

Descrizione del linguaggio Scala40

sssbbf

1 Soluzioni non standard

1.1 Gestione del log tramite la monade **Writer**

La gestione del log merita un discorso dedicato, per comprendere appieno cosa è stato svolto bisogna partire da una analisi del modulo **Errors.hs**.

Gli elementi di log, che possono essere **Warning** oppure **Errori**, contengono la posizione in cui si è verificata l'eccezione e la rispettiva eccezione. Le eccezioni sono di tipo **TException** e ogni eccezione ha un messaggio di notifica dedicato.

Il modulo **Environment.hs** che fornisce l'interfaccia al modulo di analisi statica per la gestione dell'environment del programma, utilizza una versione modificata della monade di default per la gestione degli errori utilizzata da BNFC. Nella versione da noi utilizzata in caso di errore viene ritornato un oggetto di tipo **TException** invece che un oggetto **String**.

All'interno del modulo **StaticAnalysis.hs** è invece la monade **Writer** ad occuparsi di immagazzinare gli elementi di log creati durante l'analisi statica del programma. La monade **Writer** si occupa di immagazzinare gli elementi di log all'interno di una lista di **LogElement**. Un nuovo elemento di log viene aggiunto alla lista tramite la funzione **tell**.

Giusto per chiarezza sintattica, nel nostro codice è stata aggiunta una funzione **saveLog** che è poco più che un alias della funzione **tell**, ed è stato definito l'alias **Logger a** per **Writer [LogElement] a**. Ogni funzione del modulo di analisi statica che può scovare un'eccezione durante l'analisi dell'albero di sintassi astratta, ritorna dunque un elemento **Logger a**.

1.2 Gestione delle istruzioni TAC tramite la monade **State**

Per la gestione della creazione e dell'ordine di esecuzione delle istruzioni TAC durante la creazione di quest'ultimo, è stata utilizzata la monade **State a**. Questa permette di trattare l'inserimento di nuove istruzioni, la creazione di nuove etichette e di nuovi temporanei, come fossero un cambio di stato in una computazione. La monade **State a** utilizza due funzioni per operare sullo stato corrente: **get**, per ottenerlo, e **put** per aggiornarlo.

La monade **State a** utilizzata durante la creazione del Three Address Code opera su stati formati da tuple di quattro elementi:

- un intero che indica l'indice dell'ultimo temporaneo creato;

- un intero che indica l'ultima etichetta creata;
- la lista di istruzioni TAC che compongono il TAC in output;
- una lista di liste di istruzioni TAC. Ogni lista contiene un insieme di istruzioni di una singola funzione.

Lo stato viene modificato nel momento in cui si crea una nuova etichetta e/o un nuovo temporaneo (per evitare duplicazioni tra le istruzioni) (funzioni `newLabel` e `newTemp`, rispettivamente), quando una nuova istruzione TAC di una funzione viene inserita nella lista delle istruzioni di quella funzione (tramite `out`) e, nel momento in cui tutti gli statetement di una funzione sono stati tradotti in istruzione TAC, quando l'insieme delle istruzioni di quelle funzioni vengono aggiunte in testa all'insieme delle istruzioni TAC principale (quelle che compongono il TAC in output), tramite la funzione `pushCurrentStream`.

Nel file `ThreeAddressCode.hs` è stato utilizzato l'alias `TacState` a per `State (Int, Int, [TAC], [[TAC]])` a, dove `TAC` è un'istruzione TAC.

2 Descrizione soluzione

2.1 Il modulo di analisi statica

In questa sezione viene fornita una breve descrizione della logica retrostante il funzionamento del modulo di analisi statica `StaticAnalysis.hs`.

Il modulo di analisi statica fornisce un'interfaccia alle restanti componenti del compilatore, che permette di ottenere un albero di sintassi astratta annotato ed eventualmente una lista di log a partire da un programma `Scala40` rappresentato in sintassi astratta.

Per la gestione dei log si rimanda alla sezione riguardante le tecniche non standard utilizzate.

Il modulo è stato progettato seguendo un approccio *top-down*, andando di volta in volta a risolvere i sotto-problemi che ci apparivano dinanzi.

Il modulo di analisi statica fa largo uso del modulo `Environment.hs` che gli offre un'interfaccia per la gestione dell'environment del programma sotto analisi.

L'environment di un programma è una pila di *scope*, dove ogni scope è formato da: una *tabella di lookup* che mette in corrispondenza gli identificatori definiti all'interno dello scope con le informazioni a riguardo, un *tipo* che eredita dalla funzione in cui lo scope è stato creato ed infine un *valore booleano* che indica se nello scope è presente o meno un `return` con tipo compatibile con quello dello scope.

