Alma Mater Studiorum - Università di Bologna

Campus di Cesena

Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione di un sistema di categorizzazione delle regressioni per il compilatore Rust

Tesi di Laurea in Programmazione ad Oggetti

Relatore:

Prof. Mirko Viroli

Correlatore:

Dott. Roberto Casadei

Presentata da:

Giacomo Pasini

Anno Accademico 2019/2020

Sommario

Il linguaggio di programmazione Rust ha subito una crescita costante negli ultimi tempi e da cinque anni è votato come il linguaggio più amato dagli utenti nell'annuale sondaggio di StackOverflow [1]. Parte fondamentale del linguaggio è il compilatore, che in questo caso si occupa anche di tutti quei controlli aggiuntivi che distinguono Rust da altri linguaggi come C/C++ e che ne garantiscono la sicurezza. Il ciclo di vita del compilatore prevede il rilascio di una nuova versione ogni 6 settimane, un intervallo di tempo molto piccolo se confrontato con altri progetti come GCC, che rilascia una nuova versione ogni anno. Un tempo così ridotto rende più difficile verificare la correttezza della modifiche introdotte e quasi del tutto inutilizzato per tale motivo è il testing da parte degli utilizzatori del linguaggio, che avrebbero a disposizione una finestra molto ristretta per testare manualmente le nuove funzionalità. Per permettere un rilascio così frequente con la confidenza di non introdurre bug o regressioni viene utilizzato Crater. Crater, prima di una nuova release, testa automaticamente tutto il codice Rust ottenibile online su GitHub e su *crates.io*, il registry pubblico ufficiale. I risultati ottenuti sono poi presentati ai team di rilascio e del compilatore per sistemare eventuali criticità trovate. Il progetto di questa tesi si inserisce in tale contesto per migliorare l'analisi dei dati ottenuti e automatizzare il lavoro di categorizzazione degli errori. In tale modo si semplificano le operazioni di triage delle problematiche individuate e si alleggerisce il carico sugli sviluppatori. In particolare, si suddividono gli errori in base al codice riportato e si analizzano le relazioni di dipendenza tra di essi.

[1] Most Loved Languages

Indice

Elenco delle figure

Capitolo 1

Contesto di lavoro

Sommario

In questo capitolo si analizzano gli aspetti preesistenti l'intervento sull'applicativo in questione e si delinea il contesto d'uso di tale sistema.

1.1 Il linguaggio Rust

Rust è un linguaggio relativamente recente, la versione 1.0 è stata rilasciata nel maggio 2015, orientato principalmente alla sicurezza ed alle elevate prestazioni. Il linguaggio si pone in un certo senso come sostituto di C/C++, in quanto promette efficienza comparabile ma con maggiore attenzione agli errori che potrebbero essere commessi nella stesura del programma, come ad esempio gli errori derivati dalla mancanza di *memory safety*. Obbiettivo del linguaggio è anche supportare la realizzazione di sistemi paralleli, individuando a compile time la presenza di costrutti potenzialmente pericolosi come quelli che possono introdurre *data races*.

Alcune delle feature che distinguono Rust sono:

- Ogni valore ha una sola variabile *owner*. Quando l'*owner* esce dallo scope, il valore viene rilasciato e la memoria liberata. In questo modo la memoria viene gestita automaticamente senza bisogno di un garbage collector.
- È possibile prestare i valori invece di trasferire l'ownership. Il compilatore si occupa di controllare che in uno stesso scope esista un solo riferimento mutabile o un numero illimitato di riferimenti immutabili. Questo previene *data races*, in quanto vieta di avere due puntatori allo stesso dato che potenzialmente possono modificarlo. Controlli aggiuntivi verificano che il tempo di vita dei valori prestati sia compatibile con l'uso che di essi si fa.
- In generale, le specifiche del linguaggio garantiscono l'assenza di data races, buffer overflows, stack overflows e accessi a memoria non inizializzata o deallocata.

Per parte del linguaggio è stata formalmente verificata l'osservazione delle promesse sopra esposte, e si sono inoltre illustrate le caratteristiche che rendono sicura una sua estensione, come l'aggiunta di una libreria, anche in relazione all'uso di codice `unsafe`, che permette di eludere alcuni controlli del compilatore [13].

Rust è quindi particolarmente interessante poichè promette di evitare, per design, molti dei bug[1] che attualmente affliggono codice scritto in linguaggi come C/C++, conservando al contempo una velocità comparabile. Non facendo uso di un garbage collector è inoltre adatto a sistemi real time.

Recentemente molte aziende stanno iniziando ad utilizzarlo all'interno dei loro prodotti, tra le più famose possiamo citare Mozilla, Amazon, Google, Facebook e Microsoft, che l'ha definito come *The Industrys Best Chance at Safe Systems Programming* [2] [11].

Il progetto è nato internamente a Mozilla ma ormai la maggior parte delle persone che contribuiscono sono volontari o esterni, con solo una manciata di persone stipendiate da Mozilla per lavorare, part-time o full time, a Rust. Un aspetto fondamentale, soprattutto ai fini di questa tesi, è che il compilatore Rust ha un ciclo di rilascio di sole 6 settimane. Questo significa che ogni 6 settimane esce una nuova versione del compilatore che aggiunge nuove feature. La scelta di un ciclo di vita così corto nasce dal fatto che la maggior parte degli sviluppatori del compilatore sono volontari, e che se una feature non è pronta la prossima finestra disponibile per il rilascio non è un anno dopo ma un mese e mezzo dopo. Questo permette di non avere particolare pressione e fretta per sviluppare nuove funzionalità, in quanto un ritardo di 6 settimane raramente inficerà pesantemente l'esperienza d'uso del linguaggio.

Con un ciclo di rilascio così veloce diventa quindi molto importante predisporre un'infrastruttura di testing che assicuri che il software prodotto raggiunga determinati standard qualitativi, velocizzi il lavoro degli sviluppatori e automatizzi il processo il più possibile. Ovviamente la test suite del compilatore è una buona partenza per controllare la correttezza delle modifiche proposte ed una consistente infrastruttura si occupa di gestire la continuous integration (CI) nel repository GitHub. Per ogni *pull request* effettuata sul repository GitHub, 60 macchine virtuali si occupano di eseguire i numerosi test lavorando dalle 3 alle 4 ore. Questo però non è sufficiente in quanto la complessità di un linguaggio non può essere racchiusa facilmente in una test suite. Risulta anche difficile chiedere agli utenti di provare le nuove versioni e riportare eventuali problemi riscontrati, in quanto avrebbero a disposizione un tempo abbastanza limitato per tale valutazione.

[1] Microsoft Security Response Center

[2] Ryan Levick - Rust at Microsoft

1.2 Crater

Proprio per incrementare la quantità di codice testato per ogni nuovo rilascio nasce **Crater**. Crater è un progetto che si occupa di raccogliere tutto il codice Rust disponibile su *crates.io* (il registry ufficiale per le librerie Rust) e su GitHub, per poi usarlo come base per il testing di una nuova versione del compilatore. Nello specifico vengono eseguite le stesse operazioni sia con la versione corrente del compilatore, sia con la nuova versione di cui si vuole testare la correttezza. In seguito si analizzano quindi i risultati ponendo l'attenzione soprattutto sulle differenze tra le due versioni. Si chiamano quindi *regressioni* quegli errori del software, particolarmente critici e da evitare il più possibile, che si verificano su nuove versioni e non erano presenti precedentemente. In questo modo si riesce a verificare il comportamento del compilatore in un insieme molto numeroso e variegato di progetti Rust, con le più disparate provenienze e stili di programmazione [1].

Si illustra ora a grandi linee il funzionamento e l'architettura di Crater, in modo poi da poter valutare più approfonditamente le scelte effettuate in fase di progettazione. Nel farlo si definiranno anche una serie di termini di uso comune all'interno del progetto. Per ogni termine è stato scelto di riportare anche il nome in inglese poiché è quello che viene utilizzato nel codice, nella documentazione e nei canali di comunicazione.

1.2.1 Terminologia

Cargo

Cargo è il package manager di Rust. Esso viene utilizzato sia per compilare ed eseguire i progetti, rispettivamente con `cargo build` e `cargo run`, sia per eseguire unit e integration test, attraverso `cargo test`. Tramite il file di configurazione `Cargo.toml` è possibile indicare le dipendenze utilizzate dall'applicazione, che verranno automaticamente scaricate e compilate. In aggiunta è possibile installare *plugin* aggiuntivi, che ne estendono o modificano le funzioni. Uno tra i più famosi è *clippy*, che contiene una collezione di lint per individuare errori comuni e migliorare la qualità del codice.

Esperimenti

Un esperimento (**experiment**) è l'unita di lavoro basilare di Crater. Esso definisce le varie azioni da effettuare e la lista di crate [3] da utilizzare. Tra i vari dati di configurazione alcuni tra i più importanti sono:

- Modalità (**mode**): Specifica le azioni da effettuare per ogni crate. Ad esempio se il cambiamento introdotto non modifica la fase di *codegen* sceglierò la modalità `CheckOnly` che corrisponde ad eseguire solamente `cargo check` per ogni crate. Analogamente, la modifica apportata potrebbe riguardare soltanto alcune parti del compilatore (o dell'ecosistema Rust in generale) come clippy e a tale scopo userei la modalità `Clippy` che per ogni crate esegue `cargo clippy`.

  La modalità più comune (e completa, per quanto riguarda il compilatore), `BuildAndTest`, esegue invece `cargo build` e `cargo test`: compila ed esegue i test per ognuna delle crate indicate.

  Di seguito si chiamerà anche più brevemente con *test* l'insieme dei comandi specificati dalla modalità.

- Le versioni del compilatore da utilizzare (**toolchain**): Specifica nel dettaglio le versioni del compilatore da usare. La prima, chiamata anche *start*, viene utilizzata come riferimento (spesso si usa la versione corrente del compilatore) e la seconda, *end*, identifica la versione da testare.

Report

Viene chiamato **report** l'insieme di documenti prodotti da Crater al termine dell'esecuzione di un esperimento. Esso contiene la lista di crate divise per risultato di test consultabile online attraverso pagine html, la configurazione dell'esperimento, i **log** prodotti dall'esecuzione dei test per ogni crate ed un file JSON con tutti i dati.

Le crate sono divise nelle seguenti categorie sulla base del confronto dei risultati ottenuti con le due toolchain. Qui sono riportate descrizioni non troppo rigorose, se si vogliono conoscere con precisione le regole che determinano l'appartenenza alle diverse categorie si può fare riferimento al repository ufficiale.

[3] In ambito Rust si chiama *crate* una unità di compilazione (solitamente quindi una libreria o un binario)

- `Regressed`: la toolchain *end* presenta un risultato *peggiore* della toolchain *start*. Ad esempio si passa da un'esecuzione corretta dei test della crate ad un errore in fase di compilazione.

- `Fixed`: al contrario delle crate regredite, il risultato *migliora* con la toolchain *end*.

- `Error`: l'esecuzione dei test per la crate ha riscontrato un errore non atteso.

- `Broken`: l'esecuzione dei test fallisce per problematiche legate alla crate stessa e non dipendenti da errori nel compilatore. Ad esempio perché presenta un file di configurazione `Cargo.toml` non valido o utilizza dipendenze locali o private.

- `Skipped`: quelle crate che pur facendo parte della lista iniziale sono state saltate per problemi noti o perchè non è stato possibile trovare una versione utilizzabile.

- `SameTestPass`, `SameTestFail`, `SameBuildFail`: le crate presentano lo stesso risultato con entrambe le toolchain.

- `SpuriousFixed`, `SpuriousRegressed`: analoghe a `regressed` e `fixed` ma in cui le cause degli errori in fase di test sono classificate come *spurie*, come ad esempio il superamento del limite di memoria assegnato ad ogni crate.

1.2.2 Architettura

Crater è dispiegato in cloud attraverso una architettura client-server. I client, chiamati **agent**, si occupano del testing vero e proprio sulle crate mentre il server svolge il restante lavoro, come ricezione e validazione dei comandi dall'esterno, distribuzione del lavoro tra gli agent e generazione del report una volta completato l'esperimento. Attualmente vengono utilizzate 5 macchine virtuali, di cui una svolge la funzione di server coordinatore, come è possibile vedere in figura 1.1.

Il server si occupa quindi delle comunicazioni con l'esterno, in quanto è l'unico ad essere esposto su una rete pubblica. La comunicazione tra agent e server avviene attraverso endpoint http in cui il messaggio è rappresentato in formato JSON. Ogni agent condivide un token di autenticazione con il server coordinatore che gli permette di autenticare, e autorizzare, le richieste effettuate.

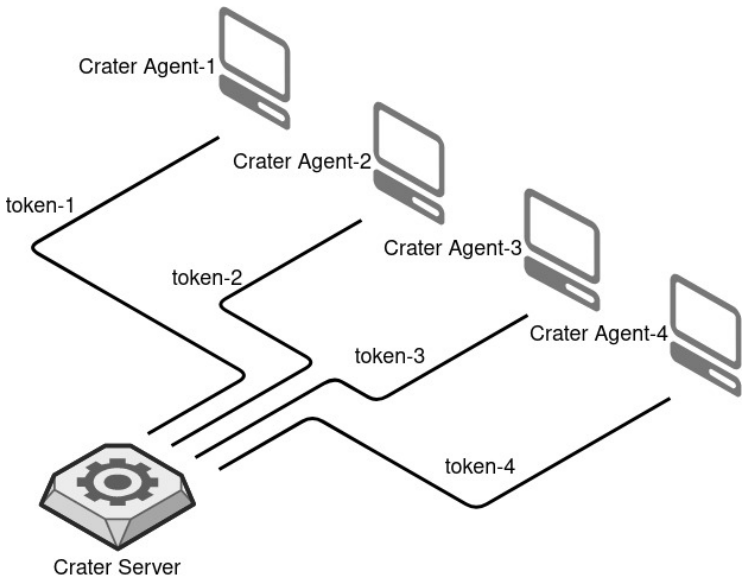

Figura 1.1: Schema dell'architettura client-server

Nel corso di questa relazione verranno completamente tralasciati alcuni aspetti di Crater, come l'esecuzione parallela degli esperimenti, poiché giudicati poco importanti ai fini di questa modifica. Si avvisa quindi che per una panoramica completa del programma è necessario consultare il repository ufficiale.

1.2.3 Rustwide

Rustwide è una libreria che si occupa dell'esecuzione di comandi in un ambiente Rust riproducibile. Inizialmente era codice che faceva parte di Crater ma poi è stato estratto come libreria indipendente per permetterne il riuso in altri progetti come *docs.rs*.

