Progetto 2020
Compilatori e Interpreti

Giacchè Alessandro, Boccuto Alessandra, Celozzi Michele

3 settembre 2020

Indice

1 Esercizio 1

Realizzare un sistema di Analisi Semantica per il linguaggio SimplePlus allegato.

In particolare, il sistema deve controllare:

1. variabili/funzioni non dichiarate;
2. variabili dichiarate più volte nello stesso ambiente
   (in questa analisi è corretto il codice `int x = 4; delete x; int x = 5;`);
3. parametri attuali non conformi ai parametri formali (inclusa la verifica sui parametri passati per riferimento);
4. la correttezza dei tipi;
5. accessi a identificatori "cancellati" con particolare attenzione all'aliasing.

1.0 Regole di inferenza

1.0.1 Type Inference

Di seguito vengono elencate le regole di inferenza per il type checking utilizzate all'interno dell'esercizio.

$$\text{SeqArg}\ \frac{T \neq void \qquad x \notin dom(top(\Gamma)) \qquad \Gamma[x \mapsto T] \vdash A : \Gamma'}{\Gamma \vdash T\ x,\ A : \Gamma'}$$

$$\text{DecFun}\ \frac{id \notin dom(top(\Gamma)) \qquad \Gamma|_{\text{FUN}} \cdot [\,] \vdash T_1\ x_1, ..., T_n\ x_n : \Gamma|_{\text{FUN}} \cdot \Gamma' \qquad \Gamma|_{\text{FUN}} \cdot \Gamma' \vdash B : T' \qquad T' = T}{\Gamma \vdash T\ id(T_1\ x_1,\ \ldots\ ,\ T_n\ x_n)\ \{\ B\ \}\ : \Gamma[id \mapsto T_1 \times ... \times T_n]}$$

$$\text{DecVar}\ \frac{\cancel{id \notin dom(top(\Gamma))}\ (*) \qquad T \neq void}{\Gamma \vdash T\ id : \Gamma[id \mapsto T]}$$

(*) NOTA: L'esistenza dell'id in top di gamma non viene controllata perchè il controllo viene effettuato successivamente durante l'analisi degli effetti. Questo meccanismo è necessario per essere certi che la variabile precedentemente dichiarata non sia stata, in realtà, cancellata.

$$\text{DecVarAssigment}\frac{\Gamma \vdash e : T' \quad \Gamma \vdash T\ id : \Gamma' \quad T' = T}{\Gamma \vdash T\ id = e : \Gamma'} \qquad \text{Assign}\frac{\Gamma(id) = T \quad \Gamma \vdash e : T' \quad T = T'}{\Gamma \vdash id = e : void}$$

$$\text{Delete}\frac{x \in dom(top(\Gamma)) \quad \Gamma(x) = T' \quad T' \neq T_1 \times ... \times T_n \to T}{\Gamma \vdash delete\ x : void} \qquad \text{Print}\frac{\Gamma \vdash e : T \quad T \neq void}{\Gamma \vdash print\ e : void}$$

$$\text{ReturnExp}\frac{\Gamma \vdash e : T}{\Gamma \vdash return\ e : T} \qquad \text{Return}\frac{}{\Gamma \vdash return : void}$$

$$\text{Ite}\frac{\Gamma \vdash e1 : bool \quad \Gamma \cdot [\,] \vdash s_1 : T \quad \Gamma \cdot [\,] \vdash s_2 : T' \quad T = T'}{\Gamma \vdash if\ (e1)\ s_1\ else\ s_2 : T}$$

$$\text{Call}\frac{\Gamma(id) = T_1 \times ... \times T_n \to T \quad (\Gamma \vdash e_i : T'_i \quad T'_i < T_i)_{i \in 1...n}}{\Gamma \vdash id(e_1, ..., e_n) : T}$$

$$\text{BaseExp}\frac{\Gamma \vdash e_1 : T}{\Gamma \vdash (e_1) : T} \qquad \text{NegExp}\frac{\Gamma \vdash e_1 : T \quad T = int}{\Gamma \vdash -e_1 : T}$$

$$\text{NotExp}\frac{\Gamma \vdash e_1 : T \quad T = bool}{\Gamma \vdash !e_1 : T} \qquad \text{ValExp}\frac{}{\Gamma \vdash number : int}$$

$$\text{BoolExp}\frac{}{\Gamma \vdash value : bool} \qquad \text{VarExp}\frac{\Gamma(id) = T}{\Gamma \vdash id : T}$$

$$\text{Plus}\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = int = T_2 \quad + : int \times int \to int}{\Gamma \vdash e_1 + e_2 : T_1}$$

NOTA: data la somiglianza tra le operazioni aritmetiche (+, -, *, /) e per evitare un'eccessiva ripetizione, le altre regole di inferenza sono state omesse.

$$\text{GreaterThan}\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = int = T_2 \quad > : int \times int \to bool}{\Gamma \vdash e_1 > e_2 : bool}$$

NOTA: data la somiglianza tra le operazioni booleane $(>, \geq, <, \leq)$ e per evitare un'eccessiva ripetizione, le altre regole di inferenza sono state omesse.

$$\text{Equal}\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = T_2 \quad == : T_1 \times T_2 \to bool}{\Gamma \vdash e_1 == e_2 : bool}$$

NOTA: data la somiglianza con $\neq$, la regola relativa a quest'ultimo è stata omessa.

$$\text{And}\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = bool = T_2 \quad \&\& : T_1 \times T_2 \to bool}{\Gamma \vdash e_1\ \&\&\ e_2 : bool}$$

NOTA: data la somiglianza con ||, la regola relativa a quest'ultimo è stata omessa.

