Università degli Studi di Verona

Dipartimento di Informatica

Corso di Laurea in Informatica

Estensione di Turing Arena Light: Integrazione di valutatori WebAssembly

Filippo Barbieri

Relatore
Prof. Romeo Rizzi

Correlatore
Dr. Dario Ostuni

A.A. 2023/2024

Indice

Capitolo 1

Introduzione

Nel primo capitolo si vogliono brevemente descrivere le diverse tecnologie esplorate durante il tirocinio, grazie al quale è stato possibile acquisire competenze pratiche e teoriche che hanno avuto un ruolo fondamentale nel lavoro di tesi.

1.1 WebAssembly

1.1.1 Nascita di WebAssembly

WebAssembly è nato dall'esigenza di superare i limiti prestazionali di JavaScript all'interno dei browser. Gli sviluppatori cercavano un linguaggio di basso livello in grado di offrire prestazioni vicine a quelle native, ma che fosse indipendente dall'architettura sottostante.
Il primo tentativo in questa direzione è stato asm.js, un sottoinsieme di JavaScript che permetteva di compilare codice scritto in C/C++ per l'utilizzo nei browser. Tuttavia, poiché era legato ai motori JavaScript, asm.js ereditava anche le limitazioni del linguaggio.
Annunciato nel 2015 e rilasciato nel 2017, WebAssembly è un linguaggio di basso livello ispirato ad Assembly progettato per fornire un target di compilazione per l'esecuzione web ai linguaggi non predisposti ed eseguire pressoché con le stesse prestazioni di quest'ultimi. Può essere eseguito su browser contestualmente a JavaScript o in altri ambienti nativi, di cui il più noto è Wasmtime, grazie a un insieme di API denominate WASI. Attualmente[1], i linguaggi che supportano stabilmente la compilazione in WebAssembly sono C, C++, Rust e Go. Esistono anche compilatori, seppur ancora instabili, per molti altri linguaggi, tra cui Python e Java.

1.1.2 Sintassi e funzionamento

Il codice WebAssembly, in formato binario (bytecode), è progettato per essere eseguito su una macchina virtuale basata su stack. Questo bytecode è pensato per essere più rapido da analizzare ed eseguire rispetto a JavaScript e per offrire una rappresentazione più compatta del codice.
Durante l'esecuzione, le istruzioni vengono elaborate concettualmente attraverso un normale program counter. Tuttavia, in pratica, la maggior parte dei motori WebAssembly converte il bytecode in codice macchina nativo per eseguirlo direttamente.

[1] Ottobre 2024.

Le istruzioni WebAssembly si suddividono in due categorie:

- istruzioni di controllo che gestiscono i costrutti di controllo del flusso di esecuzione. Possono modificare il program counter, estraggono valori dalla pila e inseriscono i risultati nuovamente sulla pila;
- istruzioni semplici che prelevano valori dalla pila, applicano un operatore, inseriscono i risultati sulla pila e avanzano implicitamente il contatore del programma.

Il formato binario `.wasm` è progettato per l'esecuzione da parte delle macchine, non per la lettura o la scrittura manuale. La rappresentazione testuale di WebAssembly, nota come WebAssembly Text Format (WAT), è pensata per essere più leggibile e interpretabile dagli sviluppatori, consentendo di visualizzare e modificare il codice in modo più comprensibile. Sebbene sia possibile scrivere manualmente il codice in WAT, è più comune generarlo automaticamente durante il processo di sviluppo. Per convertire tra il formato testuale e quello binario, si utilizzano strumenti come *wat2wasm*[2], che trasforma il codice `.wat` in bytecode `.wasm`, e *wasm2wat*[3], che esegue la conversione inversa.

WebAssembly: Somma di due numeri

```
(module
  (func $somma (param $num1 i32) (param $num2 i32) (
     result i32)
    local.get $num1
    local.get $num2
    i32.add)
  (export "somma" (func $somma))
)
```

1.1.3 WASI

WASI (WebAssembly System Interface) è un insieme di API progettate per il software compilato secondo lo standard WebAssembly con lo scopo di fornire un'interfaccia sicura e standardizzata per le applicazioni compilate in WebAssembly, permettendone l'esecuzione in diversi ambienti. L'obiettivo di WASI è semplificare l'integrazione di software scritto in linguaggi diversi, evitando l'uso di sistemi complessi e poco efficienti. La versione più recente di WASI è la 0.2 (Preview 2) e il progetto è ancora in sviluppo e aperto a contributi sul repository GitHub[4].

