Compilatori e Interpreti
Documentazione progetto "SimplanPlus"

Valentina Ferraioli, Francesco Biancucci

Settembre 2021

1 Introduzione

Questo progetto è un esercizio puramente accademico sull'implementazione di un compilatore e un interprete per un linguaggio di programmazione semplificato, *SimpLanPlus*. Il lavoro è stato svolto prendendo spunto da una base già funzionante e fornita dal professore, ovvero *SimpLan*, che è stata poi estesa con le caratteristiche richieste. In particolare, veniva richiesto di implementare un compilatore che accettasse un linguaggio con le seguenti caratteristiche:

- Dichiarazioni di variabili (booleane o intere) ;
- Dichiarazione e definizione di funzioni
- Utilizzo dei puntatori e gestione manuale della memoria

Il compilatore di **SimpLanPlus** svolge i seguenti step:

1. *analisi lessicale*, in cui il codice sorgente viene trasformato in *token*, ovvero piccoli blocchi logici, che contribuiscono alla creazione dell'**Abstract Syntax Tree (AST)**;
2. *analisi semantica*, in cui viene verificata la correttezza semantica del programma in input (ad esempio, che le variabili vengano utilizzate solo dopo una loro dichiarazione);
3. *type checking*, in cui viene controllato che i tipi all'interno delle espressioni, nelle assegnazioni e nelle istruzioni siano coerenti con quelli attesi;
4. *analisi degli effetti*, in cui viene verificato lo stato delle variabili e dei puntatori nella memoria (inizializzato $\perp$, letto/scritto rw, cancellato d, errore $\top$).

Dopodichè si passa alla generazione del codice intermedio che verrà passato come input all'interprete che procederà con l'esecuzione delle istruzioni.

Il progetto è stato scritto in `Java` ed è stato utilizzato `Antlr 4.7.2`

2 Sintassi

In seguito procederemo con l'analisi della sintassi fornita:

- Un programma *SimpLanPlus* inizia con un **block**, il quale è composto da una sequenza di dichiarazioni a cui segue una sequenza di statement. Facciamo notare che entrambe le sequenze potrebbero essere vuote.

```
grammar SimpLanPlus;

block: '{' declaration* statement* '}';
```

- Nel caso delle dichiarazioni le possibili istruzioni sono due:
  - dichiarazione di variabile di tipo intero, booleano o puntatore. Durante la dichiarazione si potrebbe procedere anche con l'assegnazione di una **exp**, che vedremo in seguito approfonditamente, alla variabile appena dichiarata.
  - dichiarazione di funzione che possono essere solo di tipo intero, booleano o void.
    Facciamo notare che durante l'implementazione del progetto è stato assunto che le funzioni non possono ritornare puntatori.

```
declaration : decFun | decVar ;

decVar      : type ID ('=' exp)? ';';
type        : 'int' | 'bool' | '^' type;

funType     : type | 'void';
arg         : type ID;
```

- Nel caso degli statement invece, le operazioni possibili sono molteplici, come ad esempio :
  - assegnamento ad una variabile di un **exp**;
  - cancellazione di un puntatore, ovvero il puntatore non fa più riferimento ad alcuna cella di memoria, e la cella a cui puntava precedentemente può essere liberata dal Garbage Collector se sono soddisfatti i requisiti necessari.
  - costrutto if then else
  - dichiarazione di un nuovo blocco annidato
  - chiamata di funzione
  - ritorno di un **exp**
  - stampa a video del contenuto di un **exp**

```
assignment  : lhs '=' exp ;
lhs         : ID | lhs '^' ;
deletion    : 'delete' ID;
print       : 'print' exp;
ret     : 'return' (exp)?;
ite         : 'if' '(' exp ')' statement ('else'
    statement)?;
call        : ID '(' (exp(',' exp)*)? ')';
```

- Infine, in diverse delle regole viste finora è stato fatto riferimento alle **exp** , ad esempio nell'assegnamento, o nel ritorno e stampa di un valore.
  Le **exp** possono essere le seguenti:

```
exp: '(' exp ')'        #baseExp
   | '-' exp            #negExp
   | '!' exp            #notExp
   | lhs                #derExp
   | 'new' type         #newExp
   | left=exp op=('*' | '/') right=exp          #
     binExp
   | left=exp op=('+' | '-') right=exp          #
     binExp
   | left=exp op=('<' | '<=' | '>' | '>=') right=exp
      #binExp
   | left=exp op=('=='| '!=') right=exp         #
     binExp
   | left=exp op='&&' right=exp                 #
     binExp
   | left=exp op='||' right=exp                 #
     binExp
   | call             #callExp
   | BOOL             #boolExp
   | NUMBER;          #valExp;

BOOL        : 'true'|'false';
// IDs
fragment CHAR   : 'a'..'z'|'A'..'Z';
ID              : CHAR (CHAR | DIGIT)*;
// Numbers
fragment DIGIT  : '0'..'9';
NUMBER          : DIGIT+;
//blank space and errors
WS              : (' '|'\t'|'\n'|'\r') -> skip;
LINECOMMENTS    : '//' (~('\n'|'\r'))* -> skip;
BLOCKCOMMENTS   : '/*'( ~('/'|'*')|'/'~'*'|'*'~'/'
    |BLOCKCOMMENTS)* '*/' -> skip;
ERR     : . { errors.add("Invalid character: "+ getText
    ()); } -> channel(HIDDEN);
```