Il modulo `Environment.hs` fornisce tutte le funzioni necessarie per la gestione dell'environment di un programma. La funzione `lookup` permette di ottenere le informazioni a riguardo di un identificatore dichiarato in precedenza nello scope corrente, o in un *super-scope* nel quale lo scope corrente è contenuto; nel caso in cui l'identificatore cercato non sia trovato un'eccezione viene ritornata. La funzione `update` permette di inserire una nuova corrispondenza *identificatore-info* nello scope corrente, nel caso in cui l'identificatore che si sta cercando di

inserire fosse già stato dichiarato in precedenza all'interno dello scope corrente una eccezione viene ritornata. Per ottenere il tipo dello scope corrente è presente la funzione *getScopeType*, la funzione *hasReturn* ritorna invece il valore booleano che indica se nello scope corrente è presente o meno un **return** di tipo coerente con quello dello scope; quando un tale **return** viene trovato la funzione **setReturnFound** permette di impostare tale valore booleano a **True**. La funzione **addScope** permette di aggiungere un nuovo scope all'environment, la funzione prende come argomento il tipo dello scope che si vuole aggiungere. Qui è importante porsi una domanda:

Quale tipo viene passato ad **addScope**?

La risposta è: *dipende*. Se lo scope è lo scope globale del programma per una nostra convenzione esso ha tipo **TSimple_TypeVoid**; se lo scope viene creato da una dichiarazione di funzione, allora il tipo passato ad **addScope** è quello della funzione dichiarata; infine se lo scope viene aggiunto dalla creazione di un blocco di statements, allora il tipo passato ad **addScope** è quello dello scope in cui il blocco di statements è racchiuso. Non è difficile vedere che tali accorgimenti verificano la proprietà stabilità per il tipo di uno scope, ossia che il tipo di uno scope coincide con il tipo della funzione in cui è racchiuso.

Forniamo ora una breve panoramica del funzionamento del modulo di analisi statica **StaticAnalysis.hs**. La funzione principale del modulo è **typeCheck** che preso un programma rappresentato in sintassi astratta di **Scala40** ritorna tale programma annotato ed eventualmente una lista di log. Il compito di annotare gli elementi della sintassi astratta e di aggiungere eventuali log alla lista di log ogni qual volta venga rilevata una eccezione è affidato alle funzioni **infer*** (per esempio: **inferDecl**, **inferStm**, **inferExp**, etc.), tali funzioni preso un elemento della sintassi astratta effettuano i controlli per verificare che tale elemento sia coerente con le specifiche del linguaggio **Scala40** e nel caso non lo siano tramite la funzione **saveLog** si occupano di aggiornare la lista di log; al termine dei controlli l'elemento analizzato viene ritornato arricchito con annotazioni quali le informazioni sugli identificatori utilizzati e il tipo delle espressioni coinvolte.

Le funzioni **infer*** hanno tutte un comportamento naturalmente ricorsivo, si prenda per esempio il caso di una dichiarazione di variabile con assegnamento di un valore, per poter inferire la dichiarazione e determinare se è valida o meno è necessario inferire l'espressione che si sta cercando di assegnare alla variabile e verificare che i tipi combacino.

Per una nostra decisione personale, nel caso in cui si stia inferendo un elemento che coinvolge una espressione la quale abbiamo già verificato essere problematica, non vengono aggiunte al log nuove eccezioni riguardanti l'elemento sotto-analisi. Questa scelta è giustificata dal desiderio di eliminare ridondanza negli errori, dato che una espressione errata, potrebbe portare ad una lunga serie di errori a catena in tutti gli elementi in cui è compresa.

Per ottenere questo comportamento, abbiamo nella sintassi astratta un tipo interno **SType_Error** che viene assegnato alle espressioni che generano un'eccezione, o alle espressioni le cui sotto-espressioni generano eccezioni. Se un qualsiasi elemento ha come sotto-espressione una espressione di tipo **SType_Error** non

vengono generate ulteriori eccezioni riguardo all'interazione di tale sotto-espressione con l'elemento considerato.