[2]W3C WebAssembly Community Group, *WebAssembly Binary Toolkit*.
[3]W3C WebAssembly Community Group, *WebAssembly Binary Toolkit*.
[4]W3C WebAssembly Community Group, *WASI repository*.

1.1.4 Wasmtime

Wasmtime è un ambiente di esecuzione per WebAssembly al di fuori del web, utilizzabile sia come utility a riga di comando che come libreria incorporata in un'applicazione più ampia. È pensato per essere:

- veloce: Wasmtime è progettato per istanziare moduli in modo efficiente, mantenendo basse le latenze nelle chiamate tra l'host e WebAssembly;
- sicuro: la sicurezza è un elemento centrale nello sviluppo di Wasmtime, che sfrutta le garanzie di sicurezza di Rust;
- configurabile: Wasmtime offre ampie opzioni di configurazione per adattarsi a diversi ambienti, dal consumo ridotto di risorse in piccoli dispositivi alla gestione di molteplici istanze su grandi server;
- WASI "friendly": Wasmtime supporta una gamma completa di API WASI per consentire alle applicazioni di interagire con il sistema host, offrendo accesso a risorse come file system, rete e altro;
- conforme agli standard: Wasmtime è pienamente conforme agli standard ufficiali di WebAssembly e supporta le proposte future di WebAssembly, con gli sviluppatori di attivamente coinvolti nel processo di standardizzazione.

1.2 Turing Arena Light

Turing Arena Light è una piattaforma di gestione di gara (di programmazione competitiva[5]) pensata da Romeo Rizzi[6] e Dario Ostuni[7] per l'insegnamento della programmazione in classe, con focus su semplicità, interattività, flessibilità ed estensibilità.

1.2.1 Architettura e design

L'idea centrale di Turing Arena Light è far interagire due programmi tramite *stdin* e *stdout*. Il primo è il "manager", che fornisce l'input e valuta l'output, emettendo un verdetto; il secondo è la "soluzione", scritta dal concorrente per risolvere il problema. A differenza di altri sistemi di gestione, dove i programmi vengono eseguiti sulla stessa macchina, Turing Arena Light esegue i due programmi su macchine diverse, consentendo interazioni in tempo reale.
In Turing Arena Light, un problema è definito da un insieme di servizi e allegati descritti nel file `meta.yaml`. Un servizio specifica i parametri e il programma da eseguire, ossia il manager o **valutatore**, che interagisce con la soluzione del concorrente.
Si noti a questo punto che Turing Arena Light è una mera specifica che definisce come strutturare un problema e gestire l'interazione tra il gestore del problema e la soluzione del concorrente. Essendo semplice e senza richiedere tecnologie particolari come il sandboxing[8], può avere diverse implementazioni. Al momento, esiste un'unica implementazione di riferimento, **rtal** (Rust Turing Arena Light), scritta in Rust.

[5] La programmazione competitiva è l'abilità di risolvere rapidamente problemi informatici, competendo contro altri partecipanti. È un ottimo metodo per apprendere algoritmi, strutture dati e affinare le capacità di problem-solving.

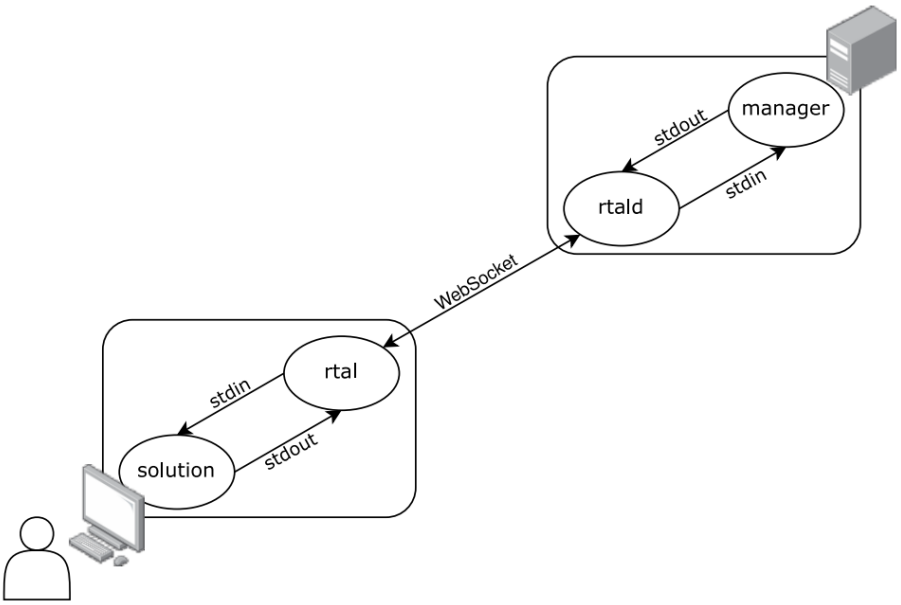
[6] Prof. Romeo Rizzi, `https://www.di.univr.it/?ent=persona&id=8814`.