Dalla grammatica appena descritta sono stati generati automaticamente, con il tool fornito da *Antlr*, il lexer e il parser che si occupano di eseguire l'analisi sintattica. Al termine dell'analisi sintattica, se non sono stati rilevati errori, si procede con l'esecuzione dell'analisi semantica.

3 Analisi Semantica

La fase di analisi semantica ha lo scopo di individuare gli errori non rilevati durante la fase di analisi sintattica ed è suddivisa in tre sotto fasi, che consistono in:

- analisi degli scope
- analisi dei tipi
- analisi degli effetti

Per completezza, prima di vedere come è stata implementata l'analisi semantica, verrà fornita una panoramica sulla creazione dell'environment, il quale ha il compito principale di creare e gestire la Symbol Table. Nel corso delle tre fasi, verranno utilizzati Environment diversi, Γ e Σ, ma che condividono la struttura presentata in seguito. Invece le peculiarità di Γ e Σ saranno trattate in 3.1 e 3.3.

Environment e STentry

Il design della tabella dei simboli è stato realizzato in accordo con le peculiarità di SimpLanPlus. In primo luogo, il linguaggio utilizza scoping e sistema di tipaggio statici, e per questo motivo è necessario dichiarare ogni variabile prima del suo utilizzo. Inoltre, è stata negata la possibilità di utilizzare uno stesso identificatore per la dichiarazione di più variabili all'interno dello stesso scope; tuttavia, è possibile utilizzare uno stesso identificatore all'interno di scope annidati. Infine, è stato utilizzato lo stesso scope per i parametri di funzioni e per le variabili dichiarate all'inizio di tali funzioni.
Il design risultante per la tabella dei simboli è stato quello di una lista di tabelle hash.
Le *Entries* della tabella hash sono create dalla classe *STentry* e affinchè non appesantissero la struttura della tabella è stato scelto di mantenerle inizialmente il più semplice possibile, per questo motivo sono composte solo da 3 campi:

- *TypeNode type* , il nodo tipo relativo al simbolo a cui l'entry fa riferimento;
- *int nl*, il suo nesting level;
- *i*il suo offset all'interno del nesting level.

La *SymbolTable* invece, viene creata e gestita attraverso la classe *Environment* che fornisce i seguenti metodi:

- **Environment** associa la stringa di un identificatore con la corrispondente instanza di *STentry*
- **void onScopeEntry()** che consiste nell'incremento del nesting level e corrisponde all'operazione $\Gamma \cdot []$
- **STEntry addDec(String id, TypeNode type)** che corrisponde a $\Gamma[id \rightarrow \text{type}]$
- **STEntry lookup(String id)** che corrisponde all'operazione $\Gamma(id)$
- **void popScope()** che corrisponde all'operazione $\Gamma \leftarrow \Gamma'$ partendo da $\Gamma = \Gamma' \cdot top(\Gamma)$.

3.1 Analisi degli scope

In questa prima fase viene creato l'environment Γ e viene attraversato l'albero di sintassi astratta, costruito dal parser, durante il quale viene eseguito il metodo **ArrayList<SemanticError> checkSemantics(Environment Gamma)** che si occupa di controllare la sua correttezza e di invocare il metodo *checksemantic()* sui suoi figli, controllando conseguentemente anche la loro correttezza. Quindi per ogni scope all'interno del programma si devono:

- processare le dichiarazioni : per ogni nuova variabile dichiarata deve essere aggiunta una *entry* corrispondente nella tabella dei simboli. Questa fase si occupa anche del **rilevamento delle dichiarazioni multiple** all'interno di uno stesso scope del programma. Per fare ciò, in *checkSemantic()* viene invocato il metodo *addDec* della classe *Environment* passando come argomento l'identificativo della variabile (o funzione) da dichiarare e il tipo associato.
  INel caso in cui lo stesso identificativo sia già presente nello *scope* corrente, viene ritornato un errore semantico.
  Di seguito la porzione di codice che si occupa di quanto descritto:

```
public ArrayList<SemanticError> checkSemantics(
    Environment env) {

   ArrayList<SemanticError> errors = new ArrayList<>();
   STentry addedEntry;
   if(exp != null) {
       errors.addAll(exp.checkSemantics(env));
   }
   try {
       addedEntry = env.addDec(id.getTextId(), type);
       id.setSTentry(addedEntry);
   } catch (AlreadyDeclaredException exception) {
       errors.add(new SemanticError(exception.getMessage
           ()));
   }
}
```