Per rendere riproducibile il risultato, e per questioni di sicurezza, vengono utilizzati container Docker in cui si restringe l'accesso alla rete. L'immagine di base che si utilizza per i vari container, chiamata *crates-build-env* e disponibile su DockerHub, è basata su Ubuntu 20.04 e contiene tutte le dipendenze più comuni come libc o OpenSSL.

In particolare, analizziamo ora quelle parti della libreria più utilizzate da Crater e che sarà necessario poi modificare in parte.

Ambiente di build

Rustwide mette a disposizione funzionalità apposite per poter compilare e testare le crate in una specifica directory. In particolare, data la crate, la toolchain, la directory e

la funzione `test_fn` da eseguire sulla crate, il modulo `src/build` si occuperà di scaricare la crate e la toolchain specificate nella directory. In seguito, dopo aver montato tale percorso all'interno di un container Docker basato sull'immagine *crates-build-env* accennata in precedenza, esegue la funzione `test_fn` all'interno del container.

Interazione con i comandi

Il contenuto della funzione `test_fn` è spesso rappresentato da un insieme di comandi Rustwide. Realizzati tramite la struct `Commmand`, essi estendono le funzionalità dei comandi della libreria standard di Rust e permettono facilmente l'integrazione con Docker. Alcune tra la funzionalità più utilizzate da Crater sono le seguenti:

- **Limitazione dell'uso di risorse da parte del comando**. Quando si eseguono molte build concorrenti si vuole evitare che una crate particolarmente pesante, o appositamente costruita per attaccare l'infrastruttura di testing, influenzi negativamente le altre crate. Per tale motivo Crater impone di default un limite di 1.5 Gb di memoria utilizzabile durante l'esecuzione dei test.

  In aggiunta, sempre per ragioni di sicurezza, non sarà possibile per le crate utilizzare la rete per comunicare con l'esterno.

- **Log dell'output**. Si può abilitare/disabilitare il logging dell'output del comando. Questo viene utilizzato in congiunzione con un'altra funzione Rustwide chiamata `capture` per salvare l'output dell'esecuzione dei test per ogni crate.

- **Interazione real-time con l'output del processo**. Si può specificare una funzione F che verrà eseguita per ogni riga di output, utile se si vuole analizzare l'output del comando senza salvarlo interamente in memoria.

1.3 Scenario d'uso e motivazioni

Crater si inserisce nel ciclo di sviluppo del compilatore Rust sia come valutazione della correttezza precedente un nuovo rilascio sia come valutazione della fattibilità di determinate modifiche che si pensa possano impattare in modo significativo codice esistente.

In entrambi i casi la modalità d'uso, rappresentata anche in figura 1.2, è la seguente:

- Si invoca Crater, solitamente nella stessa *pull request* che apporta la modifica o che promuove il branch per la release, attraverso un bot GitHub. Un esempio di invocazione che crea un esperimento con nome *foobar*, modalità di default `BuildAndTest` e usando `stable` e `beta` come toolchain è il seguente:

  ```
  @craterbot run name=foobar start=stable end=beta
  ```

  `stable` e `beta` identificano rispettivamente la versione stabile corrente del compilatore e la prossima versione del compilatore ancora da testare e rilasciare e sono gli stessi nomi utilizzati in tutti gli altri strumenti dell'ecosistema Rust come `cargo` e `rustup`. In alternativa è possibile specificare un commit specifico su GitHub.

  Una reference completa per l'interazione con il bot è disponibile qui. Viene inoltre effettuato un controllo dei permessi per assicurarsi che uno sconosciuto non possa sovraccaricare l'infrastruttura.

- Al termine dell'esecuzione dell'esperimento Crater invia una notifica tramite messaggio nel thread GitHub contenente il link al report generato.

- I membri del *Release Team*, nel caso di nuovi rilasci, o l'autore della PR, nel caso di studio di impatto di modifiche particolari, analizzano manualmente il report e, soprattutto per le regressioni, identificano le cause e segnalano i problemi tramite `issue` nel repository GitHub.

- Il *Compiler Team* si occupa poi di sistemare le problematiche evidenziate.

Una delle parti più lunghe e tediose del processo è analizzare uno a uno i log delle crate regredite per identificarne le cause. In aggiunta, se una crate regredisce, tutte le crate che la utilizzano come dipendenza potrebbero regredire a loro volta (questo non succede se tali progetti già non erano funzionanti con versioni precedenti del compilatore), generando quindi confusione nel report. Scopo di questo progetto è quindi quello di semplificare tale processo, categorizzando automaticamente per quanto possibile i risultati di un esperimento, individuando specialmente la cause dell'errore e le cosiddette **root regression**, cioè quelle crate la cui causa di regressione è interna ad esse e non dovuta alle loro dipendenze. L'elenco delle dipendenze inverse (**reverse dependencies**) è comunque importante per

poter valutare l'impatto che tale crate, e quindi regressione, ha su codice esistente. Se ad esempio la regressione è causata da un warning che è diventato un errore nella nuova versione del compilatore è utile sapere se tale problema si presenta in una libreria largamente utilizzata o in un progetto potenzialmente dimenticato che nessuno usa più.

Figura 1.2: Processo di analisi di modifiche al compilatore tramite Crater

1.3.1 Processo completo

Visti a grandi linee i vari pezzi proviamo a dare una visione d'insieme con un diagramma di sequenza in figura 1.3 che illustra gli attori in gioco e le varie fasi nel completamento di un esperimento.

Questo processo può richiedere un lasso di tempo non indifferente. Il tempo effettivo può dipendere dal numero di agent disponibili ma a grandi linee un esperimento in modalità `Check` impiega circa 3 giorni ed uno in modalità `BuildAndTest` circa 1 settimana. Il numero delle crate testate in questi esperimenti è al momento di poco superiore alle 110.000 unità [4]. Una volta realizzate le correzioni necessarie si può inoltre decidere di rieseguire il test soltanto sulla lista di crate regredite prodotta automaticamente da Crater, riducendo in questo modo considerevolmente il tempo di attesa.

[4] Crater report for pr-73461

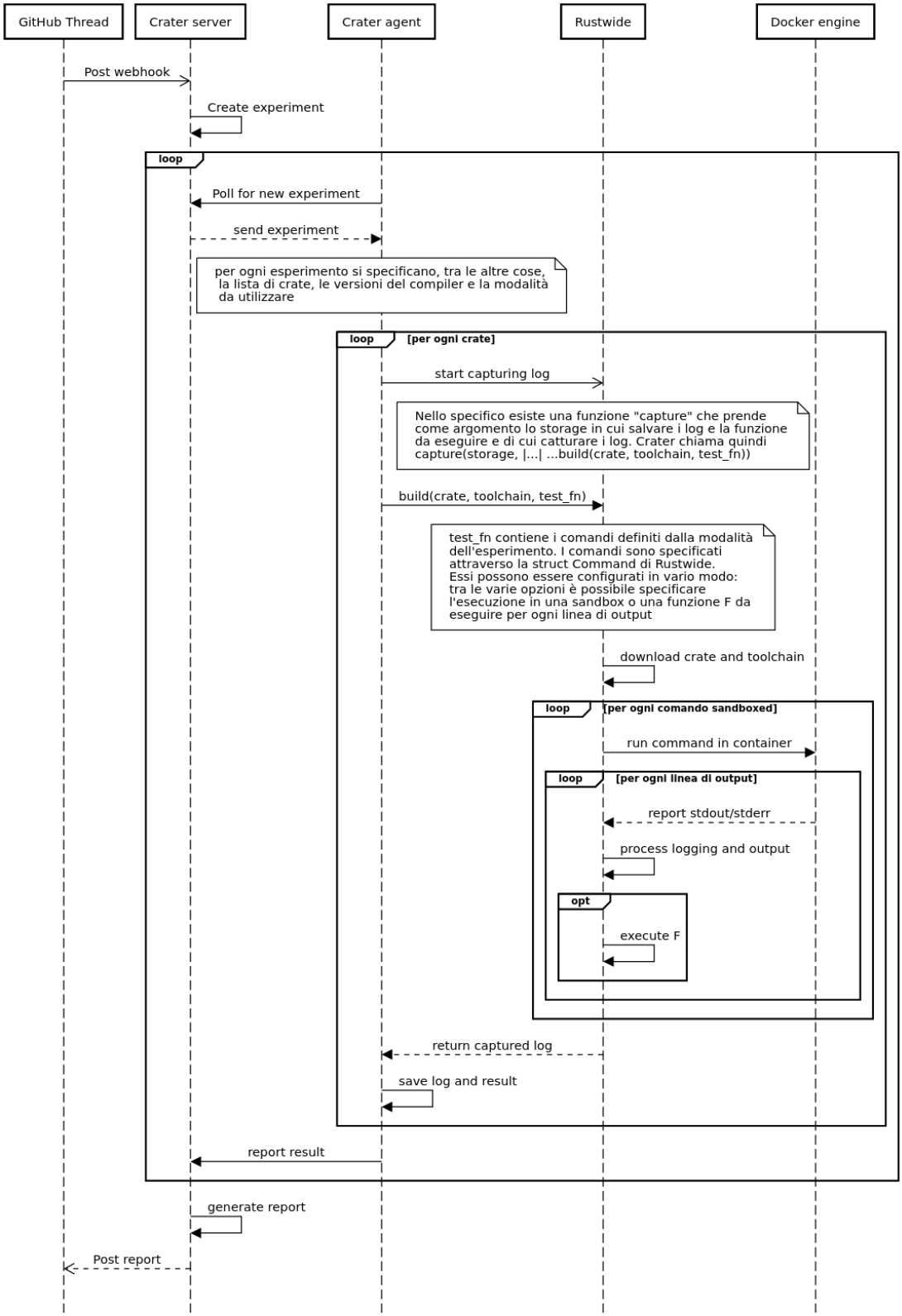

Figura 1.3: Diagramma di sequenza dello svolgimento di un esperimento

1.4 Comparazione del modello di sviluppo con altri linguaggi

Si analizzano in questa sezione i processi di sviluppo di altri progetti in comparazione a quello Rust. In particolare si è interessati alla modalità con cui viene verificata la correttezza delle nuove versioni.

1.4.1 Rust

Rust adotta un modello di sviluppo con rilascio ogni 6 settimane. Sono inoltre presenti tre canali di release: **Stable**, **Beta**, **Nightly**.

Ogni notte, una nuova versione di nightly viene rilasciata dal codice del branch master del repository ufficiale. Ogni sei settimane il contenuto del branch beta viene aggiornato con le feature di nightly, escludendo quelle giudicate ancora instabili. Nelle successive sei settimane eventuali correzioni di bug sono applicate al branch beta, ma lo sviluppo di nuove feature avviene solo su nightly. Al termine di questo periodo il branch beta viene promosso a stable. Allo stesso tempo una nuova versione di beta è ottenuta da master.

I metodi utilizzati da Rust per testare la correttezza delle nuove versioni sono:

- Testing manuale da parte degli utenti. La maggior parte utilizza stable o nightly.
- Test suite del compilatore. Nel repository sono predisposti strumenti per il controllo automatico dei test prima della modifica dei contenuti. Tali test vengono eseguiti su tutte le piattaforme specificate come *Tier 1*, come `x86_64-unknown-linux-gnu`.
- Crater, il cui ruolo all'interno del processo è già stato analizzato.
- *rustc-perf*: questo tool esegue regolarmente ed in modo automatico benchmark per individuare regressioni o cambiamenti delle performance del compilatore.

1.4.2 GCC

GCC adotta un modello di sviluppo con ciclo di rilascio annuale di nuove versioni [15]. Le varie fasi che compongono lo sviluppo sono:

- **Stage 1**: durante questa fase, *feature driven*, vengono proposti e implementati i cambiamenti più importanti da introdurre. Questa fase dura almeno 4 mesi.

- **Stage 2**: non più utilizzata.
- **Stage 3**: a partire da questa fase non si introducono più nuove funzionalità. I soli cambiamenti accettati sono correzione di bug e porting che non richiedono la modifica di altre parti del compilatore. Questa fase dura 2 mesi.
- **Stage 4**: le uniche modifiche ammesse in questo periodo sono correzione di regressioni o bug particolarmente critici. Dura fino all'apertura dello stage 1 della prossima versione del compilatore.

I metodi attraverso cui GCC testa la correttezza delle versioni sono i seguenti [14]:

- Test suite automatizzata del compilatore. I risultati e le regressioni sono condivisi nelle apposite mailing list.
- Build del compilatore ed esecuzione di run apposite per individuare le regressioni in modalità manuale da parte degli utenti.

Tra le idee proposte per testing aggiuntivo possiamo trovare:

- Implementazione di un sistema che notifichi gli sviluppatori quando sottomettono patch che introducono problemi nelle build.
- Compilazione ed esecuzione dei test di alcune delle librerie più famose, come Boost o Qt.
- Esecuzione manuale di benchmark per evidenziare regressioni nelle performance.

1.4.3 Python

Python ha recentemente cambiato il suo modello di sviluppo accorciando il ciclo di rilascio [16]. Esso avverrà, per la versione 3.9, nel seguente modo:

- Durante i primi 7 mesi, denominati *alpha*, si introducono le nuove feature.
- Nei successivi 3 mesi, denominati *beta*, si introducono solamente bug fix e non si sviluppano più nuove funzionalità.
- Nei restanti 2 mesi si perfezionano ulteriormente i candidati alla realease e il periodo si conclude con il rilascio della nuova versione.

La correttezza delle nuove versioni viene controllata principalmente attraverso l'esecuzione automatica della test suite. È inoltre presente anche un sistema in grado di verificare che non ci siano regressioni a livello di performance. Ulteriore supporto nel testing viene fornito dagli utenti che provano manualmente le versioni prima del rilascio.

Anche in questo ambito ci sono state proposte, nessuna per ora accettata ufficialmente, di migliorare l'integrazione con un ristretto numero delle librerie più utilizzate, per ovviare al problema che alcuni progetti molto famosi producono versioni compatibili con la nuova release del linguaggio settimane o mesi in ritardo [5].

1.4.4 Typescript

Typescript è uno dei linguaggi con il ciclo di rilascio più veloce, in quanto viene pubblicata una nuova versione ogni 3 mesi. Recentemente si è allungato il ciclo di sviluppo, da 2 a 3 mesi, per meglio testare il materiale prodotto [18].

Anche in questo caso la test suite del compilatore riveste grande importanza nella verifica della correttezza delle modifiche e sempre molto utili sono i report manuali dei bug da parte degli utenti.