[7] Dr. Dario Ostuni, `https://www.di.univr.it/?ent=persona&id=64817`.

[8] Il sandboxing è una tecnica di sicurezza informatica che isola e limita l'esecuzione di programmi o processi all'interno di un ambiente controllato e sicuro, noto appunto come "sandbox".

Il server (`rtald`) gestisce le richieste dei client, genera il valutatore e coordina l'interazione tra la soluzione e il valutatore. Il client (`rtal`) esegue la soluzione del concorrente (in locale, scritta in qualsiasi linguaggio), connettendosi al server e fungendo da proxy per la comunicazione.

Schema di comunicazione di Turing Arena Light



[a]

[a]Dario Ostuni, «Design and Evaluation of Competitive Programming Platforms for Computer Science Education».

1.2.2 Dettagli implementativi

L'implementazione di Rust Turing Arena Light è composta da tre componenti: server (`rtald`), client (`rtal`) e checker (`rtalc`). Questi componenti condividono una descrizione del problema, serializzata e deserializzata in YAML attraverso *serde*, un framework di serializzazione per Rust.
Il checker è un programma a riga di comando indipendente che prende in input una directory contenente la descrizione di un problema e ne verifica la validità prima che venga caricato sul server.
Il client e il server si occupano principalmente della creazione di processi e della gestione della rete, utilizzando la libreria *tokio* per l'asincronia. Il server genera il valutatore del problema e il client la soluzione, gestendo i canali di input e output in modo efficiente. Le comunicazioni tra client e server avvengono tramite *WebSockets*, rendendo possibile anche l'implementazione del client come applicazione web.
Entrambi i componenti eseguono processi in un ambiente senza sandbox: questa scelta è giustificata dal fatto che il client esegue il codice sulla macchina locale del concorrente, mentre il server esegue codice tendenzialmente scritto dall'organizzatore (considerato affidabile), riducendo così la complessità senza compromettere la sicurezza.

1.3 Rust

Rust è un linguaggio di programmazione compilato progettato per essere efficiente e sicuro, rendendolo ideale per lo sviluppo di software di sistema. La prima versione del compilatore Rust (`rustc`) è stata rilasciata nel gennaio 2012, mentre la versione stabile 1.0 è stata pubblicata il 15 maggio 2015.

1.3.1 Concetti peculiari del linguaggio

Ownership

La ownership (in italiano "proprietà", il termine inglese è stato conservato in quanto la versione italiana non esprime appieno il significato originale) è un concetto centrale in Rust che consente di gestire la memoria in modo sicuro senza l'uso di un garbage collector[9]. A differenza di altri linguaggi, dove la memoria viene liberata automaticamente o manualmente, Rust adotta un sistema basato su regole controllate a tempo di compilazione. Questo garantisce la sicurezza della memoria senza compromettere le prestazioni del programma. Le regole di ownership sono semplici ma fondamentali:

- ogni valore ha un proprietario;
- può esserci un solo proprietario alla volta;
- quando il proprietario di un valore esce dallo scope, il valore viene automaticamente deallocato, liberando la memoria associata.

A seconda del tipo, le variabili in Rust possono assumere due comportamenti (*trait*): `move` o `copy`. Con `move`, la ownership di un dato viene trasferita a una nuova variabile, rendendo la variabile originale non più utilizzabile. Con `copy`, invece, i dati vengono duplicati, permettendo a entrambe le variabili di rimanere valide e distinte (modificare una non influisce sull'altra).

References e borrowing

In Rust, le reference (in italiano "riferimenti") sono simili ai puntatori in altri linguaggi come C o C++, in quanto permettono di accedere a dati senza trasferirne la ownership. Tuttavia, a differenza dei puntatori, un riferimento è garantito che punti a un valore valido di un particolare tipo per tutta la durata del riferimento stesso, evitando problemi di dereferenziamento di puntatori null o non validi.
L'atto di creare un riferimento è noto come "borrowing" (prendere in prestito, trad.), analogamente a quanto avviene nella vita quotidiana: quando una persona possiede un oggetto, è possibile prenderlo in prestito, ma una volta terminato il suo utilizzo, è necessario restituirlo.
Quando una funzione accetta una reference a un valore, non ne acquisisce la ownership e il valore rimane utilizzabile dal chiamante anche dopo la chiamata.