- processare le istruzioni : ogni istruzione del programma viene analizzata per **rilevare eventuali utilizzi di identificatori di variabili non dichiarati** e per aggiornare e collegare correttamente i nodi dell'albero di sintassi astratta alle relative entry della tabella dei simboli. Per fare ciò, viene invocata il metodo *lookup* della classe *Environment* passando come argomento l'identificativo della variabile (o funzione) da utilizzare. In caso non venga trovato viene ritornato un errore semantico.
  Di seguito la porzione di codice che si occupa di quanto descritto:

```
@Override
public ArrayList<SemanticError> checkSemantics(Environment
     env) {
  ArrayList<SemanticError> res = new ArrayList<
      SemanticError>();
    try {
        this.entry = env.lookup(id);    //throw
            NotDeclaredException if lookup return null
        nl = env.getNestingLevel();
    }catch (NotDeclaredException e){
        res.add(new SemanticError(e.getMessage()));
    }
    return res;
}
```

  Nel caso in cui l'istruzione in analisi sia una chiamata di funzione, oltre che controllare che la funzione sia presente all'interno della symbolTable è richiesto anche che venga fatto un primo controllo sulla conformità dei parametri attuali rispetto a quelli formali, che consiste nel controllare che il numero di parametri passati sia coerente con la definizione di funzione. Il controllo sulla conformità dei parametri sarà poi concluso durante l'analisi dei tipi.

```
@Override
public ArrayList<SemanticError> checkSemantics(Environment
     env) {
    ArrayList<SemanticError> errors = new ArrayList<>();
    //check if the id exists, if not we're using an
        undeclared identifier
    errors.addAll(id.checkSemantics(env));
    if(!errors.isEmpty())
        return errors;
    //check all the parameters, if there's something wrong
         with them during the invocation such as int +
        bool
    for(ExpNode p : params)
        errors.addAll(p.checkSemantics(env));
    //check if the number of actual parameters is equal to
         the number of formal parameters
    currentNl = env.getNestingLevel();
    int nFormalParams = 0;
    int nActualParams = params.size();

    if(id.getSTentry().getType() instanceof FunTypeNode)
        nFormalParams =  ((FunTypeNode) id.getSTentry().
            getType()).getParamsType().size();
```

```
        if(nActualParams != nFormalParams)
            errors.add(new SemanticError("There's a difference
                 in the number of actual parameters versus the
                 number of formal parameters declared in the
                function " + id.getTextId()));
        //idk if i'm missing something but this seems fine to
            me
        return errors;
    }
```

Gli errori, rappresentati con un istanza di *SemanticError*, saranno raccolti e ritornati al nodo padre al termine dell'esecuzione del metodo *CheckSemantics()*. Al termine della visita si controlla che il nodo radice non abbia ritornato alcun errore e se vero, si procede con l'analisi dei tipi. Altrimenti l'esecuzione viene bloccata.

3.2 Analisi dei tipi

In questa fase viene attraversato l'AST nuovamente e verranno processati tutti gli statement del programma usando le informazioni raccolte in Γ nella fase precedente, in modo da definire il tipo di ogni espressione e di rilevare eventuali errori di tipo. Prima di tutto, vediamo i tipi implementati in questo linguaggio:

- *bool*: rappresenta un tipo booleano;
- *int*: rappresenta un tipo intero;
- *pointer*: rappresenta un tipo puntatore che punta a un tipo booleano, intero o a un altro tipo puntatore;
- *void*: rappresenta il tipo vuoto;
- *function*: tipo utilizzato per le funzioni, contiene la lista dei tipi dei parametri e il tipo ritornato.

L' analisi dei tipi viene gestita tramite il metodo *TypeNode typeCheck()* che restituisce:

- un istanza di *TypeNode*, ovvero il tipo associato al nodo su cui è stato invocato;
- null se è invocato su un nodo che ha come supertipo *TypeNode*, oppure è un *ArgNode*, o rappresenta una dichiarazione di variabili o di funzioni.