In aggiunta, questo progetto adotta una strategia simile a Crater. Il repository DefinitelyTyped contiene infatti le dichiarazioni dei tipi per molteplici progetti (comunque un ordine di grandezza inferiore rispetto a quelli testati da Crater), che vengono utilizzati dal team di sviluppo per rafforzare la robustezza dell'ecosistema in relazione ai cambiamenti introdotti nel linguaggio [17].

1.4.5 OpenJDK

OpenJDK segue un ciclo con rilascio ogni 6 mesi di una nuova versione. Le varie fasi nello sviluppo del codice sono le seguenti [20]:

- Sviluppo di nuove funzionalità durante i primi tre mesi di vita della nuova versione.
- Successivamente si ha la fase denominata **Rampdown Phase 1**, dalla durata circa di un mese. Non si introducono più nuove funzionalità a partire da questo momento, e gli sforzi sono invece dedicati alla risoluzione dei bug individuati.

[5] PEP 608

- Alla **Rampdown Phase 1** segue la **Rampdown Phase 2**, sempre dalla durata indicativa di un mese. In questa fase si correggono soltanto i bug particolarmente importanti che sono stati introdotti con questa release e che ne compromettono la validità.

- Infine si giunge alla **Release-Candidate Phase**, in cui sono trattati solamente i bug più critici rimanenti. Durante le fasi precedenti, RDP 1 e RDP 2, potrebbero essere aggiunte piccole funzionalità mancanti, seppur soppesando attentamente la scelta. Tale possibilità non è invece presente in questa fase, che precede il rilascio della nuova versione.

La correttezza del software prodotto viene controllata con numerosi test integrati, microbenchmarks e con appositi test per verificare l'assenza di regressioni conosciute. In aggiunta, il report dei bug riscontrati dagli utenti gioca un ruolo importante.

1.4.6 Altri esempi

In generale, analizzando un numero consistente di progetti si è visto come numerosi problemi possano essere evitati utilizzando un ciclo di sviluppo con rilasci ben definiti. In aggiunta, suddividendo le feature in più versioni maggiormente contenute, risulta più facile il testing delle modifiche, in quanto si riescono ad isolare meglio i cambiamenti introdotti [21].

Non si sono inoltre trovati progetti rilevanti condotti alla stessa scala di Crater. Un elemento, non sempre presente in altri linguaggi, che aiuta questo tipo di approccio è la standardizzazione dei tool utilizzati per la compilazione e per l'esecuzione dei test, in questo caso **cargo**.

1.5 Analisi dell'efficacia dell'utilizzo di Crater

Si riportano ora alcuni dati per cercare di determinare l'impatto che Crater ha all'interno del processo di sviluppo, sia per quanto riguarda la capacità di trovare casi non coperti dalla test suite del compilatore, sia per valutare la sua efficacia nel rappresentare il reale uso del linguaggio.

Analizzando le release dalla 1.35 alla 1.38, si sono ricavate per ognuna il numero di righe di codice cambiate, il numero di crate regredite individuate da Crater (1), e quindi

Ver.	LOC cambiate	Regr. Crater (1)	Regr. codice valido (2)	Altre regr. (3)
1.35	102163	73	0	5
1.36	136306	750	2	2
1.37	139667	34	3	0
1.38	206324	616	2	3

Tabella 1.1: Analisi delle regressioni introdotte dalle modifiche al linguaggio

sfuggite ai test del compilatore, ed il numero di regressioni riportate dagli utenti dopo il rilascio ufficiale. Le regressioni individuate dopo la pubblicazione della nuova versione sono state ulteriormente categorizzate a seconda che rigettassero codice valido (2), oppure riguardassero peggioramenti di prestazioni o di messaggi d'errore (3).

Si può notare come, considerando che al tempo il numero di crate testate da Crater si aggirava intorno alle 60000 unità, le regressioni individuate interessavano in certe occasioni anche più di un punto percentuale delle crate testate, una porzione sicuramente non insignificante. Si può infatti calcolare che, assumendo di individuare nuove regressioni nel 1% di crate testate ad ogni versione, in un anno di sviluppi circa il 10% dei progetti Rust diventerebbe inutilizzabile.

Crater viene inoltre utilizzato spesso per valutare l'impatto di cambiamenti nel comportamento del compilatore riguardo casi non validi di utilizzo del linguaggio. Prendiamo ad esempio la pull request 73461 al repository ufficiale Rust. Essa applica controlli aggiuntivi nel caso di attributi utilizzati in posizioni non valide, precedentemente ignorati senza notifica all'utente. In questo caso non sarebbe facile valutare quanto questo impatti l'ecosistema delle librerie, in quanto bisognerebbe analizzare manualmente ognuna delle crate. Utilizzando Crater invece si raccolgono informazioni utili a capire come tali casi non validi siano effettivamente usati in codice reale, semplificando la decisione sul comportamento da tenere.

1.6 Obiettivo dell'intervento

Obiettivo principale dell'intervento è quello di semplificare il lavoro di triage delle problematiche risultate dall'esecuzione di un esperimento di Crater. Più in dettaglio, sarà necessario identificare le relazioni di dipendenza che esistono tra le crate che presentano errori e classificare questi ultimi in relazione al codice emesso dal compilatore. Infine, si dovrà modificare il report fornito per includere anche tali informazioni e per aumentare

l'integrazione con i canali di comunicazione solitamente usati dal team di sviluppo di Rust.

Per quanto riguarda il back-end, è necessario potenziare il modo con cui sono ottenuti i risultati per ogni crate per poter permettere di analizzare direttamente l'output del processo ed estrarre i messaggi del compilatore. In aggiunta, sarà necessario rappresentare adeguatamente all'interno dell'applicativo le informazioni raccolte, non essendo alcuni elementi del dominio al momento sufficienti a racchiudere tutte le possibili varianti che si possono presentare durante l'esecuzione dei test.

Successivamente si interverrà sul front-end, modificando il modo in cui sono presentate le informazioni. Nel nuovo scenario d'uso sarà necessario, oltre a mostrare nel report esistente in formato HTML le nuove analisi condotte sui dati, produrre anche un report in formato Markdown, per semplificare l'interazione con i canali di comunicazione come Zulip e GitHub. Tali modifiche dovranno essere precedute da una riorganizzazione del codice che si occupa della creazione del report, che non è stato pensato per la produzione di molteplici formati.

Capitolo 2

Analisi

Sommario

La progettazione necessita di una fase precedente di definizione dei requisti. Tale processo non è stato effettuato per intero prima della progettazione ma funzionalità differenti sono state discusse in tempi differenti ed una stessa funzionalità è stata trattata con approfondimento crescente nel tempo.

Nel dettaglio, i requisiti delineati sono stati concordati sul canale pubblico ufficiale Zulip di Rust, soprattutto con membri del *Release* e *Compiler team* e con il maintainer attuale di Crater. In questo modo si sono raccolte le preferenze degli utilizzatori finali del software mediandole con la fattibilità delle modifiche al codice esistente.

Si è scelto inoltre di non utilizzare un procedimento Waterfall in quanto difficilmente si può comunicare con esattezza uno scenario d'uso abbastanza complicato come quello di Crater. Come conseguenza, dopo una prima analisi e soprattutto per quanto riguarda la parte di visualizzazione dei risultati, veniva nuovamente discussa la bozza realizzata con i

committenti, modificando al bisogno dettagli a cui prima non si era fatto caso e non erano stati definiti sufficientemente.

2.1 Requisiti

Il sistema commissionato mira ad una classificazione più accurata delle cause di fallimento dei test per ogni crate per velocizzare il lavoro di analisi delle problematiche evidenziate.

In questa sezione si evidenziano i requisiti così come richiesti dagli utenti di Crater. Alcuni di essi sono stati considerati come opzionali sin dall'inizio del processo, tenendo conto sia delle difficoltà presentate sia delle migliorie che apportano.

2.1.1 Requisiti funzionali

- Il sistema deve essere in grado di classificare più approfonditamente le cause del fallimento dei test per ogni crate. In particolare, si è deciso di non analizzare il risultato di `cargo test` in quanto i vari errori sono difficilmente confrontabili tra di loro. Di conseguenza, ci si aspetta che l'output dei comandi `cargo check`, `cargo build`, `cargo clippy`, `cargo doc` sia utilizzato per trovare le cause d'errore per ogni crate.

- Il sistema deve essere in grado di differenziare le **root regression**, cioè le crate che falliscono per cause interne, e le crate che invece non compilano a causa di dipendenze.

- Il report generato deve mostrare queste classificazioni e velocizzare il lavoro degli utenti.

Requisiti funzionali opzionali

- Il report include anche un file *Markdown* contenente soltanto le categorie più importanti per poter essere più facilmente integrato all'interno di GitHub.

- Viene comunicato all'utente se esiste una versione più recente della crate testata. Essendo che un esperimento completo può richiedere anche una settimana per essere ultimato è possibile che nel frattempo siano uscite nuove versioni delle crate testate. È quindi utile in tale caso avvisare che esiste una nuova versione che potrebbe aver sistemato eventuali errori individuati da Crater.

- Si da la possibilità all'utente di raggruppare ulteriormente le crate in categorie definite manualmente.

- Viene generata una lista dei maintainer delle crate regredite per poterli più facilmente contattare.

2.1.2 Requisiti non funzionali

- Il sistema deve essere integrato all'interno del codice esistente e non come progetto esterno.

- Il costo in performance dell'analisi delle cause deve essere il più possibile contenuto, in quanto impatterebbe in modo considerevole i tempi di testing che già sono sostanziosi.

2.2 Raffinamento dei requisiti

Ulteriori confronti hanno poi portato ad una più specifica definizione del comportamento atteso.

- La classificazione ulteriore delle cause verrà fatta sulla base dei codici d'errore emessi dal compilatore e sulla dipendenza da determinate crate che presentano problematiche. In aggiunta sarà necessario individuare anche i cosiddetti *ICE*, *internal compiler error*, i bug del compilatore, particolarmente importanti e delicati.

- Nel report html, per ogni codice d'errore dovrà essere riportata la lista delle crate in cui compare. Questo vuol dire che se una crate presenta più di un errore essa dovrà essere replicata in tutte le liste.

- Analogamente, per ogni crate dovrò essere riportata la lista di crate che falliscono la compilazione a causa sua. Di nuovo, si dovrà replicare la crate se essa non compila a causa di più crate.

- Nel report Markdown ogni root regression andrà indicata una volta soltanto con l'elenco completo degli errori presentati. Le dipendenze invece saranno replicate per ognuna delle crate che le ha causate.

2.3 Analisi del comportamento del compilatore

Prerequisito essenziale per ognuna delle modalità elencate, tranne `Doc`, è la compilazione della crate corrente e delle sue dipendenze. Se la modalità scelta è `CheckOnly` non sarà prodotto alcun eseguibile ma saranno comunque rilevati eventuali errori. Questo significa che errori nelle dipendenze della crate o direttamente nella crate corrente possono sempre influenzare negativamente il risultato del test ed è necessario individuare tali problemi. Ad esempio, anche se siamo interessati ai lint emessi da `cargo clippy` con la modalità `Clippy`, è possibile che il test fallisca a causa di un errore di compilazione.

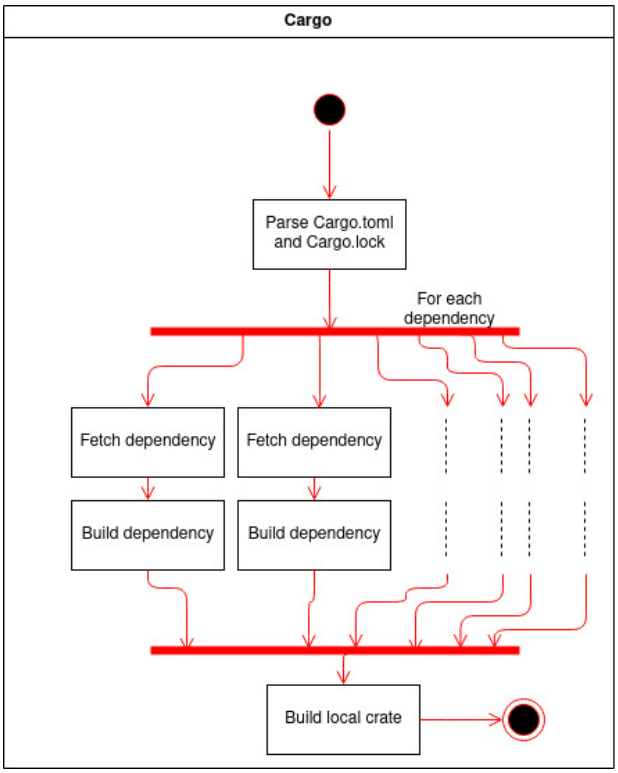

Figura 2.1: Diagramma di attività della compilazione di una crate

Nel diagramma di attività in figura 2.1 è rappresentato il funzionamento semplificato di cargo. Non è effettivamente vero che tutte le dipendenze possono essere compilate allo stesso momento, in quanto potrebbero esserci relazioni di dipendenza ulteriori tra di esse. Tale aspetto non è però importante ai fini di questa analisi e si è preferito mostrare come, nei limiti del possibile, cargo compili parallelamente le diverse librerie. Per questo motivo può capitare che il compilatore emetta errori relativi a crate differenti prima di interrompere la compilazione. È necessario quindi tenere traccia di tutte le dipendenze che

causano problemi. Analogamente, è possibile che il compilatore emetta più di un errore proveniente dalla stessa crate. Anche in questo caso è necessario salvare tutti i dati forniti.

I codici emessi dal compilatore sono raccolti nel rust-compiler-error-index e sono al momento rappresentati con una sigla alfanumerica di 5 caratteri. Nella modalità `Clippy` vengono però emessi anche gli errori di `clippy`, i quali hanno invece solitamente una descrizione più discorsiva, come `clippy::print_with_newline`. In ogni caso, gli errori individuati non hanno sempre un codice associato. Alcuni, come quelli legati alla sintassi presenti anche nella crate locale *build-fail*, ne sono al momento sprovvisti.

Capitolo 3

Progettazione

Sommario

In questo capitolo si mostrano le scelte effettuate in fase di design del sistema. In particolare, vengono illustrati nel dettaglio i componenti necessari a soddisfare i requisiti individuati in fase d'analisi e come essi interagiscono tra loro.

3.1 Analisi dei log per l'individuazione delle cause

Primo step fondamentale per l'implementazione delle nuove funzionalità è quello di individuare le cause degli errori in fase di test. A tale scopo si è deciso di utilizzare la capacità di *cargo*, il package manager di Rust, di produrre output in formato JSON. In questo modo risulta molto più semplice l'estrazione dei dati interessanti. Se *cargo* viene invocato con `--message-format=json`, le seguenti informazioni saranno presenti in output:

- Errori e warning del compilatore
- Gli artefatti prodotti, emessi per ogni step di compilazione
- Risultati derivati dagli script di build

Per l'individuazione delle cause d'errore sono utili soltanto i messaggi del compilatore, mentre le restanti informazioni sono comunque importanti nel log per capire il contesto in cui gli errori avvengono. In ogni oggetto JSON prodotto il campo `reason` distingue le varie tipologie di messaggio, possiamo quindi filtrare in base a tale criterio i messaggi che ci interessano.