Lifetime

I lifetime (in italiano "tempo di vita") garantiscono che i riferimenti rimangano validi per tutto il tempo necessario. Nella maggior parte dei casi, i lifetime sono impliciti e

[9] Meccanismo di gestione della memoria attraverso cui il sistema operativo, oppure il compilatore insieme a un modulo di runtime, si occupa di liberare automaticamente le porzioni di memoria non più utilizzate dalle applicazioni.

vengono inferiti automaticamente dal compilatore. Tuttavia, è necessario specificarli esplicitamente quando i lifetime dei riferimenti possono interagire o sovrapporsi in modi diversi. Consideriamo, per esempio, una funzione che restituisce il riferimento alla stringa più lunga tra due passati come parametri:

Rust: lifetime espliciti

```rust
fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
```

Senza indicare i lifetime, il compilatore di Rust non sa per quanto tempo le stringhe vivranno e non può verificare che il riferimento restituito rimanga valido per l'intera durata necessaria. Esplicitando i lifetime, si stabilisce che il riferimento restituito avrà lo stesso lifetime di uno degli input, evitando così problemi di dangling references (riferimenti a dati non più validi).

1.3.2 Scelta del linguaggio per implementare Turing Arena Light

La scelta di utilizzare il linguaggio Rust è stata guidata dalla sua natura di linguaggio system level, particolarmente adatto per sviluppare applicazioni che devono comunicare con il sistema operativo e altre applicazioni. Inoltre, un elemento fondamentale è la portabilità: essendo un linguaggio compilato, Rust produce binari che richiedono pochissime dipendenze esterne per funzionare. Questa caratteristica lo rende ideale per la creazione di programmi facilmente distribuibili. È un aspetto cruciale, considerando che Turing Arena Light è pensato per studenti che potrebbero non avere le competenze tecniche necessarie per installare e configurare sistemi complessi. La possibilità di fornire un unico file binario da scaricare ed eseguire senza bisogno di configurazioni aggiuntive rappresenta un vantaggio significativo.

Capitolo 2

Estensione dei servizi

> *"Turing Arena Light [...] strives to be as simple as possible, while being very flexible and extensible."*[1]

Mantenendo fede ai princìpi su cui si basa Turing Arena Light, il lavoro finale consiste nell'integrare la possibilità di eseguire valutatori WebAssembly per i servizi ad esso predisposti dei problemi caricati sul server, con l'obiettivo di avere un binario che funzioni a prescindere dall'architettura su cui è eseguito e dal linguaggio in cui è stato scritto il sorgente. Questo approccio consente di superare le limitazioni delle piattaforme e dei linguaggi specifici, garantendo così un'elevata portabilità e compatibilità del codice, permettendo agli sviluppatori di compilare una sola volta il sorgente e di eseguirlo ovunque, senza dover affrontare problemi di compatibilità tra diverse architetture hardware.

2.1 Compilazione in WebAssembly

La prima parte del lavoro è volta a compilare un valutatore, scritto in Rust, C o C++, in WebAssembly per i servizi che ne sono predisposti, ossia che nel `meta.yaml` presentano il campo `wasm_evaluator_source`. Il codice si trova nel file `check.rs`, che viene poi compilato nel binario `rtalc` ed utilizzato come utility per verificare la correttezza e preparare un problema ad essere caricato sul server. La funzione chiave è `deploy_wasm`, che si occupa della gestione della compilazione del codice sorgente in un file `.wasm`, utilizzando il compilatore corretto con le flag necessarie.

2.1.1 Descrizione del codice

Fase 0: Inizializzazione e chiamata alla funzione

Il file `meta.yaml` è rappresentato nella variabile `meta` che nel main viene utilizzata per iterare sui servizi esposti dal problema. Per ogni servizio valido, cioè avente valutatore non vuoto, viene verificata la presenza del campo `wasm_evaluator_source` e, in caso positivo, chiamata la funzione `deploy_wasm`.

[1] Dario Ostuni, «Design and Evaluation of Competitive Programming Platforms for Computer Science Education».

check.main: inizializzazione e chiamata

```
if let Some(src) = &service.wasm_evaluator_source {
    if let Err(e) = deploy_wasm(service, name, &dir, src)
        {error!(e);}
}
```

La funzione viene invocata con i seguenti parametri attuali (formali tra parentesi):

- `service` (`service`): riferimento alla struttura del servizio;
- `name` (`service_name`): il nome del servizio;
- `dir` (`service_root`): la directory del servizio, necessaria per accedere ai file;
- `src` (`wasm_src`): il nome del file sorgente che verrà compilato.