Inoltre, il metodo solleva eccezioni di tipo *TypeException*, nel caso in cui vengano rilevati errori di tipo durante l'esecuzione.
Gli statement, per scelta implementativa, ritornano il tipo *void* ma con alcune eccezioni, infatti:

- **RetNode** ritorna il tipo dell'espressione, o *void* nel caso non ce ne sia alcuna;
- **CallNode** ritorna lo stesso tipo ritornato dalla funzione chiamata;
- **IteNode** che ritorna il tipo comune ai due rami e solleva un' eccezione nel caso i rami ritornino tipi differenti;
- **BlockNode** ritorna *void* in assenza di istruzioni o quando non vi è né *return* né *if-then-else* altrimenti ritorna il tipo di **RetNode** o di **IteNode** .

3.3 Analisi degli effetti

Arrivati a questo punto possiamo assumere che i programmi non presentino errori semantici e che siano correttamente tipati e quindi si può procedere con l'analisi degli effetti.
In questa fase si vuole essere sicuri che non siano fatti degli accessi ad aree di memoria non allocate in modo corretto.
Per cui ad ogni variabile viene associato uno tra i seguenti effetti di $\beta = \{\bot, rw, d, \top\}$, dove :

- $\bot$, ovvero la variabile è stata dichiarata, ma non allocata.
- *rw*, ovvero la variabile è accedibile in lettura o scrittura.
- *d*, assegnabile solo ad una variabile di tipo puntatore quando si vuole liberare lo spazio di memoria puntato.
- $\top$, assegnata ad una variabile il cui stato è inconsistente.

Per gestire nel modo più opportuno gli effetti è stata implementata la classe *Effect* in modo tale che potesse garantire l'ordinamento degli effetti $\bot < rw < d < \top$ e che permettesse l'esecuzione delle operazioni **max, par, seq**. Infatti, la classe è composta da:

- un campo statico di tipo intero per ognuno degli effetti a cui è associato un incrementale rispetto all'ordinamento
- campo *type* di tipo intero, rappresentante l'effetto corrente.
- metodi
  - *Effect max(Effect e1, Effect e2)* che ritorna il massimo fra *e1* e *e2* (vengono confrontati i campi *value*)
  - *Effect seq(Effect e1, Effect e2)* che ritorna *max(e1, e2)* se $\leq rw$, *d* se *e1* $\leq rw$ e *e2*— $= d$ oppure *e1* $= d$ e *e2* $= \bot$, $\top$ in tutti gli altri casi;
  - *Effect par(Effect e1, Effect e2)* che ritorna il massimo fra gli effetti ritornati dalle invocazioni *seq(e1, e2)* e *seq(e2, e1)*.

Oltre alla classe *Effect*, sono state estese le classi *STEntry* e *Environment*.

- In *STentry*, nello specifico sono stati aggiunti i seguenti campi per la gestione degli effetti:
  - *HashMap<String, Effect> varStatus*
  - *List<HashMap<String,Effect>> funStatus*

  e metodi:
  - *initializeStatus*

```
public void initializeStatus(IdNode id){

if ( (this.type instanceof FunTypeNode)) {
    HashMap<String, Effect> init_env_0 = new
        HashMap<>();
    HashMap<String, Effect> init_env_1 = new
        HashMap<>();

    for (ArgNode param : this.getFunNode().getArgs
        ()) {
        init_env_0.put(param.getId().getTextId(),
            new Effect(Effect.BOT));
        init_env_1.put(param.getId().getTextId(),
            new Effect(Effect.BOT));
        }
    this.funStatus.add(init_env_0);
        //Sigma_0
    this.funStatus.add(init_env_1);
        //Sigma_1
//at this point funstatus is Sigma_0 -> Sigma_1
    where  Sigma_0 =Sigma_1 = [x1 -> bot ..xn ->
    bot]
}
else{
    this.varStatus.put(id.getTextId(), new Effect(
        Effect.BOT));
}
```

  - *updateArgStatus*
  - *AddEntry()*
  - *getter/setter* vari

- In Environment sono stati aggiunti i seguenti metodi principali per la gestione dell'analisi degli effetti:
  - *Environment max(Environment env1, Environment env2)* che per ogni scope, a partire da quello di livello 0, per ogni variabile presente sia in *env1* che in *env2* applica l'operazione *max* fra gli stati mentre per quelle presenti solo in *env1*, le ritorna così come sono;
  - *Environment seq(Environment env1, Environment env2)* si comporta come *max*, applicando però l'operatore *seq* fra gli stati delle variabili;