Le annotazioni aggiunte dal modulo di analisi statica riguardano esclusivamente le espressioni, sia R-espressioni (elementi **Exp**), sia L-espressioni (elementi **LExp**). Le espressioni annotate sono identificate rispettivamente dagli elementi di sintassi astratta interni: **ETyped** per le R-espressioni e da **LExpTyped** per le L-espressioni. Entrambi gli elementi condividono la stessa struttura, una espressione tipata contiene la espressione, il suo tipo, e la locazione in cui viene utilizzata. Un'ulteriore caratteristica delle espressioni tipate è che la locazione contenuta all'interno di un identificatore utilizzato all'interno dell'espressione non è la locazione di utilizzo, che invece è in un campo apposito, ma è la locazione di dichiarazione dell'identificatore. La locazione di dichiarazione di un identificatore utilizzato all'interno di una espressione ci risulta fondamentale per la successiva fase di generazione del codice TAC. La definizione degli elementi tipati con le loro proprietà è presente all'interno del modulo **Typed.hs**.

2.2 Descrizione soluzione - Three Address Code

Il Three Address Code (TAC) viene costruito a partire dall'albero annotato risultante dall'analisi di semantica statica (Type Checking). Per la costruzione viene fatto largo uso della monade State, descritta nella sezione precedente. In particolare lo stato ha quattro campi: il numero di temporanei e di etichette finora utilizzati, la lista di istruzioni che compongono il TAC e una lista di liste di istruzioni (utili per la gestione degli scope e delle funzioni innestate).

Nel TAC gli indirizzi utilizzati nelle istruzioni sono:

- Letterali: per valori costanti di tipo base.
- Indirizzi di variabile e di funzioni: hanno la seguente forma,

`ident@loc`

dove `ident` è il loro identificatore e `loc` la locazione di dichiarazione.

- Temporanei: utilizzati per identificare espressioni.

Inoltre, nel TAC sono presenti le etichette che vengono utilizzate per indicare una locazione univoca nel flusso di esecuzione. Nel caso delle funzioni, come etichetta, viene utilizzato il loro indirizzo.

La costruzione del Three Address Code inizia dalla funzione

`genProg :: Program -> Bool -> TacState ()`

che ha i seguenti compiti:

- Inserire un'etichetta in fondo a tutte le istruzioni (e il relativo **Goto** dopo le dichiarazioni globali) se durante l'analisi di semantica statica non è stato trovato un metodo main da cui far partire la computazione. In alternativa, viene inserita un'istruzione **Goto** all'etichetta del main;

Tipo	Valore di default
Int	0
Float	0.0
Char	\0
String	""
Bool	False
Pointer	Null
Array	-

Table 1: Valori di default per dichiarazione senza assegnamento

- iniziare la discesa nell'albero annotato per poter costruire le singole istruzioni, tramite la funzione

```
genDecls :: [Declaration] -> TacState ()
```

La funzione `genDecls` prende in input una lista di dichiarazioni e le scorre producendo le relative istruzioni TAC. Questo avviene tramite la funzione

```
genDecl :: Declaration -> TacState ()
```

la quale divide le dichiarazioni nei quattro casi possibili:

- Dichiarazione di variabile con assegnamento: viene creato un nuovo indirizzo a partire dall'identificatore della variabile e dalla sua locazione di dichiarazione. Gli viene assegnato un valore tramite la funzione `genExpAssign` che permette di evitare la creazione di temporanei inutili quando ci troviamo nel caso di un'espressione semplice (es. $x = 3$, $*y = x$, $x = y + z$ ecc.).
- Dichiarazione di variabile: come nel caso precedente costruiamo un indirizzo a partire da identificatore e locazione, e gli assegniamo un valore di default () tramite la funzione `buildDefaultValue`.
- Dichiarazione di funzione/procedura: in questo caso viene creata una nuova etichetta per la funzione. Viene chiamata la funzione `genBlock` che (tramite `genStms`) controlla la presenza dell'istruzione `return` (nel caso delle procedure verrà aggiunto un'istruzione di return vuoto) e costruisce tutte le istruzioni TAC relative al corpo della funzione. Al termine di questo processo le istruzioni TAC che sono state create vengono estratte tramite operazione di pop dalla lista di liste di istruzioni dello stato e inserite in testa al codice globale del TAC. Questo per far sì che una funzione dichiarata nel corpo di un'altra venga inserita sopra quest'ultima a livello di Three Address Code, e non al suo interno.