In particolare, il formato dei messaggi del compilatore, omettendo le informazioni non pertinenti l'analisi effettuata, è il seguente:

```
{
    /* The "reason" indicates the kind of message. */
    "reason": "compiler-message",
    /* The Package ID, a unique identifier for referring to the package. */
    "package_id": "my-package 0.1.0 (path+file:///path/to/my-package)",
    /* The Cargo target (lib, bin, example, etc.) that generated the message. */
    /* ... */
    /* The message emitted by the compiler.
    See https://doc.rust-lang.org/rustc/json.html for details.
    */
    "message": {
        {
            /* The primary message. */
```

```
14              "message": "unused variable:  x ",
15              /* The diagnostic code.
16                 Some messages may set this value to null.
17              */
18              "code": {
19                  /* A unique string identifying which …
                       diagnostic triggered. */
20                  "code": "unused_variables",
21                  /* An optional string explaining more detail …
                       about the diagnostic code. */
22                  "explanation": null
23              },
24              /* The severity of the diagnostic.
25                 Values may be:
26                 - "error": A fatal error that prevents …
                      compilation.
27                 - "warning": A possible error or concern.
28                 - "note": Additional information or context …
                      about the diagnostic.
29                 - "help": A suggestion on how to resolve the …
                      diagnostic.
30                 - "failure-note": A note attached to the …
                      message for further information.
31                 - "error: internal compiler error": Indicates a …
                      bug within the compiler.
32              */
33              "level": "warning",
34              /* ... */
35              "rendered": "warning: unused variable:  x \n --> …
                   lib.rs:2:9\n  |\n2 |      let x = 123;\n  | …
                            ^ help: if this is intentional, prefix …
                   it with an underscore:  _x \n  |\n  = note: …
                    #[warn(unused_variables)]  on by default\n\n"
36          }
37      }
38  }
```

I campi importanti ai fini della classificazione sono:

- `reason`: come già accennato, siamo interessati soltanto ai messaggi che hanno `compiler-message` come valore del campo, in quanto sono quelli che contengono i

messaggi del compilatore come warning ed errori.

- `package_id`: identifica il package a cui fa riferimento il messaggio in questione. Viene utilizzato per distinguere se l'errore avviene nella crate corrente o in una sua dipendenza. Nel caso l'errore sia estraneo alla crate corrente permette anche di distinguere in quale delle crate dipendenti esso si origini.

- `message/code`: contiene il codice univoco che identifica l'errore in questione.

- `message/level`: identifica il livello del messaggio, cioè se ad esempio è un warning, un errore, o un *internal compiler error*.

- `message/rendered`: questo è il messaggio che normalmente viene mostrato all'utente quando si sceglie il formato output (human) di default.

Per la classificazione richiesta bisognerà quindi salvare il contenuto del campo `code` soltanto se il livello è `error`. In aggiunta, bisogna tenere conto del fatto che una crate potrebbe presentare più di un errore.

Come illustrato precedentemente, `package_id` identifica la crate in cui si origina l'errore. Più nel dettaglio tale campo è composto nel seguente modo:

```
"package_id": "package_name package_version (crate_kind+url)"
```

La parte `crate_kind+url` dipende dal modo in cui sono state definite le dipendenze all'interno del file `Cargo.toml`. I possibili valori sono:

- `path+file:///path/to/package`: identifica le dipendenze locali. Queste dipendenze non sono supportate in Crater in quanto non sarebbe possibile reperire tali package. Per tale motivo nello step di preparazione della crate sono eliminate dal file. L'unica crate valida che presenta un path locale è la crate che viene testata, dove il path indica il percorso della cartella corrente.

- `git+http[s]://url/to/package#commitsha`: identifica le dipendenze specificate tramite repository git. In tale caso viene anche salvato lo sha dell'ultimo commit al momento del checkout.

- `registry+url/to/registry`: identifica le dipendenza da registry. Solitamente esse appartengono a *crates.io*, il registry pubblico ufficiale di Rust. In tal caso il valore completo sarà `registry+https://github.com/rust-lang/crates.io-index`, dove l'url contiene il *registry index* ospitato su GitHub.

Un problema che però si pone è il fatto che tale formato JSON è più difficilmente leggibile da un essere umano, risultando in un deciso peggioramento dell'usabilità dei log, che sono ottenuti catturando l'output dell'esecuzione dei test. Si provvederà quindi a sostituire al messaggio JSON il valore contenuto nel campo `message/rendered`, dopo aver estratto le informazioni utili. A tale scopo è necessario modificare Rustwide per permettere tale operazione.

3.1.1 Rustwide

Prima dell'intervento la funzione F che si poteva fornire al comando Rustwide era definita nel seguente modo:

```
1 &mut dyn FnMut(&str)
```

cioè una qualsiasi funzione che prende in input un riferimento immutabile alla riga di output in questione. `FnMut` identifica una funzione che (opzionalmente) prende riferimenti mutabili alle variabili che cattura.

Non è quindi consentito modificare la riga di output o influenzare in alcun modo il comportamento del comando.

Una modifica veloce potrebbe essere quella di cambiare la signature della funzione per ottenere

```
1 &mut dyn FnMut(&mut str)
```

in modo da permettere alla funzione di modificare la stringa corrispondente alla riga di output. Tale scelta però male si presta per espansioni future. Assumiamo infatti di voler aggiungere una qualche funzionalità che deve accedere al comportamento interno di Rust-

wide, come ad esempio terminare il comando, che potrebbe essere anche molto lungo, se si legge nell'output la stringa `invalid input sequence`. Come abbiamo visto precedentemente, Rustwide può eseguire i comandi all'interno di container Docker ma i dettagli di come questo avvenga, e dati come l'id del container utilizzato, vengono mantenuti nascosti dall'implementazione. Una funzione esterna a Rustwide non avrebbe accesso quindi a tali informazioni.

La soluzione che si è scelta è quella di creare una sorta di **proxy**, esposto da Rustwide come argomento della funzione F, che si occupa di gestire tutte le richieste che l'utilizzatore della funzione F vuole effettuare e che necessitano di un accesso alle risorse interne di Rustwide. Nel caso si voglia in futuro aggiungere una nuova funzionalità, sarà sufficiente aggiungere un metodo al proxy, mantenendo la retro-compatibilità con le versioni precedenti.

Si è di conseguenza modificata la signature della funzione F nel seguente modo:

```
1 &mut dyn FnMut(&str, &mut ProcessLinesActions)
```

Al momento di chiamare la funzione, Rustwide passa come argomento oltre alla riga di output anche la struct `ProcessLinesAction` che funge da proxy tra i due attori. Lo stato interno di `ProcessLinesAction` è privato e mantiene al momento le modifiche effettuate all'output. Essa espone metodi con due diverse classi di visibilità.

- `pub(super)`: i metodi con questa visibilità sono visibili soltanto dal modulo padre del modulo in cui essi sono dichiarati. In questo caso si permette al modulo che si occupa della gestione dei comandi Rustwide di avere accesso ai metodi di creazione della struct e di ottenere le modifiche desiderate dall'utente.

- `pub`: i metodi con questa visibilità sono visibili a tutti gli utilizzatori della funzione. In questo modo l'utente finale della libreria può indicare i cambiamenti che desidera effettuare.

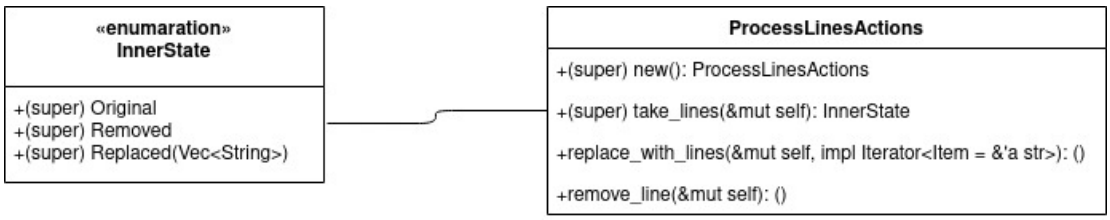

Figura 3.1: Schema UML della struct ProcessLinesActions

Analogamente, la visibilità dell'enum che rappresenta lo stato interno è limitata al modulo padre.

3.1.2 Procedimento completo

Vediamo ora come Crater analizza l'output del comando Rustwide invocato. Se la riga ricevuta non è JSON essa viene semplicemente ignorata: questo può avvenire in quanto Rustwide può essere configurato ad esempio per includere informazioni sul comando eseguito all'interno dei log. Se il messaggio ricevuto ha `reason: "compiler-message"` allora dopo aver estratto le informazioni necessarie viene chiamato il metodo `replace_with_lines` per sostuire l'oggetto JSON con il contenuto del campo `message/rendered`, per ottenere nei log una stringa leggibile. Nel caso invece il campo `reason` contenga un valore diverso si elimina semplicemente l'oggetto dall'output con `remove_line`.

Per determinare se l'errore si origina nella crate corrente o in una sua dipendenza bisogna analizzare il contenuto di `package_id`. Inizialmente viene invocato all'interno del contesto di build il comando `cargo metadata --no-deps --format-version=1` che produce informazioni in formato JSON sui metadati dei membri dello spazio di lavoro corrente (con `--no-deps` si omettono le informazioni riguardanti le dipendenze). Per questo comando viene disabilitato il logging da parte di Rustwide in quanto altrimenti andrebbe a riempire i log della crate con informazioni inutili. Il formato dell'oggetto JSON ottenuto è il seguente:

```
1 {
2     /* Array of all packages in the workspace.
3        It also includes all feature-enabled dependencies ...
           unless --no-deps is used.
4     */
5     "packages": [
6         {
```

```
7               /* The name of the package. */
8               "name": "my-package",
9               /* The version of the package. */
10              "version": "0.1.0",
11              /* The Package ID, a unique identifier for …
                   referring to the package. */
12              "id": "my-package 0.1.0 …
                   (path+file:///path/to/my-package)",
13              /* The license value from the manifest, or null. */
14              "license": "MIT/Apache-2.0",
15              /* The license-file value from the manifest, or …
                   null. */
16              "license_file": "LICENSE",
17              /* The description value from the manifest, or …
                   null. */
18              "description": "Package description.",
19              /*... */
20          }
21      ],
22      /* Array of members of the workspace.
23         Each entry is the Package ID for the package.
24      */
25      "workspace_members": [
26          "my-package 0.1.0 (path+file:///path/to/my-package)",
27      ],
28      /* The resolved dependency graph for the entire workspace. …
           The enabled
29         features are based on the enabled features for the …
              "current" package.
30         Inactivated optional dependencies are not listed.
31
32         This is null if --no-deps is specified.
33      */
34      "resolve": null,
35
36      /* ... */
37  }
```

Successivamente si raccolgono soltanto gli id dei package contenuti nella lista `packages` e li si salva in un set. Questo comando viene chiamato una sola volta per crate e toolchain

prima di eseguire i test e ne viene salvato l'output in modo da poterlo riutilizzare.

Per ogni nuovo messaggio del compilatore si controlla se il `package_id` del messaggio è contenuto nel set dei package locali, ed in tal caso si salva l'errore come riferito alla crate corrente.

In caso contrario bisogna determinare a quale crate esso fa riferimento. A tale scopo si è implementato `TryFrom<PackageId>` per `Crate`, l'enum che rappresenta le varie crate all'interno di Crater, dove `PackageId` è la struct ottenuta deserializzando il campo indicato. `TryFrom<U>` è un `trait` della libreria standard di Rust che indica una conversione che può non andare a buon fine. Nel caso la conversione avvenga con successo si segnala invece che la crate corrente ha delle dipendenze che ne impediscono la compilazione.

Il vantaggio della conversione *al volo*, seppure necessiti di aggiornamenti nel caso il formato interno utilizzato dal compilatore per la rappresentazione dei `package_id` cambi, è che permette di non usare memoria aggiuntiva per salvare le crate esistenti. Un altro approccio, scartato appunto per l'eccessivo utilizzo di memoria che potrebbe protrarsi anche per l'intero tempo di compilazione della crate, è quello di salvare l'intera lista delle dipendenze (ottenuta sempre con `cargo metadata` ma senza `--no-deps`) e poi riconoscere la crate dal campo `name` presente nella serializzazione JSON di ogni package. Quest'ultima alternativa aveva invece come vantaggio il fatto di utilizzare interfacce stabili con il compilatore.

3.2 Salvataggio delle informazioni ottenute

Il prossimo passo necessario è quello di salvare le informazioni ottenute per poi riutilizzarle al momento della creazione del report.

3.2.1 Gestione delle crate

Come si può immaginare, una componente fondamentale nel dominio sono le crate. La rappresentazione preesistente però non è sufficiente per definirne interamente le caratteristiche.

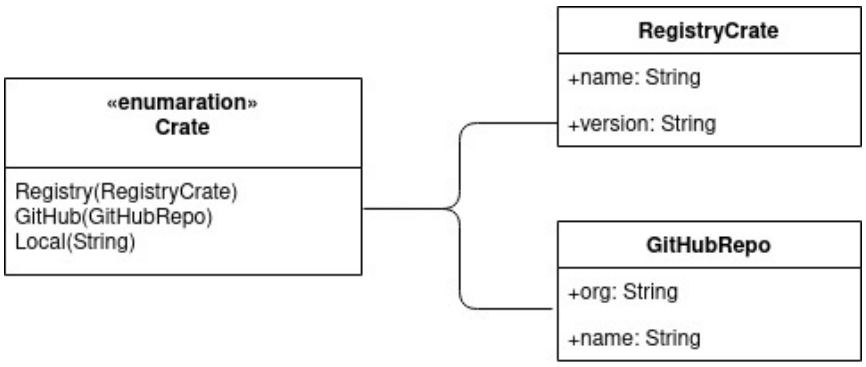

Figura 3.2: Schema della rappresentazione delle crate prima dell'intervento

Innanzitutto è necessario introdurre altre due varianti che rappresentino le crate con riferimento git (non GitHub) e con percorso locale. Anche se tali crate non saranno presenti nelle liste iniziali delle crate su cui eseguire l'esperimento, poiché come detto al momento esiste uno scraper soltanto per GitHub e per *crates.io*, potrebbe capitare che ad esempio una delle crate della lista contenga come dipendenza un repository git di qualche altro provider. Questa situazione è assolutamente valida e non c'è motivo di precludere la possibilità di analizzare tali crate. Può quindi capitare che una crate A non direttamente testata da Crater sia causa di regressioni tra le crate della lista. In tal caso si dovrà comunque mostrare correttamente le relazione di dipendenza, anche se non saranno disponibili i risultati in dettaglio della crate A.