- *Environment par(Environment env1, Environment env2)* assume che *env1* ed *env2* siano ambienti con un solo scope (perché solo in tale contesto viene invocata questa funzione), per ogni variabile presente in entrambi gli ambienti viene fatto il *par* fra gli stati, altrimenti le variabili vengono semplicemente aggiunte all'ambiente risultante.
- *Environment update(Environment env1, Environment env2)* assume che *env1* e *env2* abbiano rispettivamente un livello di annidamento ≥ 1 e $= 1$. Se *env2* è vuoto oppure non ha variabili definite viene ritornato *env1* (caso base) altrimenti, $\forall u \in$ *env2* prima si rimuove u da *env2*, successivamente se $u \in top(env1)$ allora $env1[u \rightarrow u.type]$ e viene ritornato *update(env1, env2)* (è una chiamata ricorsiva, con *env1* e *env2* che sono stati aggiornati), altrimenti effettua un serie di chiamate ricorsive riassumibili nella formula *update(update(env,* $[u \rightarrow u.type]$*)·* $top(env1)$*, env2)* con $env1 \leftarrow env \cdot top(env1)$

A questo punto abbiamo visto tutti gli elementi che verranno utilizzati durante l'analisi degli effetti vera e propria.
In questa fase prima di tutto, viene creato il nuovo environment Σ e viene attraversato l'albero di sintassi astratta, costruito dal parser, durante il quale viene eseguito il metodo *ArrayList<SemanticError> checkEffects(Environment Sigma)* con lo stesso meccanismo visto per il *checkSemantics*.
Facciamo notare, che la creazione di Σ differisce da quella di Γ in quanto in questo caso abbiamo già a disposizione le *entries* associate ai nodi dell'AST, percui è stato sufficiente aggiungerle alla symbolTable tramite la funzione *Environment.AddEntry()* Per L'implementazione dei metodi *checkEffects(Environment Sigma)* è stato fatto riferimento alle regole di inferenza studiate a lezione .
In particolare, di seguito, vengono descritti più in dettaglio alcuni dei controlli ritenuti di maggiore importanza.

- **variabili non inizializzate** e **utilizzo di variabili cancellate**
  Per impedire l'utilizzo di variabili non inizializzate o cancellate precedentemente è stato implementato un controllo nell'espressioni che accedono alle variabili.
  Di seguito viene presentato la porzione di codice che se ne occupa:

$$\frac{\Sigma(x) \neq \bot}{\Sigma \vdash x : \bot}[Id] \qquad \frac{\Sigma(x) \neq d}{\Sigma \vdash x : d}[Del]$$

  In *LhsNode*

```
@Override
public ArrayList<SemanticError> checkEffects(Environment
    env)  {
    ArrayList<SemanticError> res = new ArrayList<>();

    if (lhs == null) {      //we are processing a
        dereference node
        res.addAll(id.checkEffects(env));
```

```
            if(id.getSTentry().getIVarStatus(id.getTextId()).
                getType() == Effect.BOT)
                res.add(new SemanticError(id.getTextId() + "
                    is used before being initialized"));
            if(id.getSTentry().getIVarStatus(id.getTextId()).
                getType() == Effect.DEL)
                res.add(new SemanticError(id.getTextId() + "
                    is used after being deleted."));

            return res;
        }

        res.addAll(lhs.checkEffects(env));

        return res;

    }
```

- **corretto uso dei puntatori**
  I controlli sui puntatori sono presenti in diversi parti dell'AST:

  – quando viene utilizzato come un'espressione o durante un'assegnamento si controlla che sia il puntatore sia stato inizializzato.

  – al momento della dichiarazione di una funzione visto che non è possibile sapere quali saranno gli stati dei paramatri al momento dell'invocazione si assume che saranno tutti a rw, su questo stato verranno calcolati gli effetti dei parametri al termine della funzione.
  Per poter capire però se i parametri attuali al momento della chiamata siano effettivamente compatibili con quanto accade nelle diverse istruzioni del corpo della funzione, è stato necessario controllare quali operazioni vengono fatte sui parametri attuali, istanziando gli stati con quelli che avranno al momento della chiamata.

  Inoltre se i puntatori vengono inizializzati correttamente il valore puntato viene inizializzato in automatico a -1, è compito poi del programmatore settare correttamente il valore puntato.

- **Dichiarazione e chiamata di funzioni con punto fisso**
  Come è stato detto precedentemente, **SimpLanPlus** è un linguaggio ricorsivo, per cui in caso di funzioni ricorsive è necessario che gli effetti associati alle variabili al termine dell'esecuzione siano corretti. Perchè questo sia possibile gli effetti vengono calcolati utilizzando il metodo del punto fisso. Per l'implementazione del metodo è stato fatto riferimento alla seguente regola d'inferenza:

$$\Sigma_0 = [x_1 \to \bot, \ldots, x_m \to \bot, y_1 \to \bot, ..]$$

$$\Sigma|_{FUN} \cdot \Sigma_0[f \to \Sigma_0 \to \Sigma_1] \vdash s : \Sigma|_{FUN} \cdot \Sigma_1[f1 \to \Sigma_0 \to \Sigma_1]$$

$$----------Fseq-e]$$

$$\Sigma \vdash f(varT_1x_1, \ldots, varT_mx_m, T_1'y_1, \ldots, T_n'y_n)s : \Sigma[f \to \Sigma_0 \to \Sigma_1]$$