1.0.2 Behaviour Analysis

Di seguito vengono elencate le regole di inferenza per la behaviour analysis utilizzate all'interno dell'esercizio.

$$\text{Block}\frac{\Sigma \cdot [\,] \vdash S : \Sigma'}{\Sigma \vdash \{\ S\ \} : \Sigma'} \qquad \text{SeqS}\frac{\Sigma \vdash s : \Sigma' \quad \Sigma' \vdash S : \Sigma''}{\Sigma \vdash s\ S : \Sigma''}$$

$$\text{DecVar}\frac{x \notin dom(top(\Sigma)) \vee \Sigma(x) = d}{\Sigma \vdash T\ x : \Sigma[x \mapsto \bot]} \qquad \text{DecVarAssign}\frac{\Sigma \vdash T\ x : \Sigma' \quad \Sigma' \vdash e : \Sigma''}{\Sigma \vdash T\ x = e : \Sigma'' \triangleright [x \mapsto rw]}$$

$$\text{Assign}\frac{\Sigma \vdash e : \Sigma'}{\Sigma \vdash x = e : \Sigma' \triangleright [x \mapsto rw]} \qquad \text{Print}\frac{\Sigma \vdash e : \Sigma'}{\Sigma \vdash print\ e : \Sigma'}$$

$$\text{Return}\frac{}{\Sigma \vdash return : \Sigma} \qquad \text{ReturnExp}\frac{\Sigma \vdash e : \Sigma'}{\Sigma \vdash return\ e : \Sigma'}$$

$$\text{Delete}\frac{}{\Sigma \vdash delete\ x : \Sigma \triangleright [x \mapsto d]}$$

$$\text{Ite}\frac{\Sigma \vdash e : \Sigma' \quad \Sigma' \vdash s_1 : \Sigma_1 \quad \Sigma' \vdash s_2 : \Sigma_2}{\Sigma \vdash if(e)\ s_1\ else\ s_2 : max(\Sigma_1, \Sigma_2)}$$

NOTA: se s_1 o s_2 sono blocchi, la generazione di un nuovo scope è delegata alla regola "Block"; se uno dei due statements è, invece, una singola istruzione, allora viene aperto un nuovo scope direttamente dalla regola in oggetto.

$$\text{DecFun}\frac{\begin{array}{c}\Sigma_0 = [x_1 \mapsto \bot,\ \ldots\ , x_m \mapsto \bot,\ y_1 \mapsto \bot,\ \ldots\ ,\ y_n \mapsto \bot] \\ \Sigma|_{\text{FUN}} \cdot \Sigma_0[f \mapsto \Sigma_1] \vdash s : \Sigma|_{\text{FUN}} \cdot \Sigma_1[f \mapsto \Sigma_1]\end{array}}{\Sigma \vdash T\ f(T_1\ var\ x_1,\ \ldots\ ,\ T_m\ var\ x_m,\ T'_1\ y_1,\ \ldots\ ,\ T'_n\ y_n)\ s : \Sigma[f \mapsto \Sigma_1]}$$

NOTA: la regola sovrastante richiede intrinsecamente l'applicazione dell metodo del punto fisso per il calcolo di Σ_1.

$$\text{FunCall}\frac{\begin{array}{c}\Gamma \vdash f : \&T_1 \times \ldots \times \&T_m \times T'_1 \times \ldots \times T'_n \to T'' \\ \Sigma(f) = \Sigma_1 \qquad \Sigma' = \otimes_{i\,\in\,1\ldots m}[u_i \mapsto \Sigma(u_i) \triangleright \Sigma_1(x_i{}^{(*)})] \\ (\Sigma \vdash e_i : \Sigma'')_{i\,\in\,1\ldots n} \qquad \Sigma''' = update(\Sigma', \Sigma'')^{(**)}\end{array}}{\Sigma \vdash f(u_1,\ \ldots\ ,\ u_m,\ e_1,\ \ldots\ ,\ e_n)\ : update(\Sigma, \Sigma''')}$$

NOTE:

(*) "x" rappresenta il parametro formale nella dichiarazione di funzione.

(**) Nella regola di inferenza, i parametri passati per riferimento e quelli passati per valore sono ben distinti. Questo non corrisponde all'implementazione effettiva.

1.1 Controllo variabili e funzioni non dichiarate

Il controllo relativo all'utilizzo di variabili e all'invocazione di funzioni non dichiarate viene effettuato nel metodo "checkSemantics" delle classi "ExpVar", "StmtAssignment", "StmtDelete" e "StmtCall".

