Mini-Rust: i lifetime di Rust implementati con SML

Simone Meddi, Eleonora Rocchi, Davide Sforza, Giovanni Varricchione

1 Introduzione

Per il progetto abbiamo deciso di implementare i *lifetimes* ed il sistema che usa Rust per validare le referenze; in questo modo Rust può controllare che i programmi non abbiano puntatori a zone di memoria che, in quel momento, non sono più occupate da dati.

Rust implementa questo concetto dei lifetime sia per le **variabili**, assicurandosi che una variabile può contenere una referenza solo ad un'altra il cui lifetime è **più grande** del suo (in pratica solo alle variabili che sono state dichiarate prima), sia per le **funzioni**, implementando i lifetime come dei *generici* all'interno di queste.

Nelle funzioni, infatti, se l'intestazione contiene una o più referenze come input per la funzione stessa, allora è possibile dichiarare dei **lifetime generici** ed assegnarli a questi parametri: questo è molto utile se l'oggetto che ritorna la funzione è a sua volta una referenza, infatti è possibile assegnare anche a ciò che si ritorna un lifetime generico. Il puntatore che verrà tornato avrà quindi il lifetime più piccolo fra quelli che sono stati passati alla funzione e che, nell'intestazione, hanno lo stesso lifetime generico del ritorno.

Oltre ad aver implementato queste caratteristiche di Rust, abbiamo utilizzato anche **ML-Lex** ed **ML-Yacc**, il primo per poter creare un *lexer* che tokenizzasse testi in modo da poterlo poi passare al *parser* generato dal secondo; di conseguenza abbiamo dovuto creare anche una grammatica ed una lista di token per il nostro linguaggio.

2 Grammatica

Di seguito è descritta la grammatica usata, con i terminali (dati dalle loro espressioni regolari), le variabili e le regole di derivazione.

2.1 Terminali

SEMI → ;

LBRA → {

RBRA → }

CONST → {digit}+

PLUS → +

LPAR → (

RPAR →)

COMMA → ,

LET → let

ASS → =

FUN → fn

LCHE → <

RCHE → >

LTIME → '{alpha}({alpha}|{digit})*

COLON → :

AMP → &

INT → i32

ARROW → ->

PRINT → println!({ws}|{eol})*(({ws}|{eol})*"{}"({ws}|{eol})*,

ID → {alpha}({alpha}|{digit})*

IDRET → {alpha}({alpha}|{digit})*({ws}|{eol})*}

MAIN → fn({ws}|{eol})+main

Dove abbiamo che:

- **alpha** = [A-Za-z]
- **digit** = [0-9]
- **ws** = [\ \t]
- **eol** = (\r\n|\n|\r)

2.2 Variabili e regole di derivazione

main →*MAIN LPAR RPAR rust RBRA*

rust →*SEMI rust*
| *block rust*
| *exp SEMI rust*
| *llet*
| *ass SEMI rust*
| *ffun*
| *print SEMI rust*
| ε

block →*LBRA rust RBRA*

exp →*expPlus* | *call* | *LPAR call RPAR*

expPlus →*CONST*
| *varId*
| *AMP varId*
| *expPlus PLUS expPlus*
| *LPAR expPlus RPAR*

call →*funId LPAR aarg RPAR*

aarg →*arg* | ε

arg →*AMP varId* | *AMP varId COMMA arg*

llet →*LET varId SEMI rust*
| *LET varId ASS SEMI rust*

ass →*varId ASS exp*

ffun →*FUN funId lltime LPAR ppar RPAR ret blockfun rust*
| *FUN funId lltime LPAR ppar RPAR block rust*

blockfun →*LBRA rust IDRET*

lltime →*LCHE ltime RCHE* | ε

ltime →*LTIME* | *LTIME COMMA ltime*

ppar →*par* | ε

par →*varId COLON AMP LTIME INT*
| *varId COLON AMP INT*
| *varId COLON AMP LTIME INT COMMA par*
| *varId COLON AMP INT COMMA par*

ret →*ARROW AMP LTIME INT* | *ARROW AMP INT*

print →*PRINT exp RPAR*

varId →*ID*

funId →*ID*

3 Sintassi Astratta

Per la sintassi del progetto ci siamo basati sulla sintassi di ***Fun*** mostrata a lezione, che includeva funzioni e chiamate a funzioni, oltre a variabili ed i *let*.

Tutte le espressioni che si trovano **all'interno di parentesi quadre** [ ] indicano la possibilità che quell'oggetto sia presente anche un numero di volte diverso da 1 (si noti che, nel caso del ritorno in *fn*, questo numero può **essere solo 0 o 1**).

```
x, y = string

lt = ε | 'string

args = ε | lt x args

M, N = undef | int | x | &x | M + N | x ([args])

r, s =  ε | ; | r;s | M; | {r} | let x = M; | x = M;
      | {r; x} | println!("{}", M);
      | fn x <[lt]> ([args]) [-> lt] {r} s
```

Le ***metavariabili*** rappresentano:

- **x, y**: i nomi delle **variabili** e delle **funzioni**;
- **lt**: un **singolo lifetime**;
- **args**: una **lista di argomenti**;
- **M, N**: un'**espressione**;
- **r, s**: un **programma rust**.