Ogni funzione è costituita da un insieme di statement (contenuti in un blocco). La funzione che si occupa di percorrere tutti gli statement di un blocco è

`genStms :: [Stm] -> TacState Bool`

che ritorna `True` se l'ultimo statement è un `return` (per evitare che nel TAC compaiano due istruzioni `return` consecutive). Nel contempo vengono esaminati gli statement singolarmente tramite la funzione

`genStm :: Stm -> TacState ()`

Quest'ultima prende in esame tutti i possibili statement:

- Dichiarazione: viene riutilizzata `genDecl` per la gestione delle dichiarazioni interne al blocco.
- Blocchi interni: viene riutilizzata la funzione `genBlock`.
- Assegnamenti: un assegnamento ha la seguente forma:

`Lexpression = Rexpression`

Le `LExpression` comprendono: accessi ad array, variabili puntatori e identificatori di variabile. La loro gestione avviene tramite la funzione `genLexp`. Le `RExpression` oltre che le `LExpression`, comprendono le espressioni.

- Chiamata di procedura: vengono create le istruzioni che indicano i parametri passati alla funzione (tramite la funzione `genParams` che crea istruzioni `Param addr`) e viene creata un'istruzione `Call` del TAC.
- Return: se l'istruzione `return` ritorna un'espressione essa viene gestita tramite `genExp`, che in caso di espressioni complesse (non binaria tra due identificatori o unaria) restituisce un temporaneo.
- If e While: in entrambi i casi la gestione delle etichette viene gestita tramite la funzione

`genCondition :: Exp -> Label -> Label -> TacState ()`

che, presa l'espressione della condizione e le due etichette che indicano dove spostarsi in caso essa sia vera o meno, gestisce la creazione dei `Goto` ed esegue il controllo del flusso.

La generazione delle istruzioni riguardanti le espressioni viene gestita tramite la funzione

`genExp :: Exp -> TacState Addr`

che ritorna l'indirizzo TAC dell'espressione: un letterale nel caso di espressioni composte da un solo letterale o un nuovo temporaneo nel caso di dereferenziazioni, `LExpression`, accessi ad array o espressioni simili a quelle descritte nel caso dell'assegnamento (gestite tramite la funzione `genExpAssign`).

Terminate tutte le dichiarazioni presenti nello scope globale, vengono inserite in testa alla lista di istruzioni TAC tutte le dichiarazioni di variabili globali seguite dall'istruzione `Goto` alla funzione `main` se presente, oppure all'etichetta 10 in fondo a tutte le istruzioni (evitando così che tutte le altre funzioni vengano eseguite sequenzialmente).

3 Caratteristiche generali di Scala40

TODO

4 Struttura lessicale di Scala40

Parole riservate

Le parole riservate in **Scala40** sono le seguenti:

if, else, do, while, def, return, var, Array, False, True,
Null, Char, String, Int, Float, Bool.

Identificatori

Un identificatore $\langle Ident \rangle$ è una lettera seguita da una sequenza arbitraria di lettere, cifre e del carattere `'_'`.

Letterali

Vi sono letterali per numeri interi, numeri in virgola mobile, singoli caratteri, booleani, stringhe. Essi seguono le convenzioni della maggior parte dei linguaggi di programmazione.

$$\begin{aligned} \langle Literal \rangle ::= & \langle Int \rangle \\ & | \langle Float \rangle \\ & | \langle Char \rangle \\ & | \langle Bool \rangle \\ & | \langle String \rangle \\ & | Null \end{aligned}$$

Commenti

I commenti in **Scala40** sono di due tipi:

- i commenti di una riga sono sequenze di caratteri che iniziano con `\` e finiscono al termine della riga;
- i commenti multi-riga sono sequenze di caratteri che iniziano con `/*` e terminano con `*/`. Non possono essere annidati.

Caratteri di spaziatura

I token possono essere separati dai caratteri di spaziatura standard o commenti.

5 Struttura sintattica di Scala40

- Un *programma* è una sequenza di dichiarazioni.
- Una *dichiarazione* ha una delle seguenti forme:

– *Dichiarazione di variabili*

$$\langle Decl \rangle ::= \text{var } \langle Ident \rangle : \langle TypeSpec \rangle = \langle Expr \rangle ; \\ | \text{var } \langle Ident \rangle : \langle TypeSpec \rangle ;$$