In particolare:

- in "ExpVar" viene controllato che la variabile utilizzata esista all'interno di uno scope;
- in "StmtAssignment" viene controllato che la variabile a cui si sta assegnando un valore sia all'interno dell'environment;
- in "StmtDelete" viene controllato che, se la variabile è locale (cioè appartenente all'ultimo scope aperto), non risulti eliminata e non sia un parametro (dato che è possibile cancellare solo i parametri richiesti per riferimento).
  Se la variabile non è locale, invece, viene verificata la sua effettiva esistenza in un qualsiasi altro scope più esterno e, se il controllo va a buon fine, che la variabile in oggetto non sia già stata cancellata e che sia un riferimento (dato che non è possibile cancellare variabili non locali all'interno del proprio scope).
  È interessante osservare che al metodo "checkSemantics" del blocco ("StmtBlock") appartenente a "StmtDecFun" viene passato un environment composto dalle sole dichiarazioni di funzioni. Non è pertanto possibile, trovandosi in una funzione annidata, cancellare un parametro definito nella funzione "padre";
- in "StmtCall" viene controllato che l'id della funzione richiamata esista in un qualche scope e che questo corrisponda effettivamente ad una funzione.

Listing 1: A titolo d'esempio, viene riportato il codice "checkSemantics" di "ExpVar"

```
public List<SemanticError> checkSemantics(Environment<STEntry> e) {
    List<SemanticError> toRet = new LinkedList<SemanticError>();

    idEntry = e.getIDEntry(id);

    if(idEntry == null)
        toRet.add(new VariableNotExistsError(id, line, column));

    return toRet;
}
```

1.2 Dichiarazioni multiple di variabili nello stesso ambiente

La gestione di dichiarazioni multiple di variabili all'interno dello stesso ambiente è implementata dal metodo "inferBehaviour" di "StmtDecVar".
La gestione di questa problematica non è stata effettuata in "checkSemantics" in quanto una variabile dichiarata precedentemente potrebbe essere stata eliminata, quindi si otterrebbero dei falsi negativi.
Spostando il controllo in "inferBehaviour", inoltre, si ha la possibilità di assicurarsi che la variabile in esame non sia stata cancellata durante l'esecuzione di funzioni a cui è stata passata per riferimento.

Nello specifico, per questo controllo si verifica l'esistenza della variabile all'interno dell'environment: se questa esiste, significa che la variabile è già stata dichiarata, perciò viene controllato che questa sia stata successivamente cancellata e, se lo è, il suo stato viene reimpostato a "Bottom".
Se la variabile non è mai stata dichiarata, invece, viene semplicemente aggiunta una entry allo scope corrente.
Se nessuno dei casi precedentemente esposti è vero significa, ovviamente, che la variabile è stata dichiarata senza essere stata mai cancellata e viene pertanto ritornato un errore.

Listing 2: "inferBehaviour" di "StmtDecVar"

```
public List<BehaviourError> inferBehaviour(Environment<BTEntry> e) {
    // ...

    BTEntry prev = e.getLocalIDEntry(ID);

    if (prev != null) {
        // If was not deleted, there's an error
        if (prev.getLocalEffect().compareTo(EEffect.D) < 0) {
            prev.setLocalEffect(EEffect.T);
            toRet.add(new IdAlreadytExistsError(ID, line, column));
        }
        // If it is not top, add the "BOTTOM" effect
        else if (prev.getLocalEffect().compareTo(EEffect.D) == 0)
            e.getIDEntry(ID).addEffect(EEffect.BOTTOM);
    } else
        e.add(ID, new BTEntry());

    // ...
}
```

1.3 Parametri attuali non conformi ai parametri formali

L'implementazione della verifica della conformità dei parametri attuali con quelli formali è stata effettuata nei metodi "checkSemantics" e "inferType" della classe "StmtCall".
Tramite il controllo semantico, si verifica che il numero dei parametri attuali sia lo stesso di quelli formali e che vengano passate variabili in corrispondenza dei parametri formali che richiedono valori per riferimento. Il type checking, a sua volta, controlla che i tipi dei parametri formali siano gli stessi di quelli passati.

Listing 3: "checkSemantics" di "StmtCall"

```
public List<SemanticError> checkSemantics(Environment<STEntry> e) {
    // ...
            // Parameters check
            List<Type> params = ((ArrowType) funT).getParamTypes();
            if (exps.size() != params.size())
                toRet.add(new ParametersMismatchError(params.size(),
                    ↪ exps.size(), line, column));

            // Check that for each "reference" parameter is passed a variable
            for (int i = 0; i < Math.min(exps.size(), params.size()); i++)
                if (params.get(i).isRef() && !(exps.get(i) instanceof ExpVar))
                    toRet.add(new PassedExpNotVariableError(i + 1, ID,
                        ↪ exps.get(i).line, exps.get(i).column));
    // ...
}
```

Listing 4: "inferType" di "StmtCall"

```
public Type inferType() {
   // ...

   List<Type> parTypes = funT.getParamTypes();

   for (int i = 0; i < parTypes.size(); i++) {
      // Get parameter type
      Type parType = parTypes.get(i);
      // Get expression type
      Type expType = exps.get(i).inferType();
      // Check the equality
      if (!parType.getType().equalsTo(expType))
         TypeErrorsStorage.addError(new TypeError("#" + (i + 1) + " parameter
            ↪ type (" + parType + ") is not equal to expression type (" +
            ↪ expType + ")", line, column));
   }

   // ...
}
```

1.4 Controllo correttezza dei tipi

Tutte le classi che vanno a comporre l'AST collaborano per il controllo della correttezza dei tipi.