Se l'esecuzione della funzione va a buon fine viene ignorato il valore di ritorno, altrimenti viene stampato l'errore generato.

Fase 1: Determinazione del nome del file

La funzione inizia determinando il nome del file binario che verrà generato. Premesso che `service.evaluator` è un vettore contenente il comando che verrà chiamato per eseguire il valutatore; se il primo elemento è `"Wasmtime"`, viene utilizzato il nome specificato come secondo elemento del vettore, altrimenti viene utilizzato il nome del servizio. Questo perché un servizio può prevedere un valutatore WebAssembly anche se non è quello che verrà eseguito, quindi il sorgente viene preparato a prescindere.

check.deploy_wasm: nome file compilato

```
let wasm_filename =
    if service.evaluator.len() > 1 && service.evaluator
       [0].eq("wasmtime") {
        service.evaluator[1].to_owned()
    } else {
        format!("{}.wasm", service_name)
    };
let wasm_file = PathBuf::from(&service_root).join(&wasm_
   filename);
if !wasm_file.exists() {...}
```

Se un file omonimo è già esistente, la funzione termina con esito positivo.

Fase 2: Verifica del file sorgente

Dopo aver verificato che il file da compilare sia presente nella directory del servizio, viene scelto il compilatore a seconda dell'estensione del sorgente.

check.deploy_wasm: Scelta compilatore

```
let mut source_path = PathBuf::from(&service_root);
source_path.push(wasm_src);
if !source_path.exists() {
    return Err(format!("{}: wasm_evaluator_source file
       doesn't exist", service_name));
}
let wasi = format!("--sysroot={}/share/wasi-sysroot", env
   ::var("WASI_SDK_PATH").unwrap_or_default());
let (command_string, command_args) = match source_path.
   extension().and_then(|ext| ext.to_str()) {
    Some("rs") => {
        let mut cargo_path = PathBuf::from(&service_root)
           ;
        cargo_path.push("Cargo.toml");
        if source_path.components().any(|comp| comp ==
           std::path::Component::Normal("src".as_ref()))
            && cargo_path.exists() {
            ("cargo", vec!["build", "--target", "wasm32-
               wasip1", "--release"])
        } else {
            ("rustc", vec![wasm_src, "-o", &wasm_filename,
                "--target", "wasm32-wasip1"])
        }
    }
    Some("cpp") | Some("cc") => {
        ("clang++", vec!["--target=wasm32-wasip1", &wasi,
            wasm_src, "-o", &wasm_filename])
    }
    Some("c") => {
        ("clang", vec!["--target=wasm32-wasip1", &wasi,
           wasm_src, "-o", &wasm_filename])
    }
    _ => return Err(format!("{}: wasm_evaluator_source
       doesn't have a valid extension (.rs, .c, .cpp|.cc
       )", service_name))
};
```

Qualora il sorgente sia scritto in Rust, viene verificato se si tratta di un file unico che può essere compilato direttamente con `rustc`, oppure se è all'interno di un progetto ed è quindi necessario eseguire `cargo build`. In entrambi i casi si deve specificare che il target di compilazione è WebAssembly; nel secondo si aggiunge la flag `release` per ottimizzare il binario prodotto.

Mentre se il sorgente è scritto in C/C++ viene compilato con `clang`, specificando il target WebAssembly e linkando la directory in cui si trovano le API WASI (avendo la root WASI memorizzata tra le variabili d'ambiente) per poter operare sui file.

Fase 3: Compilazione

Una volta definito il compilatore e il comando di compilazione, questo viene eseguito nella directory del servizio e verificato che vada a buon fine. Vengono ignorati eventuali output, discordanti a seconda del compilatore, e in caso di errore la funzione restituisce un avviso generico.

check.deploy_wasm: Compilazione

```
let status = Command::new(command_string)
    .args(&command_args)
    .current_dir(&service_root)
    .stdout(Stdio::null())
    .stderr(Stdio::null())
    .status();
if status.is_err() {
    return Err(format!("{}: Cannot compile to wasm",
       service_name));
}
if command_string == "cargo" {
    let mut target_path = PathBuf::from(&service_root);
    target_path.push("target/wasm32-wasip1/release");
    if let Ok(entries) = fs::read_dir(&target_path) {
        for entry in entries.flatten() {
            if entry.path().extension().and_then(|ext|
               ext.to_str()) == Some("wasm") {
                let mut dest_path = PathBuf::from(&
                   service_root);
                dest_path.push(&wasm_filename);
                if fs::copy(entry.path(), dest_path).
                   is_err() {
                    return Err(format!("{}: Cannot move .
                       wasm to main folder",
                       service_name));
                }
                break;
            }
        }
    }
}
```

Se il sorgente si trova all'interno di un progetto Rust (è stato compilato con `cargo`), il binario viene generato nella directory `target/wasm32-wasip1/release` interna al progetto, quindi deve essere copiato nella root del servizio.