In seguito il metodo implementato:

```
@Override
public ArrayList<SemanticError> checkEffects(Environment
    env) {
    ArrayList<SemanticError> errors = new ArrayList<>();
    //setting up the effects
    id.getSTentry().setFunNode(this);
    id.getSTentry().initializeStatus(id);       //
        inizializing [f-> Σ0-> Σ1] environment, arguments
        effects are initialized to BOT
    env.addEntry(id.getTextId(), id.getSTentry());  //
        adding function declaration to Σ0[f-> Σ0-> Σ1]

    env.onScopeEntry();

    STentry argEntry;

    for (ArgNode arg: args){
    //adding arguments entry inside Σ0 with effect setted
        as RW, in order to be usable inside function body
    //a Σ0[f-> Σ0-> Σ1,x1-> RW,..,xn-> RW]

        arg.getId().getSTentry().setVarStatus(arg.getId().
            getTextId(),new Effect(Effect.RW));
        env.addEntry(arg.getId().getTextId(), arg.getId().
            getSTentry());
    }

    // end first cicle Σ|_FUN · Σ0[f → Sigma0 → Σ1]

    Environment env1 = new Environment(env);
    Environment oldEnv = new Environment(env);
    errors.addAll(body.checkEffects(env1));
                //calculating variables effect after the
        body execution and updating Σ0
    //env1.printEnv();

    STentry argStatusInBodyEnv ;
    //updating fun
    for (ArgNode arg: args) {
        //getting the variable effects from env1 and
            updating Σ1 inside Σ0[f → Σ0 → Σ1]
        argStatusInBodyEnv = env1.lookupForEffectAnalysis(
            arg.getId().getTextId());
        env1.lookupForEffectAnalysis(id.getTextId()).
            updateArgsStatus(arg.getId().getTextId(),
            argStatusInBodyEnv.getIVarStatus(arg.getId().
            getTextId()));
        //re-setting args effect to RW for the next
            evaluation of the body effect.
        env1.lookupForEffectAnalysis(arg.getId().getTextId
            ()).setVarStatus(arg.getId().getTextId(), new
            Effect(Effect.RW));
    }
```

```
    //calculating fixed point we're going to update the
        enviroment Σ0, until
        Σ0[f → Σ0 → Σ1] = FUN · Σ1[f → Σ0 → Σ1]
    while(!oldEnv.equals(env1)) {
        oldEnv=env1;
        errors.addAll(body.checkEffects(env1));
        //updating arg
        for (ArgNode arg: args) {
            argStatusInBodyEnv = env1.
                lookupForEffectAnalysis(arg.getId().
                getTextId());
            env1.lookupForEffectAnalysis(id.getTextId()).
                updateArgsStatus(arg.getId().getTextId(),
                argStatusInBodyEnv.getIVarStatus(arg.getId
                ().getTextId()));
        }
    }

    env.replace(env1);
    env.onScopeExit();

    return errors;
}
```

Per quanto riguarda la chiamata di funzione è stato fatto riferimento alla seguente regola di inferenza:

$$\Gamma \vdash f : \&t_1x..x\&t_mxt'_1x..xt'_n \to void$$

$$\Sigma(f) = \Sigma_0 \to \Sigma_1 \ (\Sigma_1(yi) \leq d)_{1 \leq il \leq n}$$

$$\Sigma' = \Sigma[(z_i \to \Sigma(z_i)seqrw)$$

$$\Sigma'' = par[u_i \to \Sigma(u_i)seq\Sigma 1(x_i)]_{1 \leq i \leq m}$$

$$- -$$

$$\Sigma \vdash f(u_1, .., u_m, e_1, .., e_n) : update(\Sigma', \Sigma'')$$

In seguito, vedremo i punti chiavi implementati dal metodo *checkEffect* di *CallNode*

1. vengono recuperate tutte le espressioni (incluse anche variabili) non puntatori e ci si assicura che il corrispettivo parametro formale non sia in stato $\top$;
2. viene creata una copia dell'ambiente, chiamata *SigmaPrimo*, in cui viene aggiornato l'effetto a *rw* di tutte le variabili facenti parte delle precedenti trovate espressioni;
3. viene creata un'ulteriore copia dell'ambiente, *SigmaSecondo*;
4. per ogni puntatore passato come argomento alla funzione viene creato un nuovo ambiente temporaneo in cui salvare solo una *entry*, relativa a quel puntatore, il cui effetto è il *par* fra l'effetto trovato nell'ambiente di partenza e quello trovato nel corrispondente parametro attuale della funzione da chiamare;

5. tutti questi ambienti temporanei generati al punto precedente vengono messi in *par* fra loro (a cascata) e il risultato viene memorizzato nel precedentemente creato ambiente *SigmaSecondo* (nel caso non ci fossero ambienti temporanei creati, $e2 = \emptyset$);
6. infine, viene invocato *update* con parametri *SigmaPrimo* e *SigmaSecondo* e il risultato è il nuovo ambiente per le successive istruzioni.