I controlli sui tipi che sono stati effettuati sono elencati di seguito:

- Un parametro non può essere di tipo "void" – (Arg);
- Le espressioni devono essere coerenti con il loro utilizzo (un'espressione aritmetica deve avere tutti gli elementi di tipo "int", l'operazione di negazione deve avvenire con un tipo "bool", ecc...). Questi accertamenti vengono effettuati dalle rispettive classi con prefisso "Exp" – (Exp*);
- L'espressione assegnata ad una variabile deve avere lo stesso tipo di quest'ultima – (StmtAssignment);
- Se in un blocco sono presenti più "return", questi devono ritornare lo stesso tipo.
  Se viene ritornato un valore diverso da "void" dal branch "then" di un "if" e non vi sono altri return nel blocco (quindi la funzione potrebbe – talvolta – non ritornare alcun valore) l'elemento "if" in analisi deve anche avere un branch "else" – (StmtBlock);
- I tipi dei parametri passati ad una funzione devono essere conformi a quelli formali – (StmtCall);
- Il tipo ritornato dal blocco di una funzione deve essere lo stesso del tipo di ritorno della funzione stessa – (StmtDecFun);
- Il tipo di una variabile non può essere "void" e, nel caso in cui assieme alla dichiarazione venisse fatta anche una definizione, il tipo del valore assegnato deve essere lo stesso del tipo della variabile dichiarata – (StmtDecVar);
- Non è possibile eliminare una funzione – (StmtDelete);
- La condizione di un "if" deve essere di tipo "bool", inoltre, se l'"if" in questione possiede un branch "else", i tipi di ritorno dei due rami devono coincidere – (StmtIte);
- Non è possibile stampare espressioni di tipo "void" – (StmtPrint).

È degno di nota che i tipi disponibili all'interno del linguaggio generato sono "void", "int" e "bool".
Ai tipi sopraelencati, tuttavia, si aggiunge – per i metodi "inferType" – la possibilità di ritornare "null" nel caso in cui l'operazione su cui è stato richiamato il metodo non causi l'uscita dal blocco corrente e non abbia perciò peso sul tipaggio del blocco. A titolo d'esempio, mentre "return;" ritorna effettivamente un "void", l'istruzione "print 1;" non interrompe la linearità dell'esecuzione e, pertanto, il suo "inferType" ritorna "null".
Questo meccanismo è stato utile al fine di riconoscere, all'interno di un blocco, quali valori causassero l'effettiva uscita dal blocco in fase di esecuzione e quali, invece, la continuassero linearmente.

Listing 5: "inferType" di "StmtIte", a titolo di esempio

```
public Type inferType() {
    if (!EType.BOOL.equalsTo(exp.inferType()))
        TypeErrorsStorage.addError(new TypeError("Condition must be bool type",
            ↪ exp.line, exp.column));

    Type thenT = this.thenStmt.inferType();

    if (elseStmt == null)
        return thenT;

    Type elseT = this.elseStmt.inferType();

    // If elseT is null and thenT is not null or otherwise, then add an error.
    // Add an error also if the branch types does not correspond
    if (elseT != null && !elseT.getType().equalsTo(thenT) || thenT != null &&
        ↪ !thenT.getType().equalsTo(elseT))
        TypeErrorsStorage.addError(new TypeError("Then branch (" + thenT + ")
            ↪ does not return the same type of else branch (" + elseT + ")",
            ↪ this.thenStmt.line, this.thenStmt.column));

    return thenT;
}
```

1.5 Controllo accesso ad identificatori "cancellati" e aliasing

1.5.1 Controllo accesso ad identificatori "cancellati"

Al fine di gestire l'accesso ad identificatori cancellati, vengono effettuati controlli ogni qual volta si tenta di effettuare l'accesso ad una variabile.
Il metodo "inferBehaviour" delle classi "ExpVar" e "StmtAssignment" esegue l'operazione *seq* tra il behaviour state corrente e "RW", controlla poi se questo ritorna "T" e, in tal caso, genera un errore.
Anche il metodo "inferBehaviour" della classe "StmtDelete" effettua una *seq* partendo dal behaviour state corrente della variabile da cancellare ma, anziché applicarlo a "RW", lo applica a "D" facendo poi lo stesso tipo di controllo di cui sopra.
La classe sopracitata, oltre che in "inferBehaviour", effettua un controllo preliminare relativo alle variabili già eliminate anche in "checkSemantics": questa verifica è meno fine di quella effettuata in "inferBehaviour" ma è utile per evitare di effettuare gli step di compilazione successivi in un codice in cui sono già stati riconosciuti errori.
Quello che rende l'analisi di "inferBehaviour" più approfondita di quella effettuata in "checkSemantics" risiede nel fatto che, differentemente dal controllo effettuato in "checkSemantics", l'analisi del comportamento tiene traccia dell'effetto delle altre funzioni sulla variabile in oggetto (banalmente, se una funzione cancella la variabile, questo effetto viene percepito da "inferBehaviour" ma non da "checkSemantics"); questo permette ad "inferBehaviour" di essere più precisa, rimuovendo i falsi positivi.