In aggiunta, è fondamentale l'introduzione dello sha dell'ultimo commit al momento del checkout per le crate con riferimento ad un repository git. In sua assenza si potrebbero infatti confondere versioni differenti della stessa crate, falsando i risultati nel report. Questo può succedere perchè nel file `Cargo.toml` di configurazione delle dipendenze è possibile indicare sia un commit specifico sia un particolare branch.

È importante notare che al momento della creazione dell'esperimento non si specifica una versione particolare delle crate, in quanto si è interessati a quella più recente. In un secondo momento si analizza la lista delle crate (contenente fino a qual momento soltanto i nomi) e si testa la versione più recente disponibile. Questo avviene al momento della creazione dell'esperimento per le crate con riferimento a *crates.io*, e al momento dell'esecuzione dei test per la crate specifica nel caso essa provenga da GitHub. Per tale motivo si è scelto di rappresentare lo sha come `Option`, in quanto la sua presenza può essere determinata dallo stato in cui si trova. È inoltre opportuno notare che per talune crate, nello specifico

quelle il cui repository è diventato privato o è stato cancellato, non si otterrà mai uno sha, per cui anche al momento della creazione del report non è detto che tale campo sia occupato.

Avendo cambiato anche la rappresentazione delle crate nel database è stato necessario introdurre una migrazione che avesse cura di trasformare le informazioni esistenti nel nuovo formato, più facilmente leggibile.

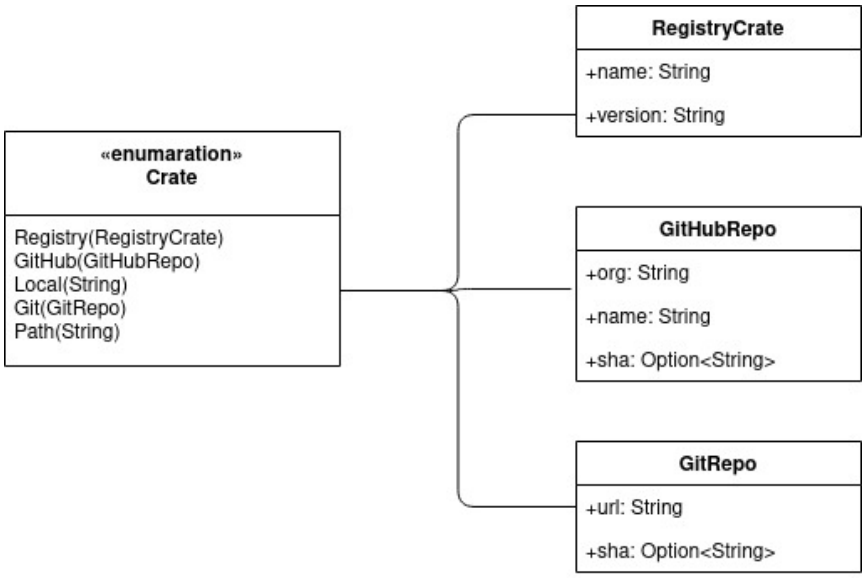

Figura 3.3: Schema della rappresentazione delle crate dopo la modifica

Quando un agent chiede al server di fornirgli un nuovo esperimento da eseguire esso gli spedisce una lista di crate in cui lo sha non è specificato, se la crate in questione proviene da GitHub. Nello step di preparazione della crate Rustwide effettua il checkout del repository, e lo sha dell'ultimo commit viene salvato per poter aggiornare la versione della crate utilizzata. Diventa a questo punto necessario comunicare al server, oltre ai risultati dell'esperimento, la versione specifica della crate utilizzata.

A tale scopo si è modificata la struttura dell'oggetto JSON utilizzato per scambiarsi dati nell'endpoint `record-progress` con cui l'agent comunica al server il risultato dell'esecuzione di ogni test. Si è scelto di includere anche un nuovo campo `version` contenente, al bisogno, la nuova versione della crate da salvare. Una soluzione più verbosa, con la specifica completa delle due crate, è stata preferita poichè permette di utilizzare lo stesso metodo con tutte le tipologie di crate, mentre una soluzione apposita per `Crate::GitHub(...)` non sarebbe potuta essere utilizzata per aggiornare la versione delle crate registry nel caso

ce ne fosse stato bisogno. Si prevede infatti di posticipare la scelta della versione delle crate registry anch'essa al momento dell'esecuzione della crate specifica, in modo da minimizzare la possibilità di testare versioni obsolete.

Le specifiche dell'endpoint realizzato sono le seguenti:

```
{
    "experiment-name": "pr-1",
    "result": [
        {
            "crate": {
                "GitHub": {
                    "org": "brson",
                    "repo": "hello-rs",
                    "sha" : null
                }
            },
            "toolchain": {
                "Dist": "stable"
            },
            "result": "TestPass",
            "log": "cGlhZGluYSByb21hZ25vbGE="
        }
    ],
    "version": [
        {
            "GitHub": {
                "org": "brson",
                "repo": "hello-rs",
                "sha" : null
            }
        },
        {
            "GitHub": {
                "org": "brson",
                "repo": "hello-rs",
                "sha" : "fa00fe"
            }
        }
    ]
}
```

Alla ricezione del messaggio il server, oltre a salvare il risultato ottenuto, aggiornerà la versione della crate utilizzata per l'esperimento.

3.2.2 Rappresentazione del risultato di test

Ogni risultato è descritto tramite una enum `TestResult`, in cui sono presenti i vari possibili casi. Particolare attenzione è data ai casi in cui i test falliscono ed è possibile specificare informazioni aggiuntive sulle cause del fallimento. Prima dell'intervento le cause disponibili erano `Unknown`, `Timeout` e `OOM`. Il metodo `is_spurious` indica se tale causa è da considerare spuria. Al momento attuale sono indicate come spurie le cause `OOM` e `Timeout`, in quanto il risultato potrebbe essere influenzato da fattori non sotto il controllo del compilatore.

Per individuare e classificare i risultati secondo i requisiti specificati è stato necessario introdurre tre nuove varianti:

- `CompilerError(...)`: con questa variante si indicano le crate che falliscono a causa di uno o più errori interni di cui si conosce il codice.
- `DependsOn(...)`: indica le crate che falliscono a causa di errori originati in una o più dipendenze.
- `ICE`: indica le crate che falliscono a causa di un bug del compilatore.

La classe di appartenenza del risultato di una crate, nel caso questa presenti errori, viene scelta nel seguente modo:

- Se il comando termina per aver raggiunto i limiti di memoria o tempo la causa sarà rispettivamente `FailureReason::OOM` o `FailureReason::Timeout`.
- Se durante l'esecuzione vengono rilevati messaggi di livello `ICE` la causa salvata sarà `FailureReason::ICE` anche se sono presenti altri errori in quanto è la più critica.
- Se non sono presenti messaggi di livello `ICE` e sono presenti errori in dipendenze della crate corrente significa che la build della crate corrente è fallita a causa delle sue dipendenze. La causa sarà quindi `FailureReason::DependsOn(...)`
- Se non sono presenti errori nelle dipendenze ma sono presenti errori nella crate corrente la causa sarà `FailureReason::CompilerError(...)`

- Se nessuna delle situazioni sopra si verifica si usa `FailureReason::Unknown`.

Il compilatore Rust non si ferma al primo errore incontrato per evitare all'utente di dover compilare il progetto tante volte quante sono gli errori e velocizzare invece il processo di correzione del codice. Per questo motivo è necessario predisporre gli oggetti del dominio per poter rappresentare questa situazione. In particolare potrebbe essere necessario rappresentare più di un errore di compilazione o di una dipendenza.

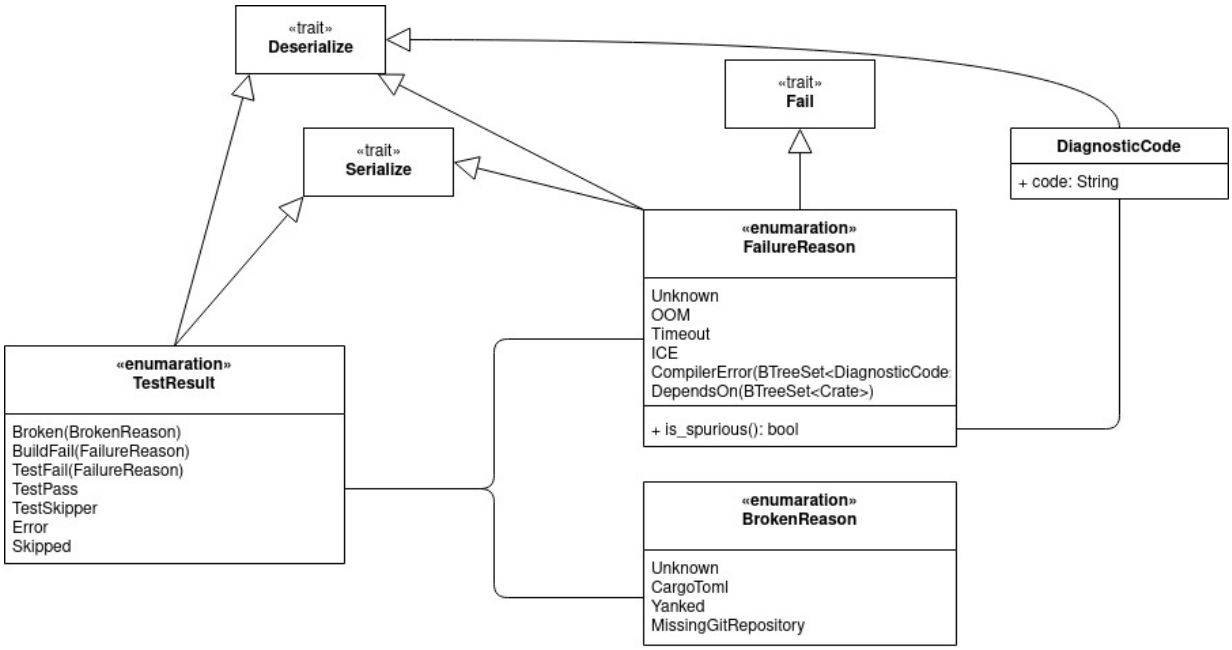

Figura 3.4: Schema della rappresentazione del risultato dell'esecuzione dei test per una crate e toolchain.

Per tale motivo si è scelto di utilizzare un `BTreeSet` come contenitore degli oggetti di `FailureReason::DependsOn` e `FailureReason::CompilerError`. Nel primo caso gli oggetti contenuti saranno enum `Crate`, e nel secondo struct `DiagnosticCode`, che contengono il codice emesso dal compilatore. È stato utilizzato un set in quanto non ci interessa né la molteplicità di un errore né quella di una dipendenza. Rust offre sia set basati su hashing, come `HashSet`, generalmente più veloci, sia set basati sull'ordinamento, come `BTreeSet`. Si è scelto di utilizzare un `BTreeSet` per facilitare le operazioni di serializzazione e deserializzazione. L'ordine con cui i valori di un `HashSet` sono letti non è infatti deterministico e questo potrebbe risultare in serializzazioni differenti dello stesso oggetto, cosa che si vuole

evitare. Allo stesso momento in questo modo ci si assicura anche che serializzazioni in cui l'unica cosa che cambia è l'ordine degli elementi portino allo stesso oggetto deserializzato.

3.3 Rappresentazione delle informazioni

Parte fondamentale del progetto è mostrare le informazioni ricavate agli utenti, in modo da velocizzare il processo di analisi delle problematiche individuate. A tale scopo si è modificato il report HTML che veniva già fornito e si è introdotto anche un report in formato Markdown, di struttura diversa, per poterlo integrare più facilmente su GitHub, dove avviene buona parte del workflow del team di sviluppo.

3.3.1 Riorganizzazione della struttura del codice

I requisiti specificati richiedono uno studio più minuzioso dei dati rispetto a quanto non fosse già presente. Per questo motivo si è deciso di estrarre la parte logica di analisi dei risultati *grezzi*, che prima si trovava nel modulo HTML, in un modulo a parte per riutilizzare lo stesso codice per tutti i formati e/o report differenti, come HTML e Markdown.

Nello specifico, si considerano *grezzi*, `RawTestResults`, i dati ottenuti direttamente al termine dell'esecuzione di un esperimento. Per ogni crate essi contengono i risultati per entrambe le toolchain e la categoria (`regressed`, `fixed`, ...) a cui essa appartiene. Come evidenziato nei requisiti questo però non è sufficiente per mostrare le relazioni tra gli errori a cui siamo interessati. Si è quindi creato un nuovo modulo, `analyzer`, incaricato di analizzare i dati grezzi secondo le specifiche indicate. Compito dei moduli specifici per ogni tipologia di report sarà quindi quello di renderizzare il determinato formato, seguendo i propri canoni, partendo dai dati già pronti prodotti da `analyzer`. Spesso infatti formati diversi portano con sè anche scenari d'uso differenti in cui sono richieste informazioni diverse. Si discuterà dello scenario d'uso di ogni formato nella sezione apposita.

Sempre per meglio riorganizzare la struttura del codice, passando da una situazione in cui era presente solo il report HTML ad una in cui potenzialmente si fa uso di molteplici formati e tipologie, sono state estratte dal modulo HTML anche le informazioni legate alla rappresentazione visiva di elementi del dominio. Più in dettaglio, nel modulo `display` sono contenuti i `trait` e le relative implementazioni che permetto di visualizzare gli elementi come `TestResult` in modo uniforme attraverso tutto il codice.

Il processo che porta alla creazione dei report è quindi il seguente:

- Il modulo principale (`report/mod`), si occupa di leggere dal database i risultati, uno per ogni toolchain e per ogni crate, e determina la categoria di appartenenza di ogni crate. Quello che si ottiene è `RawTestResults`.
- In seguito il modulo `analyzer` si occupa di analizzare `RawTestResults` evidenziando le relazioni tra i dati così come specificate nei requisiti. Produce in output `TestResults` [1].
- I vari moduli per i diversi report (HTML, Markdown, ...) producono uno o più file nel formato richiesto partendo da `TestResults`.