– *Dichiarazione di funzioni e procedure*

$$\langle Decl \rangle ::= \text{def } \langle Ident \rangle \langle ParamClauses \rangle : \langle TypeSpec \rangle = \langle Expr \rangle ; \\ \text{def } \langle Ident \rangle \langle ParamClauses \rangle : \langle TypeSpec \rangle = \langle Block \rangle \\ \text{def } \langle Ident \rangle \langle ParamClauses \rangle = \langle Expr \rangle ; \\ \text{def } \langle Ident \rangle \langle ParamClauses \rangle = \langle Block \rangle$$

dove $\langle TypeSpec \rangle$ è una specifica di tipo, che ha la forma

$$\langle TypeSpec \rangle ::= \langle SimpleType \rangle \\ | * \langle TypeSpec \rangle \\ | \text{Array } [\langle TypeSpec \rangle] (\langle Int \rangle) \\ \langle SimpleType \rangle ::= \text{Bool} | \text{Char} | \text{Int} | \text{Float} | \text{String}$$

mentre l'elemento $\langle ParamClauses \rangle$ è una sequenza, non vuota, di $\langle ParamClause \rangle$, e ciascun $\langle ParamClause \rangle$ ha la forma

$$\langle ParamClause \rangle ::= (\langle Args \rangle)$$

$\langle Args \rangle$ è una sequenza, che può essere vuota, di elementi separati da virgola della forma

$$\langle Arg \rangle ::= \langle Ident \rangle : \langle TypeSpec \rangle$$

Ad esempio, una specifica di tipo valida è `Array[*Int](2)`, che indica un array di puntatori ad interi di dimensione 2. Ad esempio, una definizione di funzione valida è

```
def foo(a: Array[*Int](2),p: *Int)(x: Int): Int =
  *a[1] + *a[2] + *p + x;
```

Essa prende come parametri un array di puntatori ad interi, un puntatore ad un intero e un intero e restituisce un intero.

- Un *blocco* è una sequenza di istruzioni racchiuse fra parentesi graffe.

$$\langle Block \rangle ::= \{ \langle StmtList \rangle \}$$

- Una *istruzione* ha la forma:

$$\begin{aligned}
\langle Stmt \rangle ::= & \langle Decl \rangle \\
& | \langle Block \rangle \\
& | \langle LExpr \rangle \langle OpAssign \rangle \langle Expr \rangle ; \\
& | \text{if} (\langle Expr \rangle) \langle Stmt \rangle \\
& | \text{if} (\langle Expr \rangle) \langle Stmt \rangle \text{ else } \langle Stmt \rangle \\
& | \text{while} (\langle Expr \rangle) \langle Stmt \rangle \\
& | \text{do } \langle Stmt \rangle \text{ while} (\langle Expr \rangle) ; \\
& | \text{return} ; \\
& | \text{return } \langle Expr \rangle ; \\
& | \langle Ident \rangle \langle ParamsList \rangle ;
\end{aligned}$$

dove, nell'ultima istruzione, che corrisponde alla chiamata di procedura o funzione, $\langle ParamsList \rangle$ è una sequenza, non vuota, di $\langle Params \rangle$ della forma

$$\langle Params \rangle ::= (\langle ExprList \rangle)$$

e $\langle ExprList \rangle$ è una sequenza, che può essere vuota, di $\langle Expr \rangle$ separate da virgola.

Invece $\langle OpAssign \rangle$ è uno dei seguenti operatori di assegnamento:

$$\langle OpAssign \rangle ::= = | += | -= | *= | /= | \%=$$

- Le *left expressions* del linguaggio hanno la seguente forma:

$$\begin{aligned}
\langle LExpr \rangle ::= & \langle Ident \rangle \\
& | \langle LExpr \rangle [\langle Expr \rangle] \\
& | * \langle LExpr \rangle \\
& | (\langle LExpr \rangle)
\end{aligned}$$

L'operatore accesso ad array $[]$ ha la precedenza sull'operatore di dereference $*$. Quindi ad esempio $*a[1]$ è sintatticamente equivalente a $*(a[1])$.

- Le *right expressions* del linguaggio hanno la seguente forma:

$$\begin{aligned}
\langle Expr \rangle ::= & \langle Literal \rangle \\
& | \langle LExpr \rangle \\
& | \& \langle LExpr \rangle \\
& | \text{Array} (\langle ExprList \rangle) \\
& | \langle Ident \rangle \langle ParamsList \rangle \\
& | \langle Expr \rangle \langle BinOp \rangle \langle Expr \rangle \\
& | \langle UnOp \rangle \langle Expr \rangle \\
& | (\langle Expr \rangle)
\end{aligned}$$

$$\langle BinOp \rangle ::= | | \mid \&\& \mid < \mid <= \mid > \mid >= \mid == \mid != \mid + \mid - \mid * \mid / \mid \% \mid ^$$

$$\langle UnOp \rangle ::= ! \mid -$$

Gli operatori hanno precedenze e associatività standard.

6 Vincoli di semantica statica di Scala40

TODO