Listing 6: "checkSemantic" ed "inferBehaviour" di "StmtDelete"

```
public List<SemanticError> checkSemantics(Environment<STEntry> e) {
    // ...

    if (candidate.isDeleted()) {
        toRet.add(new VariableAlreadyDeletedError(id, line, column));
        return toRet;
    }

    // ...
}
public List<BehaviourError> inferBehaviour(Environment<BTEntry> e) {
    // ...

    e.getIDEntry(id).setLocalEffect(BTHelper.seq(e.getIDEntry(id)
        .getLocalEffect(), EEffect.D));

    if (e.getIDEntry(id).getLocalEffect().equals(EEffect.T))
        toRet.add(new DeletedVariableError(id, line, column));

    // ...
}
```

1.5.2 Controllo problemi di aliasing

Il controllo di problemi legati all'aliasing viene effettuato analizzando gli effetti derivati dall'invocazione di una funzione ("StmtCall").
L'implementazione di tale controllo è stata effettuata seguendo pedissequamente la teoria: vengono recuperati tutti i parametri richiesti per riferimento (*var*) dalla funzione, per ognuno di questi si ottiene lo stato attuale della variabile passata (cioè quello precedente l'invocazione) e quello che la funzione "provoca" al parametro corrispondente una volta richiamata. Viene poi effettuata l'operazione *seq* tra i valori recuperati e il risultato viene registrato in un array associato al nome della variabile passata.

Al termine del ciclo si raggiunge un insieme di associazioni del tipo "<id $_{\text{variabile}}$, lista $_{\text{effetti provocati dalla funzione}}$>": per ogni elemento di questo insieme, viene effettuata l'operazione *par* tra tutti gli elementi della lista degli effetti e il risultato di questo calcolo viene salvato come effetto nella variabile a cui la lista fa riferimento.
In ultimo, viene controllata l'esistenza di behaviour states a "T" e, in tal caso, si genera un errore.
L'operazione di *update* (presente nelle regole di inferenza) corrisponde al `setLocalEffect` effettuato all'inteno del `forEach` alla fine del metodo.

Listing 7: "inferBehaviour" di "StmtCall"

```
public List<BehaviourError> inferBehaviour(Environment<BTEntry> e) {
    // ...

    // Sigma_1
    List<BTEntry> e1 = e.getIDEntry(ID).getE1();

    // Sigma'
    HashMap<String, Tuple<Integer, List<EEffect>>> eStar = new HashMap<>();

    for (int i = 0; i < parsTypes.size(); i++) {
        if (parsTypes.get(i).isRef()) {
            BTEntry prev = e.getIDEntry(((ExpVar) exps.get(i)).getId());
            BTEntry next = e1.get(i);

            List<EEffect> btList = eStar.getOrDefault(((ExpVar)
                ↪ exps.get(i)).getId(), new Tuple<>(i, new LinkedList<>())).y;

            btList.add(BTHelper.invocationSeq(prev, next));
            eStar.put(((ExpVar) exps.get(i)).getId(), new Tuple<>(i, btList));
        }
        // check not-referenced variables
        else
            toRet.addAll(exps.get(i).inferBehaviour(e));
    }

    eStar.entrySet().forEach(entry -> {
        e.getIDEntry(entry.getKey()).setLocalEffect(entry.getValue().y.stream()
        .reduce((a, b) -> BTHelper.par(a, b)).get());

        expsBehaviour.set(entry.getValue().x,
            ↪ e.getIDEntry(entry.getKey()).getLocalEffect());

        if (e.getIDEntry(entry.getKey()).getLocalEffect().compareTo(EEffect.T)
            ↪ == 0)
            toRet.add(new AliasingError(entry.getKey(), ID, line, column));
    });

    return toRet;
}
```

1.6 Nota a margine

Differentemente dalla teoria (che associa ad ogni id uno ed un unico behaviour state), all'interno del progetto una variabile può avere "contemporaneamente" più behaviour states: il primo rappresentante l'effetto provocato all'uscita della funzione, mentre quelli a seguire rappresentanti gli stati locali.
Questa scelta implementativa è stata dettata dalla necessità di mantenere memorizzata una variabile anche dopo la sua eliminazione (in modo tale da poter analizzare il behaviour state, al termine dell'invocazione di funzione, di una variabile passata per riferimento calcolando il risultato del *seq* tra l'effetto precedente e quello interno alla funzione) e, contemporaneamente, continuare ad analizzare il comportamento "locale" della variabile.