Fase 4: Verifica finale

Può accadere, per problemi legati al file system o errori di compilazione non gestiti precedentemente, che alla fine della procedura ancora non esista un file `.wasm` nella root del servizio. In tal caso viene restituito un errore, altrimenti la funzione termina con esito positivo.

check.deploy_wasm: Verifica finale

```
if !wasm_file.exists() {
    return Err(format!("{}: Could not generate evaluator",
        service_name));
}
```

2.1.2 Considerazioni sulla compilazione da C/C++

Si è scelto di utilizzare Clang direttamente, invece di Emscripten, il più diffuso toolchain per compilare C/C++ in WebAssembly, per le seguenti ragioni:

- Clang supporta nativamente il target `wasm32-wasip1`, risultando ideale per applicazioni standalone. Emscripten, sebbene utilizzi internamente Clang, richiede configurazioni aggiuntive per ottenere lo stesso risultato;
- i moduli WebAssembly generati con Clang tendono a essere più leggeri e performanti. Emscripten, invece, è principalmente ottimizzato per applicazioni in ambiente browser;
- sono state riscontrate difficoltà nel linking tra Emscripten e WASI, senza le cui API è preclusa la gestione del file system nei binari WebAssembly.

Da fare inoltre presente che lo stream *stdout* utilizza un buffer a livello di linea. Ciò significa che l'output viene visualizzato solo quando il buffer raggiunge un carattere `'\n'` o quando viene esplicitamente svuotato. Di conseguenza, per com'è strutturato Turing Arena Light, è necessario assicurarsi che il buffer venga sempre svuotato.
Rust, invece, essendo un linguaggio di recente nascita, prevede nativamente la compilazione in WebAssembly, senza alcun tipo di integrazione esterna.

2.2 Esecuzione di file WebAssembly

Una volta compilato il valutatore WebAssembly, il passo successivo consiste nell'eseguirlo nel contesto della connessione tra client e server. Il codice successivamente riportato, estratto da `connection.rs`, si occupa della configurazione e dell'esecuzione del file `.wasm`, gestendo le variabili d'ambiente necessarie e utilizzando un approccio che permetta l'esecuzione sia tramite Wasmtime che attraverso qualsiasi altro ambiente di esecuzione.

2.2.1 Descrizione del codice

Fase 0: Inizializzazione del comando

Viene istanziato il comando specificato nel primo elemento del vettore `evaluator` del servizio corrente, tendenzialmente `"python"` o `"wasmtime"`, cioè l'esecutore del valutatore.

connection.Client.connect: Inizializzazione valutatore

```
let command_name = &service.evaluator[0];
let mut evaluator = Command::new(command_name);
evaluator.current_dir(&problem.root);
```

Fase 1: Configurazione delle variabili d'ambiente

Le variabili d'ambiente vengono aggiunte in questa fase per fornire al valutatore le informazioni necessarie per l'esecuzione. Utilizzando la closure[2] `add_env_var`, il codice imposta le variabili che influenzano il comportamento del valutatore. Tra le variabili aggiunte, ci sono informazioni sul contesto del problema, quali directory di input/output e nome del servizio, e gli argomenti richiesti dal servizio.

connection.Client.connect: Configurazione ENV

```
for (k, v) in args {
    add_env_var(&format!("TAL_{}", k), &v);
}
add_env_var("TAL_META_DIR", path2string(&problem.root));
add_env_var("TAL_META_CODENAME", &problem.name);
add_env_var("TAL_META_SERVICE", &service_name);
add_env_var("TAL_META_TTY", if tty { "1" } else { "0" })
   ;

let infile_path = path2string(&infile_dir.path());
add_env_var("TAL_META_INPUT_FILES", infile_path);
let outfile_path = path2string(&outfile_dir.path());
add_env_var("TAL_META_OUTPUT_FILES", outfile_path);

if let Some(ref token_info) = token_info {
    add_env_var("TAL_META_EXP_TOKEN", &token_info.token)
       ;
    add_env_var("TAL_META_EXP_LOG_DIR", path2string(&
       token_info.path));
}
add_env_var("TAL_META_EXP_ADDRESS", address);
```

[2]Funzioni anonime che possono essere salvate in variabili o passate come argomenti ad altre funzioni. Possono essere create in un contesto e richiamate altrove, a differenza delle normali funzioni, poiché catturano i valori dello scope in cui sono definite.