3.3.2 Analyzer

Compito di questo modulo è quello di produrre dati analizzando le relazioni che esistono tra essi così come specificato nei requisiti. In particolare, viene richiesto di elencare per ogni codice d'errore l'elenco della crate che presentano tale problema e di elencare per ogni crate la lista delle dipendenze inverse che falliscono la compilazione a causa sua. Tale analisi non è richiesta per ognuna delle categorie, in quanto, ad esempio, è poco importante per `SameTestPass`.

Per evitare di rendere il sistema rigido, effettuando l'analisi solo per le categorie che al momento si reputano necessarie, si è scelto di definire la tipologia di report a livello di `Comparison`, l'enum che si occupa di rappresentare la categoria di appartenenza dei risultati come `Regressed`, `Fixed`, ... In questo modo ogni variante, implementando il metodo `report_config`, descrive la tipologia di analisi del report con cui vuole essere rappresentata, con una sorta di pattern **strategy**. Al momento i valori possibili sono `Simple`, con cui si indica una semplice lista di crate, e `Complete(ToolchainSelect)`. Con `Complete(ToolchainSelect)` si indica che si desidera ottenere un report completo sia con la categorizzazione per errore sia con la visualizzazione delle dipendenze.

Questa variante contiene anche un campo in cui è possibile specificare in quale toolchain si vuole effettuare l'analisi dei risultati. Prendiamo ad esempio `Comparison::Regressed`: in questo caso siamo interessati a categorizzare le crate in base ai risultati ottenuti nella toolchain *end*, poichè è li che si verifica il problema. Al contrario, se stiamo lavorando con

[1] si raccomanda di non confondere `TestResults` con `TestResult`, che invece rappresenta il risultato di un singolo test.

`Comparison::Fixed`, ci interessa categorizzare le crate in base ai risultati della toolchain *start*. In questo modo è possibile utilizzare lo stesso codice per entrambi i casi, semplicemente specificando la toolchain di interesse, rispettivamente `ToolchainSelect::End` e `ToolchainSelect::Start`.

Al momento della scrittura della tesi si avvalgono di un'analisi approfondita con `ReportConfig::Complete( ... )` soltanto le categorie `Comparison::Regressed` e `Comparison::Fixed`.

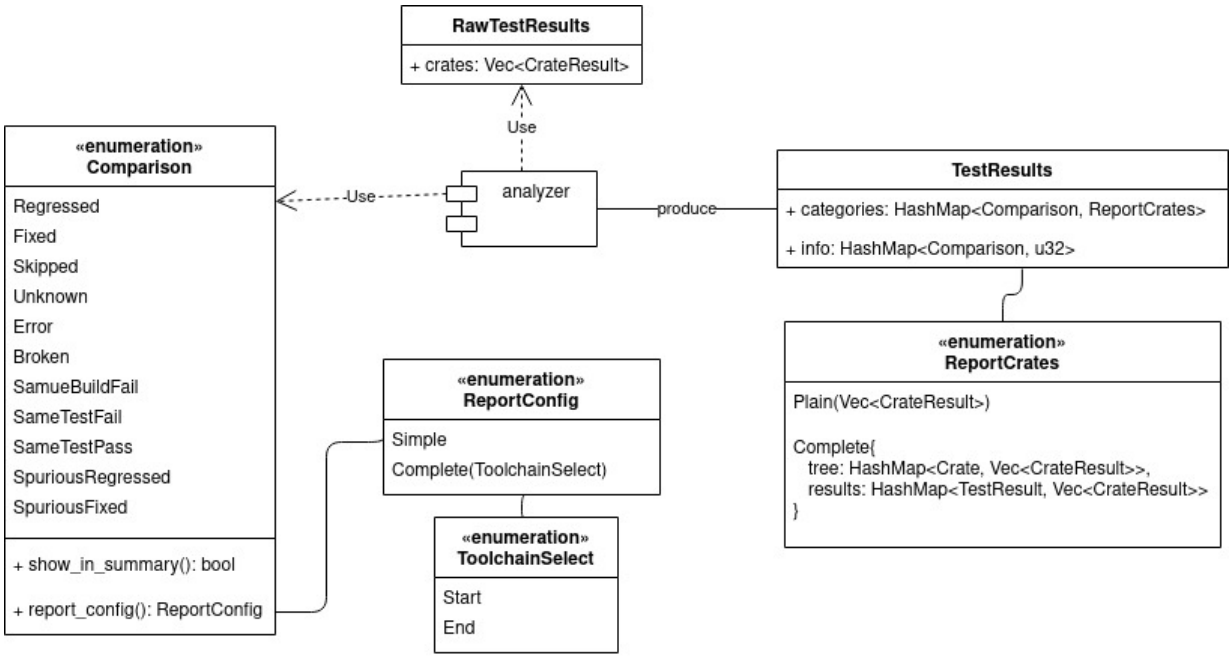

Figura 3.5: Schema degli elementi nell'analisi dei dati di test.

Analogamente, il formato dei data analizzati prodotti in output dipende dal tipo di approfondimento richiesto dalla categoria. L'enum `ReportCrates` si occupa quindi di descrivere la struttura dei dati dopo che questi sono stati analizzati. `ReportCrates::Plain` identifica una semplice lista di crate, `ReportCrates::Complete { ... }` suddivide invece il contenuto in codici d'errore e dipendenze.

Sia per quanto riguarda `ReportCrates` che `ReportConfig`, se in futuro si presenterà la necessità di un nuovo formato per il report, sarà sufficiente aggiungere una variante che rappresenta tale struttura.

3.3.3 HTML

Scenario d'uso

Il report in formato HTML è quello più completo e complesso. Composto da più pagine, è il punto principale d'accesso per reperire anche gli altri formati e file aggiuntivi. Viene utilizzato come strumento d'analisi principale dei risultati e utilizza espedienti visivi per semplificarne la navigazione.

Design

Compito del modulo HTML è quello di visualizzare le informazioni in modo intuitivo e comodo per gli utenti. Nel dettaglio si richiedere di mostrare una tab collassabile per ogni categoria. Per le categorie che presentano sia la visione per errori che quella per dipendenze si richiedere di mostrare una tab per ognuna di esse.

Per rendere più facile la parte di renderizzazione vera e propria è stato aggiunto un layer di adattamento che modifica i dati in input e produce output con la stessa struttura con cui poi sarà visualizzato. In questo modo si possono individuare due componenti differenti all'interno del modulo: uno che si occupa di strutturare i dati secondo il formato richiesto dalle specifiche, ed uno che si occupa della creazione delle pagine HTML. Per questo motivo si sono create, ove necessario, anche i corrispettivi delle struct utilizzate dal modulo principale per descrivere il report in un modo più astratto, per portare la rappresentazione ad un livello più vicino alla specifica implementazione. Ad esempio è stata creata la struct `ReportCratesHTML` che, a differenza di `ReportCrates`, differenzia in varianti diverse la categorizzazione per codice d'errore e per dipendenza, così come richiesto dalla visualizzazione del report HTML.

Le varianti al momento presenti sono proposte all'utente nel seguente modo:

- `ReportCratesHTML::Plain`: Viene semplicemente visualizzata la lista di crate che ad essa appartengono. Questo è il caso ad esempio di `Comparison::SameTestPass`

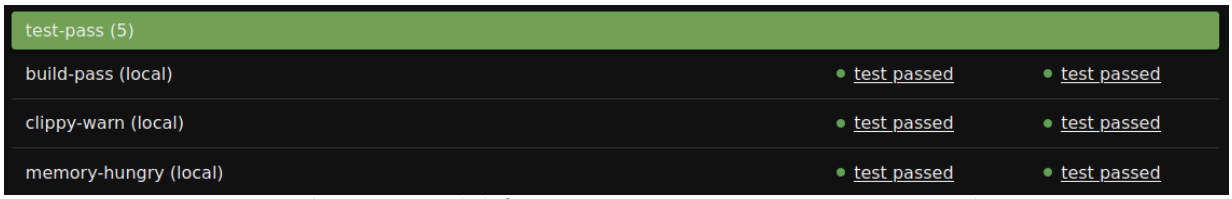

Figura 3.6: Visualizzazione del formato `ReportCrates::Plain` nel report HTML.

- `ReportCratesHTML::Tree`: È presente una tab in cui vengono mostrate le relazioni di dipendenza trovate. Per ogni crate viene mostrata la lista di crate che da essa dipendono e che per tale causa falliscono la compilazione. È possibile sia che una crate sia presente più di una volta con versioni o sha differenti sia che in questa sezione vengano mostrate crate non direttamente testate da Crater. Questo succede se le dipendenze delle crate testate presentano tale problema.

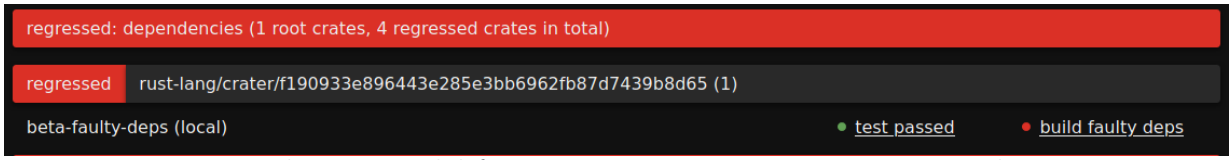

Figura 3.7: Visualizzazione del formato `ReportCratesHTML::Tree` nel report HTML.

- `ReportCratesHTML::RootResults`: È presente una tab in cui vengono mostrate per ogni tipologia d'errore le crate affette. Nel caso particolare di compilazione fallita a causa di codice di errore, si è scelto di dividere ogni errore in una sotto-categoria indipendente. In caso una crate presenti più di un codice di errore, essa viene replicata nelle rispettive liste. In questa tab non sono visualizzate le crate che falliscono a causa di errori nella dipendenze.

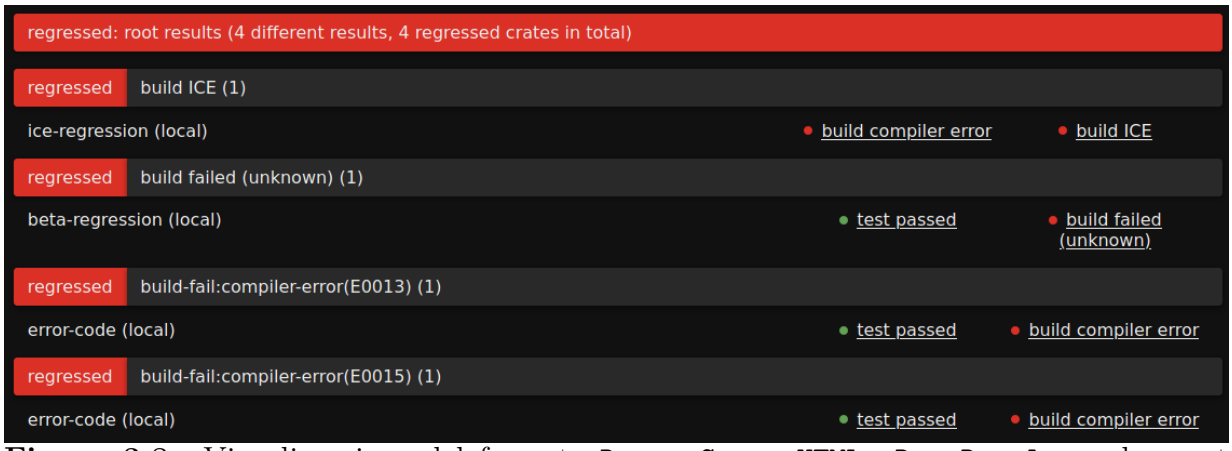

Figura 3.8: Visualizzazione del formato `ReportCratesHTML::RootResults` nel report HTML.

3.3.4 Markdown

Scenario d'uso

Il report in Markdown è una nuova aggiunta mirata a semplificare l'interazione con GitHub e Zulip. Molto della comunicazione avviene infatti su questi canali che trattano molto più facilmente file in formato Markdown, che può essere integrato direttamente nei messaggi, che HTML.

In aggiunta, è comune la creazione di un file condiviso, sempre in formato Markdown, in cui si segna lo stato dei lavori per ognuna delle crate regredite e la sua creazione manuale può essere molto tediosa se il numero delle regressioni è considerevole. Per tali motivi, il report Markdown contiene soltanto i risultati relativi alle categorie `Comparison::Regressed` e `Comparison::Fixed`.

Design

Differentemente dal report HTML, ed in accordo con lo scenario d'uso, in questo caso si preferisce non duplicare le crate per tracciarne più facilmente lo stato. Per ogni crate si indica quindi l'errore completo rilevato ed eventuali crate che dipendono da essa e falliscono la compilazione per tale causa. In caso una crate non compili a causa di più dipendenze sarà necessario duplicarla in tutte le liste corrispondenti. Questo non è un problema in quanto per tali crate non è solitamente necessario un intervento ma è invece importante conoscere per ogni **root regression** quante crate falliscono a causa sua.

Anche in questo caso si possono individuare due componenti distinti all'interno del modulo: uno che elabora i dati ricevuti da `analyzer` ed uno che effettivamente crea il file desiderato. In questo modo la funzione che si occupa di generare il report riceve in input dati con formato corrispondente alla rappresentazione richiesta e si deve occupare soltanto di tradurre tipi di dato Rust in codice Markdown.

Figura 3.9: Un esempio di report Markdown.

3.4 Problematiche nelle query API

Per risolvere alcuni requisiti, come l'individuazione di una nuova versione, è necessario interrogare i rispettivi fornitori delle crate, cioè GitHub o *crates.io*.

GitHub prevede una limitazione delle richieste API che si possono fare. Crater fa già un utilizzo intenso delle API GitHub per cercare tutte i progetti Rust nel portale, restando a fatica nei limiti imposti. In questo caso un ulteriore utilizzo delle risorse limitate sarebbe controproducente, sopratutto considerando il fatto che si dovrebbe necessariamente rallentare in un qualche modo la funzione di ricerca di nuove crate, reputata più importante della feature in questione. Per questo motivo si è scelto di non fornire questo servizio per le crate ospitate su GitHub.

Anche *crates.io* espone delle API pubbliche. Ancora una volta, l'accesso a queste risorse è controllato e scoraggiato per quanto possibile. Stando a quanto dichiarato nelle policy per i *crawler* non è infatti assicurato che le nostre richieste vengano soddisfatte [4]. In questo caso viene in nostro aiuto il mirror [2] di alcune informazioni del registry ospitato su GitHub. Esso permette, una volta scaricato, di effettuare molte delle query sui metadati di una crate localmente, con il doppio beneficio di non appesantire il servizio pubblico e di

[2] crates.io-index

velocizzare le risposte.

Al momento di generare il report, se la crate proviene da *crates.io*, si interroga l'indice locale del registry per ottenere l'ultima versione disponibile. Se essa non coincide con quella testata viene mostrato un avvertimento nel report.

