibile l'esecuzione di istruzioni WebAssembly.
In particolare, se nel modulo era stata definita una funzione *_start*, essa sarà eseguita subito dopo la creazione dell'istanza, altrimenti sarà possibile invocare funzioni esportate, chiamandole direttamente dall'istanza stessa.
Per ogni istruzione, è presente una regola che specifica l'effetto della sua esecuzione sullo stato del programma. Rimanendo coerenti con gli esempi sulla validazione, segue il comportamento dell'istruzione *t.binop*:

- In seguito alla validazione, possiamo assumere che due valori di tipo t si trovino in cima allo stack.
- Viene eseguita l'operazione di **pop** del valore $t.const\ c_1$ dallo stack
- Viene eseguita l'operazione di **pop** del valore $t.const\ c_2$ dallo stack
- Se $binop_t(c_1, c_2)$ è definita allora:

  - Sia c il possibile risultato di $binop_t(c_1, c_2)$
  - Viene eseguita l'operazione di **push** del valore $t.const\ c$ nello stack

- Altrimenti:
  - Trap

In notazione formale:

$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \quad \hookrightarrow \quad (t.\mathsf{const}\ c) \qquad (\text{if } c \in binop_t(c_1, c_2))$$
$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \quad \hookrightarrow \quad \mathsf{trap} \qquad (\text{if } binop_t(c_1, c_2) = \{\})$$

Non viene riportato l'esempio anche sull'istruzione *memory.grow* essendo una funzionalità decisamente più complessa (11 diversi passaggi, con svariati sottocasi) che comporterebbe obbligatoriamente, l'introduzione di ulteriori dettagli che esulano dallo scopo di questa tesi. Si noti che sia l'istanziazione, che l'invocazione di funzioni, sono operazioni che avvengono all'interno dell'ambiente di esecuzione dell'host.

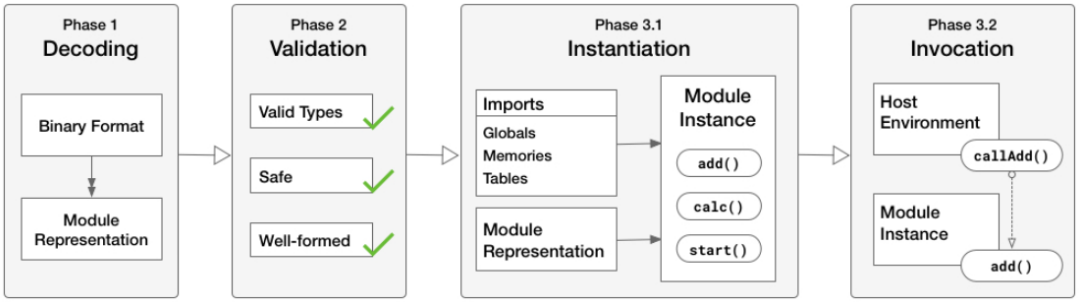

Figura 2.3: La semantica di WebAssembly

2.2.4 Correttezza logica

Grazie alla sua semantica è possibile dire che il sistema di tipi di WebAssembly è logicamente corretto (*sound*)[9]. Esso infatti garantisce sia ***type safety***, che ***memory safety***. Per quanto riguarda la sicurezza rispetto ai tipi, tutti i tipi controllati durante la validazione saranno rispettati anche a runtime:

- Ogni variabile (locale o globale) conterrà valori del tipo corretto
- Ogni istruzione verrà applicata solo ad operandi del tipo che ci si aspetta e restituirà valori del tipo corretto
- Ogni funzione restituirà valori del tipo previsto, a meno di errori (trap)

Per quanto riguarda la *memory safety*, è garantito che non verrà acceduta nesun'area di memoria, diversa da quelle esplicitamente specificate dal programma. Ogni volta che per esempio verrà fatta un'operazione di load/store in un certo indirizzo di memoria, si controllerà che quest'ultimo sia nel range di indirizzi possibili per l'istanza attuale di WebAssembly.
Inoltre sono garantite altre proprietà:

- L'assenza di comportamenti indefiniti: le regole di esecuzione sono mutualmente consistenti e coprono qualsiasi caso che possa capitare in un programma validato
- L'incapsulamento per funzioni e moduli: nessuna variabile locale può essere acceduta fuori dalla funzione dove è dichiarata e nessun componente di un modulo può essere acceduto fuori dallo stesso a meno che quest'ultimo non sia esportato o importato

Infine è importante notare che le regole di validazione si occupano solamente dei componenti statici di un programma WebAssembly. Per dimostrare correttezza in maniera precisa, sono state estese le *typing rules* anche ai componenti dinamici, come lo **store**, le **configurations** (coppie Store - Thread di esecuzione) e le istruzioni amministrative. Dopo aver dato la definizioni di configurazione valida è stato possibile enunciare due teoremi e derivare un corollario che riassumendo dicono che:
Ogni thread in una *configuration* valida o esegue per sempre, o termina con un errore (trap), o termina con un risultato del tipo atteso. Di conseguenza, dato uno *store* valido, nessun calcolo derivante dall'istanzazione o dall'invocazione di un modulo valido, può avere comportamenti diversi da quelli definiti nelle specifiche. [10]

2.3 WebAssembly System Interface

2.3.1 Ruolo di WebAssembly System Interface

2.3.2 Rust e WASI

2.3.3 Struttura di WASI

2.3.4 Wasmtime

2.3.5 Applicazioni e Casistiche d'Uso di WASI

2.4 Node.js

2.4.1 Panoramica di Node.js

2.4.2 Vantaggi di Node.js

2.4.3 Ecosistema di Node.js

2.5 Confronto tra tecnologie

2.5.1 Prestazioni

2.5.2 Sicurezza

2.5.3 Facilità di Sviluppo

2.5.4 Scalabilità ed Espandibilità

2.6 Conclusioni preliminari

```
1 import suca
2
3 print "hello␣world!"
```

Listato 2.1: I directly included a portion of a file

```
1 public void prepare(AClass foo) {
2         AnotherClass bar = new AnotherClass(foo)
3 }
```

Listato 2.2: Some code in another language than the default one

Capitolo 3

Capitolo 3

3.1 Prototipo

Ringraziamenti

Grazie a Grazia, Graziella e la sorella.

Bibliografia

[1] *Node.js*. URL: https://nodejs.org/.

[2] *WebAssembly*. URL: https://webassembly.org.

[3] *WebAssembly System Interface*. URL: https://wasi.dev/.

[4] *Rust and WebAssembly*. URL: https://www.rust-lang.org/what/wasm.

[5] *WebAssembly Design*. URL: https://github.com/WebAssembly/design.

[6] Brian Sletten. *WebAssembly: The Definitive Guide*. O'Reilly Media, Inc, 2021. URL: www.oreilly.com/library/view/webassembly-the-definitive/9781492089834/.

[7] Brendan Eich. «From ASM.JS to WebAssembly». In: (2015). URL: https://brendaneich.com/2015/06/from-asm-js-to-webassembly.

[8] Andreas Rossberg, cur. *WebAssembly Specification*. 2023. URL: https://webassembly.github.io/spec/core/#webassembly-specification.

[9] *Soundness of WebAssembly*. URL: https://webassembly.github.io/spec/core/appendix/properties.html.

[10] *Soundness of WebAssembly, theorems*. URL: https://webassembly.github.io/spec/core/appendix/properties.html#theorems.