Listing 8: esempio

```
1   {
2       int x;
3       void f(int var y) {
4           delete y;
5           int y;
6           void g(int var z) {
7               delete z;
8               int z;
9               z = 3;
10          }
11          g(y);
12      }
13      f(x);
14  }
```

Nell'esempio soprastante, il tipo finale di x è "D" (poiché viene cancellato dalla prima istruzione all'interno di "f").
All'interno della funzione, tuttavia, la variabile viene ridichiarata (e pertanto impostata a "B") ma lo stato che questa assume non deve influire sulla variabile passata per riferimento ("x"). Pertanto, i behaviour states di "y" una volta analizzata la riga 5 saranno "D" e "B", dove "D" rappresenta l'effetto "di referenza" (cioè l'effetto che deve essere considerato al momento dell'analisi del comportamento dell'invocazione "f") e "B" rappresenta il behaviour state "aggiornato" (cioè quello di riferimento per il resto dell'analisi).
Analogamente al comportamento di "y", anche "z", una volta analizzata la riga 9, avrà due effetti: "D" ed "RW".
Nella pratica, un nuovo behaviour state viene aggiunto alla lista solo quando la variabile viene dichiarata dopo essere stata eliminata.

2 Esercizio 2

Definire un linguaggio bytecode per eseguire programmi in SimplePlus, scrivere la compilazione e implementare l'interprete. In particolare:

1. il bytecode deve avere istruzioni per una macchina a pila che memorizza in un apposito registro il valore dell'ultima istruzione calcolata [vedi slide delle lezioni];
2. implementare l'interprete per il bytecode;
3. permettere la compilazione e successiva esecuzione di programmi del linguaggio ad alto livello.

2.0 Architettura Interprete

L'interprete possiede 7 registri: $a0, $t1, $fp, $ra, $hp, $sp, $al. I commenti riguardanti il loro utilizzo sono omessi in quanto già approfonditi a lezione.
Oltre ai registri, la VM utilizza una struttura a pila per gestire l'esecuzione del codice: le posizione più basse rappresentano – in senso logico – l'heap, quelle più alte lo stack.

La struttura a pila è stata implementata come un array di byte, in modo da ottimizzare la gestione della memoria in base al tipo di variabile da memorizzare: un "bool" occupa 1 byte mentre un "int" 4.

All'interno dello spazio logico dello stack, vengono memorizzati esclusivamente gli activation record – che contengono i relativi parametri richiesti dalle funzioni – e i valori utili alle operazioni intermedie. Le variabili definite esplicitamente (tramite "StmtDecVar") sono salvate nell'heap.

Il codice non viene memorizzato all'interno della pila ma in una lista dedicata ("code").

2.1 Generazione del bytecode

Ogni elemento dell'AST possiede il metodo "codeGen" responsabile della generazione del bytecode specifico per quel nodo.

Il bytecode iniziale è generato dalla decorator class "CodeMaker" che, prima di richiamare il "codeGen" della root dell'AST, genera il bytecode responsabile della creazione di un primo Activation Record che rappresenta il "main". Questo AR è stato aggiunto al fine di garantire la possibilità di effettuare un "return" dal blocco principale del programma (e non solo dalle funzioni).

Listing 9: "codeGen" di "CodeMaker"

```
1  public String codeGen() {
2     CustomStringBuilder csb = new CustomStringBuilder(new StringBuilder());
3     // Wraps the actual code in a starting AR to avoid errors in case of
          ↪ "return" statement in main block
4     csb.newLine("b CALLMAIN");
5     csb.newLine("MAIN:");
6     csb.newLine("move $fp $sp");
7     csb.newLine("push $ra 4");
8     csb.newLine();
9
10    root.codeGen(0, csb);
11
12    csb.newLine();
13    csb.newLine("lw $ra 0($sp) 4");
14    csb.newLine("addi $sp $sp 8");
15    csb.newLine("lw $fp 0($sp) 4");
16    csb.newLine("pop 4");
17    csb.newLine("jr");
18    csb.newLine("CALLMAIN:");
19    csb.newLine("push $fp 4");
20    csb.newLine("move $al $fp");
21    csb.newLine("push $al 4");
22    csb.newLine("jal MAIN");
23
24    return csb.toString();
25 }
```

Al fine di eseguire il codice utente (generato da "root") dopo l'inizializzazione dell'Activation Record, viene – dapprincipio – eseguito un "jump" alla label "CALLMAIN" la quale predispone l'AR per l'invocazione del codice utente e richiama "MAIN", che completa la costruzione dell'AR e genera il codice di "root".
Le righe 13/17 sono responsabili della chiusura dell'Activation Record una volta terminato il codice utente.