Capitolo 4

Realizzazione

Sommario

Essenziale per produrre un qualsiasi sistema, la parte di implementazione porta con sé una serie di aspetti che sono analizzati in questo capitolo. In particolare, si discute delle scelte implementative e delle tecnologie e metodologie utilizzate nello sviluppo del software.

4.1 Linguaggi e Tecnologie di sviluppo

Rust

Dovendo integrare il sistema all'interno di un applicativo già esistente è stato necessario utilizzare il linguaggio di programmazione Rust. La scelta sarebbe ricaduta su questo linguaggio anche senza i limiti imposti poichè in questo modo, dovendo gestire codice Rust come dominio applicativo, si riusano le conoscenze acquisite e l'integrazione è sicuramente più facile. Ad esempio, alcune delle librerie utilizzate, come `crates_index`, esistono al momento solo per Rust, e ciò avrebbe reso l'implementazione con un altro linguaggio sicuramente più difficoltosa.

HTML, CSS, Markdown

La realizzazione dei report ha richiesto l'utilizzo sia di HTML e CSS, per il report principale, sia di Markdown. Questi formati sono stati scelti poichè bene si adattano alla visualizzazione in browser per cui tale applicativo è pensato.

Per Markdown in particolare si è usato lo standard CommonMark, largamente supportato, in modo da ottenere risultati riproducibili in vari ambienti, sia su GitHub che su programmi esterni. Markdown è infatti solitamente convertito in HTML prima di essere visualizzato.

Cargo

Cargo è il package manager di Rust. Tra le sue funzioni gestisce le dipendenze tramite il file di configurazione `Cargo.toml`. In aggiunta, grazie al file `Cargo.lock` assicura di poter ottenere risultati ripetibili salvando i dettagli specifici dell'esecuzione della build. Un'altra funzione messa a disposizione è quella di gestione ed esecuzione di unit ed integration test. Cargo è stato utilizzando intensamente sia per sviluppare il sistema di categorizzazione dei risultati sia come strumento interno a Crater per eseguire i test sulle crate. Abbiamo visto infatti che mette a disposizione molti strumenti per facilitarne l'interoperabilità con altri tool [6], [7].

Altre funzionalità di cargo, non utilizzate all'interno di questo progetto, includono la possibilità di pubblicare direttamente la propria crate nel registry pubblico ufficiale *crates.io*.

Git e GitHub

Git è un *Distributed Version-Control System* (DVCS) che permette di descrivere le modifiche al codice facilitando la collaborazione tra sviluppatori. Ciò è stato particolarmente importante per molteplici motivi, tra i quali:

- Nel repository pubblico molteplici persone hanno accesso al codice e possono apportare modifiche previa autorizzazione. Durante lo sviluppo del sistema per la tesi è capitato di dover risolvere conflitti con modifiche non esistenti al momento della progettazione e Git ha sicuramente permesso di fare ciò molto più agilmente.
- Lo sviluppo stesso delle varie feature del sistema è avvenuto parallelamente. Capitava infatti che alcune modifiche dipendessero da fattori esterni (come il rilascio di una nuova versione di Rustwide) e fossero quindi momentaneamente bloccate. In tal caso si procedeva lavorando ad una funzionalità differente. Git è stato usato sia per tenere traccia di queste modifiche parallele, tramite diversi branch, sia per integrarle una volta completate.

L'intero progetto è ospitato su GitHub. Il processo di modifica del codice risente di questa situazione in quanto fa uso di *pull request* dove sono presenti anche integrazioni specifiche per GitHub per gestire la *continuous integration*.

4.2 Librerie utilizzate

Cargo metadata

`cargo_metadata` permette una accesso strutturato ai messaggi JSON emessi da cargo tramite struct Rust. È stato utilizzato per effettuare il parsing dei messaggi JSON in modo automatico all'interno dell'analisi dei log delle crate. In particolare la struct `CompilerMessage` rappresenta un messaggio del compilatore mentre la struct `Metadata` rappresenta l'output derivato dall'invocazione di `cargo metadata --format-version=1`.

Crates index

`crates_index` semplifica l'interazione con il mirror del registry ufficiale ospitato su GitHub. In tal modo si alleggerisce il carico a *crates.io* per quelle richieste che possono essere soddisfatte anche dal mirror come l'ultima versione disponibile di una determinata

crate. La libreria può essere configurata per scaricare l'indice delle crate in una directory indicata dall'utente o direttamente utilizzando la copia di `cargo`.

Tera

Tera è un *template-engine* che facilita la generazione di file HTML [8]. Inizialmente si scrive un file di template con estensione `.html` in cui si possono specificare, tramite sintassi HTML, tutti gli elementi statici come il layout della pagina. In tale file è possibile utilizzare anche sintassi propria di Tera per includere elementi dinamici come controlli di flusso e cicli.

In aggiunta, tramite un oggetto `Context`, è possibile passare una serie di variabili Rust che potranno essere utilizzate nel template per popolare il report con dati dinamici. Tale libreria è molto comoda per separare la parte visiva, interamente gestita da HTML e fogli di stile CSS, dai dati su cui essa si basa, permettendo anche di usare direttamente tipi di dato Rust, a patto che essi implementino il trait `Serialize`.

A titolo esemplificativo il codice seguente, una volta effettuato il preprocessing necessario, produrrà un paragrafo per ogni numero contenuto nel vettore Rust `sample` indicando se esso è pari o meno.

```
1 {% for num in sample %}
2     {% if num is odd %}
3         <p> Odd: {{ num }} </p>
4     {% else %}
5         <p> Even: {{ num }} </p>
6     {% endif %}
7 {% endfor %}
```

Serde

`serde` è una libreria che facilita la serializzazione e deserializzazione di tipi di dato Rust. Un utilizzo basilare si avvale dell'uso di *macro* per derivare i trait `Serialize` e `Deserialize` per gli oggetti desiderati.

```rust
#[derive(Serialize)]
struct ResultsContext< a> {
    ex: & a Experiment,
    categories: Vec<(Comparison, ReportCratesMD)>,
    info: IndexMap<Comparison, u32>,
    full: bool,
    crates_count: usize,
}
```

Se necessario, è possibile configurare in modo dettagliato il comportamento della libreria anche a livello di singoli campi della struttura.

```rust
#[derive(Serialize)]
enum ReportCratesMD {
    Plain(Vec<CrateResult>),
    Complete {
        // la funzione indicata si occupa di serializzare
        // il campo nel modo voluto per cambiare il
        // comportamento rispetto a quello di default

        // only string keys are allowed in JSON maps
        #[serde(serialize_with = "to_vec")]
        res: IndexMap<CrateResult, Vec<CrateResult>>,
        #[serde(serialize_with = "to_vec")]
        orphans: IndexMap<Crate, Vec<CrateResult>>,
    },
}
```

Una volta implementati i trait `Serialize` e `Deserialize`, è possibile serializzare/deserializzare gli oggetti in tutti i formati messi a disposizione dalla libreria, come JSON,

Toml e Yaml [9].

Entrambi gli esempi riportati fanno riferimento al file `src/report/markdown.rs`. In particolare è stato necessario modificare il comportamento di default di `serde` per poter serializzare correttamente i campi con `IndexMap`, in quanto le mappe JSON possono contenere soltanto stringhe come chiavi, mentre in questo caso si fa uso di tipi di dato differenti come `CrateResult`. Si è quindi convertita la mappa in un vettore di tuple prima della serializzazione.

Reqwest, warp

`reqwest` è un client HTTP di facile utilizzo. All'interno del progetto viene utilizzato dagli agent per effettuare le varie richieste al server.

`warp` invece è il server HTTP impiegato dal server Crater. Esso viene utilizzato per interagire con:

- GitHub per quanto riguarda la ricezione dei webhook utilizzati per schedulare nuovi esperimenti.
- Gli agent per esporre gli endpoint necessari alla comunicazione ed al coordinamento.
- Gli utenti sotto forma di server web per monitorare lo stato dell'infrastruttura e l'avanzamento dei lavori.

`warp` permette di utilizzare funzioni Rust per descrivere il comportamento di un endpoint, integrandosi quindi alla perfezione e minimizzando il bisogno di scrivere codice che non appartiene al dominio applicativo.

4.3 Testing

Il testing ha costituito una parte considerevole del lavoro affrontato in questo progetto. È infatti abbastanza critico che tale strumento non dia risultati sbagliati, in quanto non solo ne renderebbe inutile l'utilizzo, ma potrebbe causare anche ritardi significati in momenti in cui non si ha a disposizione molto tempo. Si pensi alla situazione in cui viene effettuato il test della nuova versione del compilatore da rilasciare. Se per una qualche modifica effettuata Crater produce ora risultati sbagliati, e lo si realizza solo al termine

dell'esecuzione dell'esperimento, si rischia di perdere una settimana di tempo in una situazione in cui invece è importante procedere con sveltezza. Per questo motivo sono stati utilizzati sia *unit test*, più veloci, sia *integration test*, più completi e affidabili.

4.3.1 Unit Test

Gli unit test sono stati utilizzati in modo estensivo per poter verificare il funzionamento dei moduli realizzati. In questo modo, se si sono predisposti test sufficientemente rappresentativi dell'utilizzo reale, si può velocizzare la scrittura del codice. Invece di eseguire l'intero programma per verificare la correttezza dei cambiamenti effettuati si eseguono solamente gli unit test fino a che essi non sono stati soddisfatti.

In alcuni casi, come nella generazione del report, permettono inoltre, definendo *artificialmente* una situazione di partenza, di poter testare il codice su casi più completi senza il bisogno di effettuare veramente la compilazione di un numero elevato di crate.

4.3.2 Integration Test

In un software discretamente complesso come Crater testare indipendentemente i vari componenti non è sufficiente a raggiungere una confidenza accettabile di assenza di bug. L'interazione tra i vari moduli può infatti introdurre *side-effect* indesiderati ed un piccolo cambiamento si può ripercuotere sull'intero sistema.

minicrater è l'infrastruttura che permette di effettuare test sull'intero processo di lavoro. Essa permette di definire nuovi esperimenti di test, ognuno con la propria configurazione. Internamente esegue tutti gli step necessari per completare gli esperimenti definiti per poi confrontare il report ottenuto con quello atteso. In questo modo si può riprodurre in piccolo l'intero comportamento del sistema.

Per avere pieno controllo sulle crate testate durante questi esperimenti si introducono le cosiddette **local-crates**. Esse sono semplici crate che presentano particolari caratteristiche ritenute interessanti nei test e sono distribuite nel repository insieme al resto del software. In questo modo si cerca di ricreare un insieme di crate di test il più possibile rappresentativo delle possibili varianti che si possono trovare realmente, limitando allo stesso tempo le risorse necessarie alla sua esecuzione.

In particolare, per assicurarsi del funzionamento delle nuove funzionalità introdotte si sono aggiunte le seguenti crate:

- `error-code`: fallisce la compilazione, solo nella toolchain *end*, a causa di errori interni noti. Viene utilizzata per testare l'individuazione di molteplici codici di errore e per la loro analisi nel report. È stato utilizzato un *build script* per assicurare la presenza di errori solo nella toolchain desiderata.

- `faulty-deps`: fallisce la compilazione a causa di errori nelle dipendenze. Viene utilizzata per testare l'individuazione di problemi in crate diverse da quella corrente e per la loro analisi nel report. Come dipendenze sono state utilizzate `error_code`, sotto il nostro controllo, ed una vecchia versione di `lazy_static`, una crate popolare mantenuta direttamente dall'organizzazione Rust nel registry ufficiale. Su *crates.io* non è permessa l'eliminazione di vecchie versioni, proprio per evitare incidenti come quello accaduto su npm [1], e ciò rende sufficientemente stabile la scelta della crate.

- `beta-faulty-deps`: analoga a `faulty-deps` ma fallisce la compilazione soltanto nella toolchain *end*. In questo modo viene testata l'individuazione delle dipendenze anche in un contesto di regressione e la loro visualizzazione nel report. Si è preferito realizzare due crate separate in quanto non è possibili indicare dipendenze da compilare opzionalmente in base alla toolchain scelta. Come conseguenza in `beta_faulty_deps` l'unica dipendenza utilizzata è `error_code`, che già presenta errori soltanto nella toolchain *end*, e non è invece stato possibile includere anche `lazy_static`. Per testare anche l'individuazione di crate con diverse provenienze si è quindi scelto di separare i due scopi.

Richiedendo un tempo non trascurabile per l'esecuzione, minicrater è di default disabilitato quando si esegue `cargo test`. Passando il flag `--ignored` è possibile forzare l'esecuzione, come comunque succede prima di effettuare il merge di PR sul repository.

Testing dei report

Si è inoltre aggiunta la possibilità di confrontare anche i risultati dei report HTML e Markdown, in modo da controllare come le informazioni vengono presentate agli utenti. Più che il report vero e proprio vengono serializzati in JSON i contenuti dei vari oggetti `Context` utilizzati per popolare i template con i dati reali. La parte di renderizzazione

[1] npm Blog post

produrrà infatti gli stessi risultati a partire dagli stessi dati in input ma il formato potrebbe peggiorare la leggibilità di eventuali differenze. Questa funzionalità non è ovviamente utilizzata in produzione per non appesantire ulteriormente il report. Tale modalità di testing è simile a quanto chiamato *Snapshot testing* dal framework di test JavaScript Jest [19].

Nella sotto-cartella `tests/minicrater` è presenta una directory per ogni esperimento da eseguire. Ognuna di queste directory contiene i file di configurazione ed i risultati attesi.

Quando si eseguono gli *integration test*, l'output ottenuto dall'esecuzione del programma viene confrontato con i valori attesi per ognuno degli esperimenti di test. In caso di differenze il test viene considerato fallito.

Per astrarre il concetto di comparazione è stata creato un trait apposito `Compare`. Definendo nome e formattazione dei file esso metterà a disposizione il metodo `compare` che si occupa di confrontare il risultato ottenuto con quello atteso. Le varianti dell'enum `Reports`, che rappresentano i vari report, implementano quindi ciascuna i metodi richiesti e possono poi essere gestite allo stesso modo.

La possibilità di specificare un metodo per la formattazione del file prima del confronto viene utilizzata, oltre che per aggiungere le indentazioni necessarie a rendere il testo più leggibile, anche per eliminare eventuali porzioni indesiderate. In `html::ResultContext`, ad esempio, sono presenti informazioni relative all'esperimento corrente, a cui durante i test è assegnato un nome casuale, e la cui data di inizio e fine dipendono dal momento in cui esso viene eseguito. Tali informazioni sono quindi eliminate prima di effettuare il confronto.

