Languages, compilers and interpreters
Relazione progetto

Lorenzo Gazzella - 546890
Lorenzo Carfagna - 546942

2019/2020

1 Estensioni del linguaggio

Il linguaggio di programmazione visto a lezione è stato esteso in modo da supportare due nuovi tipi di dato con le relative operazioni elencate di seguito:

- Stringhe
  - Operatore di concatenazione (+)
  - Operatore di confronto (=, NE)
  - Lunghezza (#)
  - print_string(str)
- Coppie
  - Operatore di proiezione (.)

Come comunemente accade, le stringhe sono racchiuse tra virgolette, al cui interno è possibile utilizzare i più classici caratteri di escaping:

- \t tabulazione
- \n new line
- \\ \
- \" "
- \' '

E le coppie fra parentesi: (elemento1, elemento2) in cui i due elementi possono essere di qualsiasi tipo, comprese stringhe e coppie, ma non modificabili. Ovvero non sono accettati codici del tipo:

```
var x = ("ciao", 1) in x.1 := "a"
```

2 Estensione scanner

Per supportare l'input di stringhe dall'utente è stato esteso lo scanner aggiungendo il supporto per il token STRING_KW, come in figura sottostante, dove:

- strcat_ è una funzione che concatena la seconda stringa in input alla prima, non ponendo vincoli alla dimensione dei parametri.
- escaped è una funzione che, invocata nella start condition $< escape >$, ossia dopo aver incontrato un backslash, restituisce il carattere di escape corrispondente a quello in lettura.

```
{DQUOTE}              {yylval.string = strdup(""); BEGIN(quote);}
<quote>[^\\\"]*       {yylval.string = strcat_(yylval.string, yytext);}
<quote>{BACKSLASH}    {BEGIN(escape);}
<quote>{DQUOTE}       {BEGIN(INITIAL); return STRING_KW;}
<quote><<EOF>>        return ERROR_KW;
<escape>{ESCAPE}      {int len = strlen(yylval.string);
                       yylval.string = realloc(yylval.string, len + 2);
                       yylval.string[len] = escaped(yytext[0]);
                       BEGIN(quote);}
<escape>.             return ERROR_KW;
```

3 Estensione parser

E' stata estesa la produzione expr con i seguenti casi corrispondenti a creazione di coppia, proiezione, creazione di stringhe e lunghezza di una stringa.

```
| '(' expr ',' expr ')' {$$ = make_pair($2, $4);}
| expr '.' VAL {$$ = make_projection($1, $3);}

| STRING_KW {$$ = make_string($1);}
| '#' expr {$$ = make_un_op('#', $2);}
```

Le funzioni make_* popolano opportunamente la struct relativa alle espressioni con tutti i parametri di interesse alla compilazione.

Le operazioni di concatenazione (+), confronto (=, NE) non hanno richiesto un'estensione dell'espressione regolare, quindi vedono la loro precedenza e associatività invariata. La lunghezza di una stringa (#), essendo un operatore unario, ha precedenza maggiore di tutti quelli binari. L'operatore di proiezione (.) è stato definito associativo a sinistra con precedenza massima in modo da valutare correttamente un programma del tipo #("str", 1).1

4 Implementazione backend

4.1 Stringhe

Le stringhe sono state definite in LLVM come puntatori a un intero di 8 bit (i8 *) e non come array di i8 siccome questa seconda rappresentazione avrebbe creato problemi in programmi in cui la lunghezza delle stringhe non fosse nota a tempo di compilazione. Qui sotto riportiamo un codice di esempio:

```
var n = read_i32(0) in
var s = "" in
while n > 0 do
  let _ = n := n - 1 in
  s := s + "str"
```

Nel codice il valore di n è un input dell'utente e quindi non può essere dedotto staticamente, così come la lunghezza di s. Questo esclude l'utilizzo di array LLVM, la cui dimensione deve essere fissata prima dell'esecuzione del programma. Le stringhe, come in C, sono dotate di carattere terminatore \0, questo comporta che l'implementazione dell'operatore # debba iterare su di esse per determinare la loro lunghezza. A questo scopo è stata utilizzata la funzione di libreria `strlen`. Una rappresentazione alternativa, che abbatterebbe il costo computazionale di #, è quella di memorizzare le stringhe come strutture LLVM del tipo {stringa: i8*, len: i32}. La procedura di creazione di una stringa sfrutta la funzione C `strdup` che alloca la sequenza di caratteri nello heap. E' quindi necessario prevedere un meccanismo di deallocazione della memoria utilizzata. Sono stati individuati due metodi per la gestione della memoria:

- Counting reference
- Deallocazione all'uscita del blocco

Counting reference Per implementare counting reference sarebbe stato necessario associare ad ogni i8* un contatore del numero di riferimenti attivi e seguire la seguente procedura:

- Alla creazione di una nuova stringa, impostare il contatore ad 1
- In caso di assegnamento x := y
  - Incrementare il contatore della stringa y
  - Decrementare il contatore della stringa x
    - In caso di contatore nullo, deallocare la stringa puntata da x
  - Assegnare y ad x

In questo modo l'assegnamento fra stringhe x := y viene implementato come assegnamento di puntatori. Tuttavia implementare questo metodo avrebbe comportato l'utilizzo di una struttura dati a runtime con i conseguenti costi computazionali per la ricerca dei riferimenti.

Deallocazione all'uscita del blocco Un approccio alternativo è quello di deallocare una stringa all'uscita del blocco in cui è stata dichiarata. Questo tuttavia ha dei problemi con il codice sottostante

```
var x = "a" in let _ = var y = "b" in x:=y in print_string(x)
```

dove alla print_string(x) del blocco più esterno, si proverebbe ad accedere ad un riferimento deallocato alla chiusura del blocco più interno (var y = ...). Per risolvere questo problema è stata prevista una copy out del valore da assegnare in caso di assegnamento tra stringhe. Questo risolve i problemi di dangling reference, ma non quelli relativi ai memory leak, siccome un codice come quello sottostante non libererebbe il literal "c".

```
var x = "a" in let y = x + "c" in x:=y
```

Per risolvere quest'ultimo problema è stato utilizzato un paradigma in cui ogni operatore riceve input che lui stesso dovrà deallocare e restituisce in output un elemento che il chiamante dovrà liberare. Questo risolve i memory leak per i literal ma ripropone i dangling reference per gli identificatori: nell'esempio la stringa x viene deallocata due volte, una prima volta nell'operazione x + "c" e una seconda alla chiusura del blocco più esterno. Il problema è stato risolto inserendo una copy out alla valutazione di ogni identificatore. La semantica dell'assegnamento è stata modificata in modo che x := y venga compilata nel seguente modo:

- y sarà una copia, perchè valutazione di un identificatore, di un operatore o di un literal
- viene liberata la stringa riferita (unicamente) da x e creato il nuovo binding

Va notato che la stringa originale y verrà deallocata alla chiusura del blocco var y = ... , mentre la copia di y (assegnata ad x) verrà deallocata alla chiusura del blocco var x = ...

4.2 Coppie

Inizialmente si è pensato di definire il tipo coppia come una struttura infinitamente ricorsiva del seguente tipo:

```
%type = { i1 | i32 | i8* | %type, i1 | i32 | i8* | %type}
```

dove con i1 | i32 |i8* |%type si intende il tipo unione dei tipi separati da |. Tuttavia il tipo unione non è nativo LLVM ed i tipi ricorsivamente infiniti non sono accettati[1]. Per questo le coppie sono state implementate come strutture aventi due campi, il cui tipo è definito a tempo di compilazione secondo la seguente procedura ricorsiva, dove i casi base sono i tipi nativi di LLVM .

[1] http://lists.llvm.org/pipermail/llvm-dev/2011-October/043764.html

Assumendo ricorsivamente che first e second abbiano i tipi corretti determinati a tempo di compilazione, il nuovo tipo viene definito come una struttura con tipo: {`type(first), type(second)`}

```
LLVMValueRef first = codegen_expr(e->pair.first, env, module,
    builder);
LLVMValueRef second = codegen_expr(e->pair.second, env, module,
    builder);

LLVMTypeRef types[] = {LLVMTypeOf(first),LLVMTypeOf(second) };
LLVMTypeRef type = LLVMStructType(types, 2, 0);

LLVMValueRef struct_tmp = LLVMBuildInsertValue(builder,
    LLVMGetUndef(type), first, 0, "");
return LLVMBuildInsertValue(builder, struct_tmp, second, 1, "");
```

Ad esempio la coppia (”a”, (1, ”b”)) avrà tipo { i8*, { i8*, i32 } } dove il tipo del secondo elemento è creato dal passo ricorsivo codegen_expr(second).
Con la procedura delineata le coppie possono interagire con qualsiasi altro tipo, comprese le stringhe. Di conseguenza valgono gli analoghi discorsi riguardo copy out e deallocazione di memoria per l’intera coppia, altrimenti il codice successivo sarebbe problematico.

```
var x = ("ciao", 1) in
let _ = var y = ("b", 1) in x := y
   in print_string(x.1)
```