4 Struttura del Progetto

Il progetto contiene una serie di file, di seguito la loro descrizione:

- **./src/datatypes.sml**: contiene tutti i datatypes definiti, dandone i costruttori e le relative segnature;
- **./src/rust.grm**: contiene la grammatica (definita in precedenza) da passare ad **ML-Yacc**;
- **./src/rust.lex**: contiene le definizioni dei token per poter creare il lexer, passandolo in input ad **ML-Lex**;
- **./src/glue.sml**: contiene codice sml usato da ML-Yacc e ML-Lex per utilizzare i datatype definiti e quindi generare correttamente il lexer ed il parser;
- **./src/main.sml**: contiene le due funzioni principali del progetto:
  1. **Rust.compile**: riceve in input la stringa di una percorso ad un programma e lo compila, dando in output un oggetto che si può passare a Rust.run;
  2. **Rust.run**: riceve in input un programma compilato e lo esegue;
- **./examples**: cartella che contiene dei programmi testabili (quelli che hanno la dicitura "*Wrong*" non compilano);
- **mini-rust.cm**: file che genera il lexer ed il parser chiamando ML-Lex e ML-Yacc;
- **test.sml**: piccolo programma sml che serve da *demo* del progetto, compilando ed eseguendo i programmi nella cartella ./examples.

4.1 Come eseguire test

Prima di tutto bisogna scrivere un programma che può essere riconosciuto dal parser (per i costrutti riferirsi alla **Sintassi Astratta** oppure agli esempi presenti nella relativa cartella).

Si noti che i programmi vanno definiti con un *main* che ne contiene tutto il corpo, incluse le funzioni dichiarate.
A questo punto ci posizioniamo nella cartella e, dopo aver lanciato sml, basta eseguire i seguenti comandi:

```
- CM.make "mini-rust.cm";
- Rust.run(Rust.compile(percorsoFile.rs));
```

Il primo comando serve per poter istanziare le strutture necessarie, mentre il secondo (che è una combinazione di due) prima esegue la compilazione del programma *percorsoFile.rs* e poi lo esegue. In alternativa, ma fuori da sml, è possibile lanciare il seguente comando per dare una demo del progetto:

```
$ sml test.sml
```

Il programma test.sml infatti esegue prima di tutto il CM.make, e poi chiama Rust.run(Rust.compile()) su dei programmi già scritti.

4.2 Programmi di esempio

4.2.1 lifetime.rs

Il programma **lifetime.rs** dà una piccola dimostrazione di un utilizzo corretto dei lifetime, sia con le referenze sia con l'output di una funzione. La variabile *k*, che è una referenza a *s*, ha un lifetime più piccolo della variabile che punta e, alla fine del main, poichè viene rilasciata prima il puntatore e poi la variabile (Rust infatti **rilascia le variabili nell'ordine inverso di dichiarazione**, allora non si creano puntatori a zone di memoria liberate. Allo stesso modo la variabile *res* non dà problemi perchè il puntatore con il lifetime generico uguale a quello di output e più piccolo (in realtà l'unico) è *w*, che ha un lifetime maggiore di res.

4.2.2 lifetimeWrong.rs

L'esempio **lifetimeWrong.rs** mostra il caso in cui, in una funzione, il puntatore che viene **ritornato** ha, **nell'intestazione**, un **lifetime generico diverso** da quello dichiarato per l'output. Nel momento in cui il compilatore si accorge di questa incongruenza (che nel programma avviene nella funzione *dummy*, in cui la *y* ha lifetime *'b* mentre l'output *'a*) stampa a schermo il seguente errore:

```
error: lifetime mismatch
```

4.2.3 doubleFuncWrong.rs

L'unico problema che abbiamo in **doubleFuncWrong.rs** è che nella funzione *dummy1*, la variabile *k* ha un lifetime **minore** della variabile *z* e, poichè z stessa diventa una referenza a k, la prima viene rilasciata prima del suo puntatore, creando quindi un cosiddetto *dangling pointer*, che punta ad una zona di memoria rilasciata.

Mini-Rust stampa di conseguenza il seguente errore:

```
error:  `k` does not live long enough
```

4.2.4 implicitLifetime.rs

Il programma **implicitLifetime.rs** (come da nome) mostra l'applicazione delle **prime due regole** definite nel caso della **lifetime elision** nelle funzioni, ossia quando l'utente dichiara nell'intestazione di una funzione dei parametri che sono puntatori, **omettendone però i lifetime**; il compilatore può considerare **corretta** una funzione solo nel caso in cui vi sia un **unico puntatore in input**, perchè così sa che quello in output avrà sicuramente lo stesso dell'input, e questo è proprio ciò che accade nella funzione *dummy* nel programma.

4.2.5 implicitLifetimeWrong.rs

A differenza del precendete, **implicitLifetimeWrong.rs** mostra proprio il caso in cui **non è possibile applicare correttamente le regole d'elisione**, in quanto la funzione riceve in input due puntatori **senza lifetime**: poichè il compilatore non sa definire il lifetime dell'output stampa il seguente errore:

```
error:  missing lifetime specifier
```