Sono state definite due funzioni ausiliarie fondamentali per supportare quanto sopra:

- `path2string`, converte i path di file e directory in stringhe; nel caso in cui la chiamata interna a `path.to_str` restituisca un errore, la funzione restituisce una stringa vuota che può generare un errore a seguire;

  connection.Client.connect.path2str

  ```
  fn path2string(path: &Path) -> &str {
      path.to_str().unwrap_or_else(|| "")
  }
  ```

- `add_env_var` è il cuore pulsante del lavoro svolto, gestisce l'aggiunta di variabili d'ambiente a seconda del comando utilizzato. In particolare, se il comando è wasmtime, il comportamento cambia in modo significativo, dato che le variabili d'ambiente devono essere specificate come argomenti utilizzando la flag `--env`. Questo significa che invece di impostare direttamente le variabili d'ambiente nel sistema, è necessario fornire ciascuna variabile come argomento della linea di comando, seguendo il formato appropriato. Per tutti gli altri comandi, invece, le variabili d'ambiente possono essere semplicemente aggiunte come variabili d'ambiente standard.

  connection.Client.connect.add_env_var

  ```
  let mut add_env_var = |key: &str, value: &str| {
      if command_name.eq(&String::from("wasmtime")) {
          evaluator.arg("--env");
          evaluator.arg(format!("{}={}", key, value));
      } else {
          evaluator.env(key, value);
      }
  };
  ```

2.3 Dipendenze

Per funzionare correttamente, il codice richiede le seguenti dipendenze e strumenti:

- Rust e Cargo: sono chiaramente necessari il compilatore `rustc` e il corrispettivo gestore di pacchetti. Per supportare l'ambiente WebAssembly, è fondamentale installare il target `wasm32-wasip1` tramite il tool `rustup`. Quando fu aggiunto nel 2019, il target era chiamato `wasm32-wasi`, basato sul primo set di syscall[3] WASI. Con l'arrivo di WASIp2, è stato rinominato per indicare la compatibilità con la Preview1 e prossimamente la dicitura originale verrà deprecata;
- Clang e WASI SDK[4]: utilizzati per compilare i sorgenti in C/C++. Il WASI SDK fornisce le librerie e le system call necessarie per compilare programmi WebAssembly che utilizzano la WebAssembly System Interface. Inoltre, su Windows è necessario scaricare il file `libclang_rt.builtins-wasm32.a`[5] e posizionarlo nella cartella `wasip1`, da creare all'interno della directory del WASI SDK per garantire il corretto funzionamento della compilazione;
- Wasmtime: l'ambiente di esecuzione dei moduli WebAssembly;
- Eccezione Windows Defender: su Windows, una delle componenti del progetto potrebbe causare un'allerta del software antivirus. È necessario creare un'eccezione per consentire l'esecuzione senza che venga bloccata.

[3] System call: operazione che un programma utilizza per richiedere servizi o risorse direttamente dal sistema operativo.

[4] Software Development Kit: insieme di strumenti e librerie che facilitano lo sviluppo di applicazioni su una specifica piattaforma o sistema

[5] `https://github.com/jedisct1/libclang_rt.builtins-wasm32.a/tree/master/precompiled`

Capitolo 3

Conclusioni

Il lavoro presentato in questa tesi ha permesso di estendere la piattaforma Turing Arena Light integrando il supporto per l'esecuzione di valutatori WebAssembly. Questo ha aumentato la flessibilità e la portabilità della piattaforma, rendendola capace di gestire codice compilato in vari linguaggi e di eseguirlo su differenti architetture senza necessità di adattamenti specifici. Il focus sull'utilizzo di WebAssembly è stato dettato dalla sua capacità di fornire un ambiente sicuro, performante e indipendente dall'hardware, caratteristiche ideali per le esigenze di una piattaforma didattica come Turing Arena Light.
Il lavoro svolto potrebbe essere implementato anche su CodeColosseum[1], un'altra piattaforma sviluppata da Dario Ostuni, ma richiederebbe alcune modifiche e affrontare delle complicazioni. A differenza di Turing Arena Light, dove è possibile eseguire i moduli WebAssembly semplicemente chiamando Wasmtime come processo esterno, in CodeColosseum questo non è sufficiente. La complicazione principale deriva dal fatto che, in CodeColosseum, i programmi dei partecipanti competono tra loro, e per gestire queste interazioni è necessario usare la libreria di Wasmtime integrata nel codice della piattaforma. Ciò è dovuto all'assenza di uno standard universale per le pipe interprocesso tra i diversi sistemi operativi, il che richiede un approccio diverso per la gestione delle comunicazioni tra i programmi.
Il codice sviluppato per l'integrazione dei valutatori WebAssembly è disponibile nel repository ufficiale del progetto su GitHub[2], con la pull request specifica consultabile all'indirizzo `https://github.com/romeorizzi/TALight/pull/7`. Questo rappresenta un contributo aperto alla comunità, che potrà utilizzare e perfezionare ulteriormente questa implementazione; ad esempio, non appena saranno disponibili compilatori stabili per altri linguaggi, la loro integrazione nel sistema sarà agevole, ampliando ulteriormente le potenzialità della piattaforma e garantendole di rimanere aggiornata.
In conclusione, questo progetto ha evidenziato come l'integrazione di tecnologie moderne, quali WebAssembly, possa migliorare significativamente una piattaforma garantendo elevata flessibilità e portabilità.