4 Generazione del codice intermedio

4.1 Codice intermedio

Una volta appurata la correttezza del programma viene generato il codice intermedio, che poi viene passato all'interprete che si occupa della sua esecuzione. La grammatica del codice intermedio.
Le istruzioni dell'interprete sono definite a partire da quelle presenti nelle slide e sono le seguenti:

```
instruction:
    'push' REGISTER
    | 'pop'
    | 'lw' out = REGISTER offset = NUMBER'('in = REGISTER')'
    | 'sw' in = REGISTER offset = NUMBER'('out = REGISTER')'
    | 'li' out = REGISTER in = NUMBER
    | 'mv' out = REGISTER in = REGISTER
    | 'add' out = REGISTER in = REGISTER in2 = REGISTER
    | 'sub' out = REGISTER in = REGISTER in2 = REGISTER
    | 'mul' out = REGISTER in = REGISTER in2 = REGISTER
    | 'div' out = REGISTER in = REGISTER in2 = REGISTER
    | 'addi' out = REGISTER in = REGISTER in2 = NUMBER
    | 'subi' out = REGISTER in = REGISTER in2 = NUMBER
    | 'muli' out = REGISTER in = REGISTER in2 = NUMBER
    | 'divi' out = REGISTER in = REGISTER in2 = NUMBER
    | 'and' out = REGISTER in = REGISTER in2 = REGISTER
    | 'or' out = REGISTER in = REGISTER in2 = REGISTER
    | 'not' out = REGISTER in = REGISTER
    | 'beq' in = REGISTER in2 = REGISTER LABEL
    | 'bleq' in = REGISTER in2 = REGISTER LABEL
    | 'b' LABEL
    | LABEL ':'
    | 'jal' LABEL
    | 'jr' REGISTER
    | 'del' REGISTER
    | 'print' REGISTER
    | 'halt'
    ;
```

Le istruzioni hanno tutte una forma del tipo:

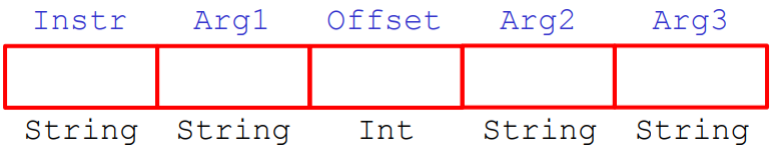

Figure 1: Formato di un'istruzione

Come si può notare dalla Figura 1 l'istruzione è composta da 5 campi, di cui solo il primo è obbligatorio, in quanto è il nome dell'istruzione. L'implementazione delle istruzioni avviene tramite una classe ad hoc e i campi opzionali hanno i seguenti ruoli:

- arg1: primo argomento, può essere un registro o un' etichetta
- offset: numero da sommare a ciò che si trova in arg2 nel caso di alcune istruzioni
- arg2: secondo argomento, nelle operazioni binarie può essere un registro o un intero
- arg3: terzo argomento, specifica un numero, un registro o un'etichetta

Come si può notare molte delle istruzioni lavorano su dei registri, all'interno dei quali possono essere salvate informazioni temporaneamente per permettere la corretta esecuzione del codice. I registri sono definiti sempre attraverso la grammatica e sono i seguenti:

- $a0, registro all'interno del quale si trova il risultato al termine di un operazione
- $t1, registro temporaneo, utilizzato come appoggio
- $sp, punta alla cima dello stack
- $al, registro che permette l'attraversamento della catena statica degli scope
- $fp, punta all'access link del record di attivazione corrente
- $ra, punta all'indirizzo in cui salvare l'istruzione da eseguire una volta usciti dal blocco
- $hp, punta alla prima cella libera dello heap
- $cl, punta al vecchio valore dello stack pointer

4.2 Interprete

Una volta generato il codice intermedio è compito dell'interprete eseguirlo, e ottenere un risultato dal codice.
L'interprete si occupa quindi di leggere il codice intermedio e, come prima cosa, sostituire alle etichette presenti nel codice il numero di istruzione a cui corrispondono. Questo è possibile tramite l'utilizzo di due hashmap

```
private HashMap<String,Integer> labelAdd = new HashMap<String,
    Integer>();
private HashMap<Integer,String> labelRef = new HashMap<Integer,
    String>();
```