Un altro problema che si è dovuto affrontare è che oggetti come le `HashMap` non hanno una serializzazione deterministica, in quanto l'ordine delle chiavi non è definito. A questo scopo è stata quindi utilizzata `IndexMap`, una hash map con interfaccia compatibile ma il cui ordine degli elementi è consistente e definito dall'ordine di inserimento.

4.4 Dettagli implementativi

Si illustrano ora alcune porzioni di codice per aiutare meglio nella comprensione dell'intervento effettuato. Per comprendere appieno come questi interagiscono con il resto del programma si consiglia fare riferimento al repository ufficiale, anche perchè potrebbero

subire modifiche nel corso del tempo.

4.4.1 Esecuzione dei test

La seguente funzione si occupa di eseguire effettivamente i test per ogni crate. Essa predispone inizialmente l'ambiente Rustwide di esecuzione, specificando crate e toolchain e limitando le risorse a cui ha accesso come la memoria.

Successivamente, viene eseguita, all'interno del contesto di build appena creato, la funzione `test_fn` specificata tramite le diverse modalità dell'esperimento. Si può notare che l'elenco dei package locali viene ottenuto prima dell'esecuzione di `test_fn`, per permettere il suo riutilizzo se tale funzione contiene più comandi al suo interno.

```
1  info!(
2      "{} {} against {} for {}",
3      action,
4      ctx.krate,
5      ctx.toolchain.to_string(),
6      ctx.experiment.name
7  );
8  let sandbox = SandboxBuilder::new()
9      .memory_limit(Some(ctx.config.sandbox.memory_limit.to_bytes()))
10     .enable_networking(false);
11
12 let krate = &ctx.krate.to_rustwide();
13 let mut build_dir = ctx.build_dir.lock().unwrap();
14 let mut build = build_dir.build(&ctx.toolchain, krate, sandbox);
15
16 for patch in ctx.toolchain.patches.iter() {
17     build = build.patch_with_git(&patch.name, &patch.repo, ...
           &patch.branch);
18 }
19
20 detect_broken(build.run(|build| {
21     let local_packages_id = get_local_packages(build)?;
22     test_fn(ctx, build, &local_packages_id)
23 }))
```

Infine, con `detect_broken`, si individuano eventuali errori della classe `Broken`, come `Broken::MissingGitRepository`, traducendo errori di Rustwide in costrutti compatibili con Crater.

4.4.2 Invio dei risultati

Dopo aver eseguito i test, compito dei vari agent è quello di comunicare i risultati ottenuti al server, che li utilizzerà per creare il report d'analisi. Come accennato in precedenza, l'agent deve anche comunicare al server eventuali cambiamenti nella versione delle crate testate. Il codice a righe 4-18 si occupa di gestire le versioni effettive salvate durante l'esecuzione dei test.

```
1  let mut updated = None;
2  let mut new_version = None;
3  // This is done to avoid locking versions if record_progress retries
4  // in loop
5  {
6      let mut versions = self.versions.lock().unwrap();
7      if let Occupied(mut entry) = versions.entry(krate.clone()) {
8          let value = entry.get_mut();
9          if value.1 {
10             // delete entry if we already processed both toolchains
11             updated = Some(entry.remove().0);
12         } else {
13             updated = Some(value.0.clone());
14             new_version = updated.as_ref();
15             // mark we already sent the updated version to the server
16             value.1 = true;
17         }
18     };
19 }
20
21 info!("sending results to the crater server...");
22 self.api.record_progress(
23     ex,
24     updated.as_ref().unwrap_or(krate),
25     toolchain,
26     output.as_bytes(),
27     &result,
28     new_version.map(|new| (krate, new)),
29 )?;
```

La chiamata a funzione a riga 21 si occupa poi di mandare il messaggio con il formato richiesto esplicitato in sezione 3.2.1.

Tale funzione, per portare a termine il proprio compito anche in presenza di disturbi nella connettività di rete, ritenta la spedizione del messaggio se questa non è andata a buon

fine. Per evitare di mantenere il lock sulla variabile condivisa in ambiente multithreaded a riga 5, ed evitare quindi di bloccare gli altri thread nel caso quello corrente presenti problemi, si è introdotto un ulteriore scope a righe 4-18. In questo modo la variabile `versions`, uscendo dallo scope a riga 18, viene rilasciata prima di effettuare la richiesta via rete, e con lei viene resa nuovamente disponibile anche la variabile condivisa.

4.4.3 Generazione del report e analisi dei risultati

Completata l'esecuzione di tutti i test specificati nell'esperimento, è compito del server quello di generare il report. Inizialmente vengono reperiti dal database tutti i risultati ricevuti, ricavando per ogni crate la categoria di appartenenza così come specificato nella sezione 1.2.1.

```
1  pub fn gen<DB: ReadResults, W: ReportWriter + Display>(
2      db: &DB,
3      ex: &Experiment,
4      crates: &[Crate],
5      dest: &W,
6      config: &Config,
7      output_templates: bool,
8  ) -> Fallible<TestResults> {
9      let raw = generate_report(db, config, ex, crates)?;
10
11     info!("writing results to {}", dest);
12     info!("writing metadata");
13     dest.write_string(
14         "results.json",
15         serde_json::to_string(&raw)?.into(),
16         &mime::APPLICATION_JSON,
17     )?;
18     /* ... */
19     let res = analyze_report(raw);
20     info!("writing html files");
21     html::write_html_report(
22         ex,
23         crates.len(),
24         &res,
25         available_archives,
26         dest,
27         output_templates,
28     )?;
29     info!("writing markdown files");
```

```
30      markdown::write_markdown_report(ex, crates.len(), &res, dest, ...
            output_templates)?;
31      info!("writing logs");
32      write_logs(db, ex, crates, dest, config)?;
33
34      Ok(res)
35  }
```

Tali risultati sono poi serializzati in JSON per poterli recuperare in caso di bisogno. Il file `results.json` è utilizzato anche durante gli integration test per verificare la correttezza dei risultati ottenuti.

Essenziale all'interno di questo progetto è l'analisi dei dati prodotti dai test. La funzione `analyze_report` all'interno del modulo `src/report/analyzer.rs` si occupa proprio di questo aspetto, trasformando i dati grezzi secondo le specifiche di progetto.

```
1  pub fn analyze_report(test: RawTestResults) -> TestResults {
2      let mut comparison = IndexMap::new();
3      for krate in test.crates {
4          comparison
5              .entry(krate.res)
6              .or_insert_with(Vec::new)
7              .push(krate);
8      }
9      let info = comparison
10         .iter()
11         .map(|(&key, vec)| (key, vec.len() as u32))
12         .collect::<IndexMap<_, _>>();
13     let mut categories = IndexMap::new();
14     for (cat, crates) in comparison {
15         if let ReportConfig::Complete(toolchain) = cat.report_config() {
16             // variants in an enum are numbered following an
17             // increasing sequence starting from 0
18             categories.insert(cat, analyze_detailed(toolchain as usize, ...
                   crates));
19         } else {
20             categories.insert(cat, ReportCrates::Plain(crates));
21         }
22     }
23 }
```

La tipologia di analisi effettuata non è inserita staticamente nel codice ma ogni variante dell'enum `Comparison`, chiamata anche `cat` all'interno di questo spezzone di codice, può

indicare autonomamente la sua preferenza tramite il metodo `report_config`. Nel caso sia necessaria un'analisi più approfondita la funzione `analyze_detailed` si occuperà di evidenziare le relazioni tra gli errori e tra le dipendenze.

Successivamente, i moduli `html` e `markdown` si occupano di renderizzare i risultati nei corrispettivi formati.

Il flag `output_templates` controlla la serializzazione aggiuntiva del contesto in cui operano i template Markdown e HTML. Disabilitato in produzione, viene usato per debug e soprattutto negli integration test.

4.5 Metodologia di lavoro

In accordo con le prassi più comuni in ambito open source, il lavoro è stato svolto sul *fork* personale del repository pubblico. In questo modo si ha pieno controllo durante la fase implementativa.

Come già accennato, lo sviluppo non è stato sempre lineare per sfruttare al meglio il tempo disponibile, anche se sicuramente alcune dipendenze hanno impedito di procedere in modo puramente parallelo. Si è scelto inoltre di effettuare una *pull request* al repository ufficiale ogni qual volta si ottiene una funzionalità indipendente. In questo modo si è potuto osservare il comportamento di ognuno dei componenti prima di avere completato l'intero processo, anticipando la scoperta di eventuali problemi e semplificando anche l'analisi e la *review* del codice.

Lavorando su un repository ufficiale dell'organizzazione Rust, le modifiche non possono essere integrate autonomamente. Per ogni pull request è infatti necessaria l'approvazione da parte del maintainer. Solitamente, ed è il caso per il repository in questione, questa approvazione è preceduta da una review del codice, sempre da parte di una persone fidata interna all'organizzazione, in cui sono discussi aspetti implementativi o di design che necessitano di una correzione o che non sono sufficientemente chiari.

In aggiunta, prima di effettuare il merge vengono anche effettuati i test presenti all'interno della crate attraverso strumenti di continuous integration come GitHub Actions o Homu[2].

[2] Homu Bot

4.5.1 Gestione del repository

Nella gestione del repository è stato seguita la seguente organizzazione:

- Il branch master viene mantenuto sincronizzato con il branch master del repository ufficiale per avere sempre una copia di riferimento.
- Ogni feature indipendente viene sviluppata in un branch a parte. Se la feature è particolarmente impegnativa possono essere utilizzati sotto-branch per obbiettivi intermedi.

Capitolo 5

Conclusioni

Sommario

In questa tesi si è estesa la capacità di Crater di individuare le cause di fallimento dei test che tale programma esegue, applicandola anche ai codici d'errore emessi dal compilatore. In seguito, questa funzionalità introdotta è stata usata per individuare relazioni tra gli errori trovati, come eventuali dipendenze, e per presentare i risultati nel modo più consono.

Il sistema realizzato, seppur mancante di alcuni requisiti opzionali, ha permesso di estrarre dai dati ottenuti le informazioni più importanti, semplificando e velocizzando il lavoro del team di release.

La raccolta dei requisiti iniziale e la successiva analisi hanno permesso di aver chiaro fin da subito un programma di massima delle funzionalità da implementare. Molto utile è stato anche il confronto continuo con gli sviluppatori del linguaggio nel caso di dubbi emersi durante la progettazione o realizzazione del sistema.

La scomposizione del sistema in parti indipendenti è stata di grande importanza per la realizzazione del progetto. In questo modo, con consegne anticipate e testando il codice su casi reali, si sono potute isolare le problematiche che si sono presentate, rendendo più facile il processo di correzione. Allo stesso momento, questo approccio consente di rendere più facile l'analisi dei cambiamenti effettuati, semplificando sia l'esame delle integrazioni

con le parti esistenti, sia la review del codice.

Inoltre, l'adozione di un processo di sviluppo *test-driven* per il sistema realizzato, assieme all'introduzione di numerosi unit test e soprattutto integration test, ha permesso di velocizzarne l'implementazione. Dovendo analizzare una mole così imponente di codice Rust, si parla di circa 100.000 crate totali testate in un esperimento eseguito su tutte le crate disponibili, Crater stesso viene messo alla prova su situazioni e casistiche molto variabili. I test sono particolarmente importanti in questo caso in quanto è praticamente impensabile verificare la correttezza di Crater riproducendo il suo comportamento in produzione con tempi e risorse ragionevoli.

5.1 Lavori futuri

Tra i requisiti opzionali elencati inizialmente alcuni rimangono non realizzati, come il raggruppamento delle crate in categorie arbitrarie definite dall'utente, e sicuramente questa è una direzione di possibile ampliamento futuro del software.

In aggiunta, può essere interessante indagare possibili sviluppi nelle seguenti aree:

- Miglioramento della user experience soprattutto per quanto riguarda il report HTML. Modifiche future potrebbero facilitare ad esempio la lettura dei log, presentando il file nella pagina corrente ed evitando di dover spostare continuamente il focus tra diverse pagine del browser.

- Semplificazione delle cause d'errore rilevate attraverso strumenti come rust-reduce. In questo caso va valutato con attenzione anche l'impatto in termini di risorse che tale cambiamento potrebbe richiedere.

- Analisi e categorizzazione dei risultati anche per quanto riguarda l'output di `cargo test` e per errori di compilazione sprovvisti di codice.

Bibliografia

[1] Pietro Albini,
Shipping a compiler version every six weeks 4

[2] Rustwide repository,
https://github.com/rust-lang/rustwide

[3] Crater documentation,
https://github.com/rust-lang/crater

[4] `crates.io`,
https://crates.io 45

[5] Rust Compiler Error Index,
https://doc.rust-lang.org/error-index.html

[6] Cargo documentation,
https://doc.rust-lang.org/cargo/index.html 48

[7] Rust compiler documentation,
https://doc.rust-lang.org/rustc/index.html 48

[8] Tera documentation,
https://tera.netlify.app/ 50

[9] Serde documentation,
https://serde.rs 52

[10] Registry index format specification,
https://rust-lang.github.io/rfcs/2141-alternative-registries.htmlregistry-index-format-specification

[11] Microsoft Security Response Center,
Why Rust for safe systems programming 3

[12] Nicholas D. Matsakis and Felix S. Klock. 2014. The rust language. Ada Lett. 34, 3 (December 2014), 103104.
https://doi.org/10.1145/2692956.2663188

[13] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foun-dations of the Rust Programming Language. *Proc. ACM Program. Lang.2*, POPL, Article 66 (January 2018), 34 pages.
https://doi.org/10.1145/3158154 2

[14] GCC Testing Efforts,
https://gcc.gnu.org/testing 13

[15] GCC Development Plan,
https://gcc.gnu.org/develop.html 12

[16] PEP 602 – Annual Release Cycle for Python,
https://www.python.org/dev/peps/pep-0602 13

[17] Changes to How We Manage DefinitelyTyped,
https://devblogs.microsoft.com/typescript/changes-to-how-we-manage-definitelytyped 14

[18] TypeScript 3.6 Iteration Plan,
https://github.com/microsoft/TypeScript/issues/31639 14

[19] Snapshot testing,
https://jestjs.io/docs/en/snapshot-testing 55

[20] JEP 3: JDK Release Process,
https://openjdk.java.net/jeps/3 14

[21] Michlmayr, M. (2007). Quality improvement in volunteer free and open source software projects exploring the impact of release management. 15

[22] Analisi delle regressioni riportate dagli utenti dalla versione 1.35 alla versione 1.38 del compilatore Rust.
https://gist.github.com/pietroalbini/b02cadb117cfe49ad17e0168ce543e2d

Per tutti i link elencati è stato verificato il corretto funzionamento in data 1/07/2020.