Alcune menzioni interessanti (che si discostano dalla teoria) sono le seguenti:

- I bytecode "PUSH", "POP", "LW" ed "SW" richiedono un ulteriore parametro rappresentante la dimensione del valore da manipolare.
  Questo approccio è stato dettato dall'impossibilità di memorizzare il tipo di un valore (e quindi la sua dimensione) a livello di codice (si sarebbe dovuto tenere traccia di tutte le variabili memorizzate e delle dimensioni dei loro tipi). È importante sottolineare che la dimensione del tipo di un parametro passato per riferimento è sempre 4 byte in quanto, all'interno della cella corrispondente, non viene memorizzato il valore della variabile a cui si fa riferimento (che può essere "bool" o "int") ma il suo indirizzo (che è sempre un intero).

- Il "codeGen" di "StmtDecFun", prima di generare il bytecode del corpo della funzione, esegue il comando "branch" all'"end" della funzione al fine di evitare l'esecuzione del corpo della stessa.

Listing 10: "codeGen" di "StmtDecFun"

```
public void codeGen(int nl, CustomStringBuilder sb) {
    String end = CodeGenUtils.freshLabel();
    sb.newLine("b ", end);
    // ...

    block.codeGen(nl + 1, sb);

    // ...
    sb.newLine(end, ":");
}
```

- Ogni qual volta si risale la catena degli activation record via "access link", il valore iniziale di $al (l'access link) viene copiato da quello di $fp (il frame pointer) anziché, come da teoria, impostato effettuando un "lw" dall'indirizzo contenuto nel frame pointer stesso (che causerebbe il jump all'activation record subito precedente anche a parità di nesting level).

- "print" stampa sempre e solo il valore in $a0.

- "jr" salta direttamente all'indirizzo del codice contenuto nel registro $ra.

- Quando una variabile viene eliminata e poi ridefinita, non si alloca nuovo spazio di memoria: la nuova variabile viene salvata nello stesso indirizzo di quella cancellata.

Listing 11: "codeGen" di "StmtDecVar"

```
public String codeGen(int nl, CustomStringBuilder sb) {
      // ...
   sb.newLine("li $t1 0");
   // always save the value of $a0 in the variable location
   sb.newLine("sw $a0 ", Integer.toString(idEntry.offset), "($t1) ",
      ↪ Integer.toString(type.getDimension()));
   // if the variable was been deleted, set it to "non-deleted",
      ↪ otherwise increase the heap pointer ($hp)
   if (idEntry.isDeleted())
      idEntry.setDeleted(false);
   else
      sb.newLine("addi $hp $hp ", Integer.toString(type.getDimension()));
}
```

Al fine di implementare tale comportamento, è stato sufficiente salvare il valore ritornato da "inferBehaviour" per le variabili passate per riferimento e, nel caso queste risultino cancellate, aggiornare il loro stato nell'"idEntry" corrispondente.

Listing 12: "inferBehaviour" e "codeGen" di "StmtCall"

```
public List<BehaviourError> inferBehaviour(Environment<BTEntry> e) {
    // ...
     expsBehaviour.set(entry.getValue().x,
        ↪ e.getIDEntry(entry.getKey()).getLocalEffect());
    // ...
}

public void codeGen(int nl, CustomStringBuilder sb) {
    // ...
   if (exps.get(i) instanceof ExpVar &&
      ↪ expsBehaviour.get(i).compareTo(EEffect.D) == 0)
     ((ExpVar) exps.get(i)).getIdEntry().setDeleted(true);
   // ...
 }
```

2.2 Implementazione interprete per il bytecode

L'interprete per il bytecode è stato generato con l'ausilio di ANTLR: il file "SVM.g4" contiene la grammatica relativa al bytecode e "VisitorImplSVM.java" popola l'array "code" di valori interpretabili dal "ExecuteVM" che, infine, esegue le dovute operazioni per ogni istruzione.

"ExecuteVM" possiede unicamente il metodo "cpu", responsabile dell'esecuzione dell'intero codice.
Risultano degni di nota i seguenti"cases":

- "PUSH": al fine di caricare il valore richiesto nello stack, viene utilizzata la funzione "decToByte" che trasforma il valore passatogli in una sequenza di byte e ritorna il byte all'indice richiesto. Nello specifico, la sequenza calcolata avrà il byte meno significativo alla posizione 0 e quello più significativo all'ultima. Questa funzione è utilizzata per salvare nello stack il valore richiesto (con la relativa dimensione): il byte più significativo del valore viene memorizzato all'indirizzo più basso, quello meno significativo in quello più alto.

Listing 13: case "PUSH"

```
case SVMParser.PUSH:
    regs[Regs.$sp.ordinal()] = regs[Regs.$sp.ordinal()] - code[ip + 1];

    for (v1 = 0; v1 < code[ip + 1]; v1++)
        memory[regs[Regs.$sp.ordinal()] + v1] = decToByte(regs[code[ip]],
            ↪ (byte) (code[ip + 1] - v1 - 1));
    ip += 2;
    break;
```

- "LOADWORD": al fine di caricare all'interno del registro specificato il valore dello stack all'indirizzo richiesto, viene utilizzata la funzione "byteToDec" che trasforma una sequenza di 4 byte in un intero. Il primo parametro passato rappresenta il byte più significativo del numero da comporre, l'ultimo, invece, quello meno significativo.