[1] `https://github.com/dariost/CodeColosseum`
[2] `https://github.com/romeorizzi/TALight`

Bibliografia

[1] Bytecode Alliance, *wasmtime.dev*.

[2] Dario Ostuni, «Design and Evaluation of Competitive Programming Platforms for Computer Science Education», tesi di dott., Università degli Studi di Verona, 2024, URL: `https://dario.ostuni.xyz/thesis/phd.pdf`.

[3] Emscripten Community, *emscripten.org*.

[4] Alex Crichton, *The rustc book, 7.53 wasm32-wasip1*, URL: `https://doc.rust-lang.org/nightly/rustc/platform-support/wasm32-wasip1.html`.

[5] Surma, *Compiling C to WebAssembly without Emscripten*, URL: `https://dassur.ma/things/c-to-webassembly/`.

[6] Docs.rs team, *Rust documentation*, URL: `https://docs.rs/`.

[7] Steve Klabnik and Carol Nichols, *The Rust Programming Language*, No Starch Press, 2022.

[8] Thomas Steiner, *What is WebAssembly and where did it come from?*, URL: `https://web.dev/articles/what-is-webassembly`.

[9] W3C WebAssembly Community Group, *wasi.dev*.

[10] W3C WebAssembly Community Group, *webassembly.org*.

[11] W3C WebAssembly Community Group, *WASI repository*, URL: `https://github.com/WebAssembly/WASI`.

[12] W3C WebAssembly Community Group, *WebAssembly Binary Toolkit*, URL: `https://github.com/WebAssembly/wabt`.

Ringraziamenti

In primis, vorrei ringraziare il già largamente citato Dario, per le possibilità che mi ha dato e per essere stato in questi tre anni sempre un punto di riferimento, oltre che una persona che stimo e ammiro.

Passiamo poi agli amici.
Cece, il mio gemello diverso, che mi rimette in riga quando ne ho bisogno e mi stimola ogni giorno a dare di più.
Chicco, che riesce sempre con la più tagliente delle battute a strapparmi una risata, anche se probabilmente la capiamo solo io e lui.
Dodi, confidente delle conversazioni più profonde e allo stesso tempo socio di grossi sketch. Lui per un amico darebbe tutto... e sono orgoglioso di essere tra questi.
Pier, che nasconde la sua incredibile empatia dietro a un velo di spensieratezza, trova sempre la parola giusta e mi insegna costantemente a prendere le cose come vengono.
Brighe, senza il quale questo giorno sarebbe arrivato tra qualche mese, le ore passate insieme a lezione e a preparare gli esami sarebbero state infinite da solo.
Quelli della sambuca, i Seguaci di Martini, i Butei in Biblio, gli amici di Las Palmas e chiunque con cui abbia condiviso un momento, mi fate maturare come individuo, confido che i nostri legami non si deteriorino mai.

Infine, chi mi è sempre attorno, pure a chilometri di distanza... la mia famiglia.
La mia sorellina, un'amica e complice anche del più infantile divertimento, di cui mi auguro di essere, almeno qualche volta, fonte d'ispirazione.
Chi vorrei tanto fosse qui, che mi guarda dall'alto e sono certo stia sorridendo... d'altronde, qualcuno ha sempre pensato che io sia un fuoriclasse. Ciao nonni.
Dulcis in fundo, coloro che in ogni scelta mi appoggiano, senza farmi mancare nulla... mamma e papà, spero di avervi reso orgogliosi e di continuare a farlo a lungo.

GRAZIE!