La prima delle quali lega ogni etichetta alla riga di codice a cui fa riferimento, mentre la seconda lega l'indirizzo dell'istruzione con l'etichetta che la punta. Tramite questo meccanismo di doppia legatura è poi possibile sostituire, nelle istruzioni che prevedono un salto, il nome dell'etichetta con l'indirizzo effettivo dell'istruzione.

```
for (Integer labelInt: labelRef.keySet()) {
    String labelString = labelRef.get(labelInt);
    Instruction instr = code.get(labelInt);
    if(instr.getInstruction().equals("beq") || instr.getInstruction
        ().equals("bleq")) {
        code.set(labelInt, new Instruction(instr.getInstruction(),
            instr.getArg1(), 0, instr.getArg2(), labelAdd.get(
            labelString).toString()));
    }else if(instr.getInstruction().equals("b") || instr.
        getInstruction().equals("jal")) {
        code.set(labelInt, new Instruction(instr.getInstruction(),
            labelAdd.get(labelString).toString(), 0, null, null));
    }
}
```

Questo procedimento è necessario perchè la sostituzione può avvenire solamente dopo aver letto tutto il codice, dato che non si conosce a che indirizzo si trovi l'istruzione indicizzata a priori.

4.3 Memoria

La memoria è stata creata a partire dall'implementazione presente nel file *SimpLan* per poi venire modificata per soddisfare le necessità dovute all'esistenza dello heap. La struttura della memoria si divide in due parti, una parte che si occupa degli elementi allocati staticamente che crescono a partire dall'indirizzo di memoria più grande e risalgono lo stack, mentre la seconda parte inizia dall'indirizzo minore e cresce in direzione opposta.

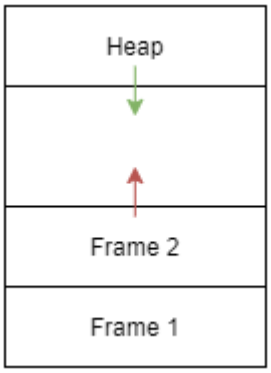

Figure 2: Struttura della memoria

Ogni frame è composto da diversi elementi, mostrati in Figura 3.

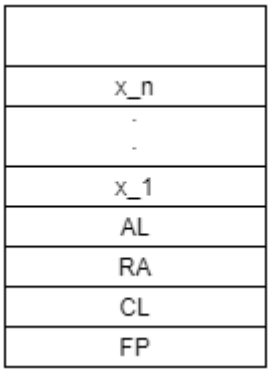

Figure 3: Struttura del frame

Ogni frame, o record di attivazione, corrisponde all'ingresso in un nuovo scope all'interno del codice, quando si esce dal nuovo scope il frame viene cancellato, ossia lo stack pointer viene riportato alla testa del frame precedente. In questo modo le istruzioni successive hanno spazio in memoria e possono essere eseguite.
Le singole celle di memoria vengono gestite tramite una classe con due campi. Il campo data è semplicemente il dato contenuto all'interno della cella di memoria, mentre il campo isUsed è il campo che serve a determinare se una cella sullo heap è in utilizzo o se è stata liberata con l'utilizzo dell'istruzione delete. All'interno della macchina virtuale che esegue poi il codice è presente un'istruzione che permette di ottenere l'indice della prima cella dello heap libera:

```
private int getFirstHeapMemoryCell() {
        int i = 0;
        while(!memory[i].isFree())
```

```
            i++;
        return i;
    }
```

4.4 Interprete

L'interprete di base è un ciclo che legge le istruzioni prodotte dal compilatore e le esegue. Questa lettura avviene in un ciclo che si interrompe nel momento in cui viene letta l'istruzione *halt* o nel momento in cui viene esaurita la memoria disponibile. Nel caso in cui la memoria si esaurisca viene sollevata una MemoryAccessException. Considerata la gestione dei puntatori in SimpLanPlus se l'utente non inizializza i valori delle variabili puntate questi poi possono accedere ad aree di memoria non appartenenti al programma; viene quindi sollevata un'eccezione con il messaggio "Segmentation Fault". All'interno del ciclo viene utilizzato un switch per leggere le istruzioni del codice.
L'interprete viene quindi inizializzato con il codice generato dal compilatore e con la dimensione massima della memoria. È presente inoltre una variabile, *ip*, che punta all'istruzione successiva da eseguire e che viene incrementato ogni volta che un'istruzione viene letta.

5 Conclusioni

È stato correttamente implementato un compilatore e un interprete che, combinati, permettono di eseguire codice di un linguaggio di programmazione "giocattolo". Il lavoro ha permesso agli autori di capire meglio ed esercitare alcuni dei meccanismi di base della teoria dei linguaggi di programmazione. Il linguaggio, per quanto semplificato, permette di calcolare funzioni di base come il fattoriale utilizzando la ricorsione.