Listing 14: case "LOADWORD"

```
case SVMParser.LOADWORD:
    if (code[ip + 3] == 1) // the value to load is boolean (1 byte)
        regs[code[ip]] = byteToDec((byte) 0, (byte) 0, (byte) 0,
            ↪ memory[code[ip + 1] + regs[code[ip + 2]]]);
    else // The value is an integer
        regs[code[ip]] = byteToDec(memory[code[ip + 1] + regs[code[ip +
            ↪ 2]]], memory[code[ip + 1] + regs[code[ip + 2]] + 1],
            ↪ memory[code[ip + 1] + regs[code[ip + 2]] + 2],
            ↪ memory[code[ip + 1] + regs[code[ip + 2]] + 3]);
    ip += 4;
    break;
```

NOTA: è importante sottolineare che "POP" non memorizza alcun valore nei registri ma incrementa semplicemente il valore dello stack pointer della dimensione passata.

2.3 Compilazione ed esecuzione

La classe "Main" gestisce le procedure di compilazione e successiva esecuzione del codice utente. I vari step sono separati da variabili di tipo "Step" (che identificano un singolo passo da compiere come l'analisi lessicale, quella semantica di preparazione, degli effetti, ecc...).
L'ordine degli step è impostato dal costruttore della classe "Main".

Listing 15: costruttore e "main" in "Main"

```
private Main(Logger logger) {
    this.logger = logger;
    this.steps.add(checkLexicalStep);
    this.steps.add(checkSemanticStep);
    this.steps.add(checkTypesStep);
    this.steps.add(analyseBehaviourStep);
    this.steps.add(generateCodeStep);
    this.steps.add(runCodeStep);
}

public static void main(String[] args) throws IOException {
    new Main(LoggerFactory.getLogger()).start(args);
}
```

Il metodo "manipulateArgs" (richiamato all'inizio del programma) controlla il numero di parametri passati: se si esporta il progetto come "Runnable JAR" (utilizzando come main class "Main.java") è possibile avviare l'applicativo da terminale mediante la dicitura

`java -jar ./nomeJar.jar input_file`.
Se non vengono passati argomenti, sarà utilizzato il parametro di default ("test.spl").

Il file di output contenente il bytecode possiede lo stesso nome del file in input ma con estensione ".out" anziché ".spl".

Ad ogni step viene controllato che non siano stati riscontrati errori. Se questo avviene, lo "step" corrispondente ritornerà "false" ed il programma terminerà.

3 Integrazione in Eclipse ed Esportazione in JAR

Per importare in Eclipse il progetto, selezionare "File" → "Import".
Dalla finestra di dialogo, selezionare "General" → "Project from Folder or Archive" → "Archive" e selezionare il file zippato.
Spuntare esclusivamente la folder "Giacche'_Boccuto_Celozzi.zip_expanded/ProgettoCompilatori2020" e cliccare "Finish".

Il progetto è già provvisto di antlr-4.6 e JUnit5 e tutte le classi base ANTLR sono già state generate. Qualora si volessero rigenerare i file ".g4", si segua il percorso "src/parser", si clicchi col tasto destro su "SimplePlus.g4" e si selezioni "Run As" → "External Tool Configurations". Dalla finestra di dialogo che compare, cliccare due volte su "ANTLR" e, all'interno del field "Arguments", incollare la seguente stringa:

```
-no-listener -visitor -encoding UTF-8 -o src/parser -package parser
```

Se lo si desidera, si può cambiare il nome della configurazione modificando il campo "Name", dopodiché si selezioni "Apply" e poi "Run".
Ripetere poi l'operazione con il file "SVM.g4".

Il progetto è provvisto di due test suite all'interno del pacchetto "tests": "CorrectCodes.java" – contenente un insieme di codici eseguibili più o meno complessi – e "WrongCodes.java": un insieme di test di codici non corretti con relative spiegazioni.
Per eseguire i tests, cliccare con il tasto destro sull'omonimo pacchetto e selezionare "Run As" → "JUnit Test".

Per eseguire il programma principale, cliccare col tasto destro su "Main.java" (all'interno dell'omonimo pacchetto) e selezionare "Run As" → "Java Application". Il programma effettuerà il controllo, la compilazione e l'esecuzione del codice all'interno del file "test.spl" (localizzato nella cartella radice del progetto).

Per utilizzare l'analizzatore in maniera esterna al progetto, selezionare "File" → "Export". Dalla finestra di dialogo aperta, selezionare "Java" → "Runnable JAR file", cliccare su "Next", impostare "Main - ProgettoCompilatori2020" nel field "Launch configuration", selezionare una destinazione in "Export destination" impostando come nome "analyzer.jar" e cliccare su "Finish". Se compaiono warnings, premere "OK".

Per eseguire il jar generato, aprire un terminale, spostarsi sulla cartella dove è contenuto il file jar ed eseguire il comando:

```
java -jar analyzer.jar program.spl
```

Dove "program.spl" è il file di input da analizzare.