Implementazione di un compilatore per linguaggio Object Oriented

Lorenzo Dematté

21 gennaio 2004

Sommario

In questo documento, parte del mio esame di "Linguaggi Formali e Compilatori" presso l'università degli Studi di Trento, presento un compilatore per un linguaggio Object Oriented C++ - like, SMC, e le particolarità, le caratteristiche e le difficoltà riscontrate durante la progettazione e l'implementazione dello stesso.

1 Il linguaggio SMC

Il linguaggio SMC è un linguaggio Object Oriented ispirato al C++. Ovviamente il linguaggio è molto semplificato per ragioni di complessità; tuttavia le caratteristiche più importanti di un linguaggio orientato agli oggetti sono presenti in SMC, e le maggiori problematiche che si riscontrano nella progettazione e nella realizzazione del compilatore per tale linguaggio sono state affrontate e risolte. In particolare, le seguenti caratteristiche sono supportate dal linguaggio:

- ereditarietà;
- incapsulamento;
- gestione della memoria dinamica;
- passaggio di oggetti come parametri a funzioni;
- polimorfismo attraverso funzioni virtuali;
- corretta gestione dello scope per le variabili membro;
- overloading.

Inoltre, per quanto riguarda la parte imperativa del linguaggio, sono state implementate le seguenti caratteristiche:

- passaggio di parametri per valore / per riferimento;
- chiamate a funzione *native*, cioè funzioni di libreria implementate dalla VM (si veda [1]);
- dichiarazione di variabili alla C++ (ovvero non solo all'inizio di un blocco, ma ovunque occorra).

Il compilatore poi è stato predisposto, mediante un design opportuno, per supportare e implementare agevolmente altre caratteristiche del linguaggio attualmente non previste. Tra queste:

- variabili e riferimenti **const**;
- namespace;
- variabili di classe (**static**), variabili statiche;
- ereditarietà multipla;
- interfacce.

Sono stati implementati ma non approfonditamente testati gli *array*; tale costrutto è da ritenersi sperimentale.

Per esempi di programmi scritti in SMC che implementano le caratteristiche sopra citate, si rimanda all'appendice.

2 La struttura del compilatore

Il compilatore è stato implementato in C++ secondo il modello a oggetti, seguendo un design ormai consolidato tra i compilatori, che prevede la distinzione tra front-end e back-end. Il front-end, costituito da parser, scanner, e symbol table, si occupa della costruzione di un parse tree, che viene poi usato dal back-end per produrre codice oggetto. In particolare il back-end è a sua volta diviso in due parti: la prima prende come input il parse-tree e genera il corrispondente codice IL (Intermediate Language, codice intermedio), la seconda prende come input la Symbol Table e il codice IL per generare l'assembly vero e proprio. Questa ulteriore suddivisione consente, se in futuro si vorrà implementare questa caratteristica, di costruire agevolmente un compilatore per linguaggio target differente. E' sufficiente sostituire la parte che da codice IL e Symbol Table genera l'assembly.

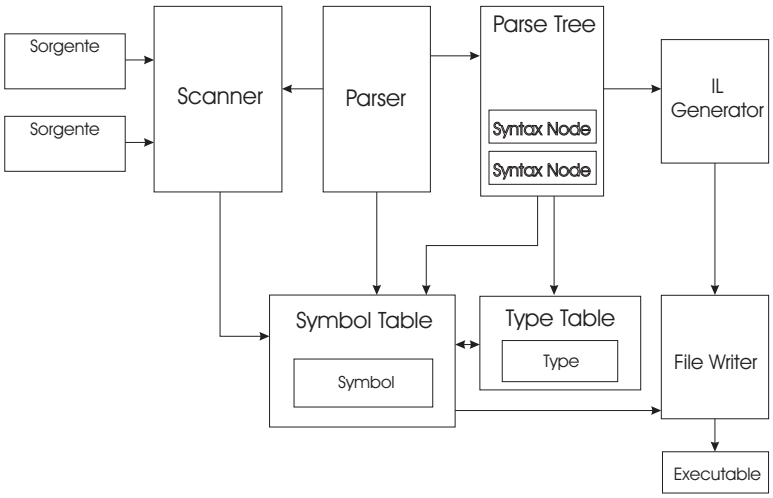

Figura 1: La struttura a moduli del compilatore

Di seguito sono illustate in maggiore dettaglio i più rilevanti di questi moduli.

2.1 Symbol Table e Type Table

In un linguaggio ad oggetti uno dei fattori più complessi è la gestione dello scope e della ricerca dei nomi nello namespace corrente. Una stessa variabile o funzione può essere definita più volte, ma la ricerca nello scope deve sempre dare il binding corretto. Come esempio, si consideri il seguente blocco di codice:

```
int j;

class pippo
{
public:
```

```
    int i;
    int j;

    int f(int i)
    {
        return i * j; //i is the parameter, j is member
    }
};
```

Nella funzione **f** la variabile **i** deve essere legata al parametro e la variabile **j** alla variabile membro della classe pippo. Un modo efficiente per la ricerca e la risoluzione del binding di variabili è ottenibile ricorrendo a una Symbol Table che raccolga le informazioni su simboli e sulle definizioni dei tipi in modo gerarchico. Di seguito sono presentate parti delle due strutture che raccolgono tali informazioni: la classe *SymDef* e la classe *TypeRec*

```
class SymDef
{
public:
    //the name of the symbol
    std::string name;

    //the type of the symbol
    TypePtr type;

    //the owner of this symbol (scope)
    SymPtr parent;

    //symbol kind (class, variable, function...)
    SYMBOL_KIND symKind;

    //the list of sons (symbols declared in this scope)
    std::list<SymPtr> declList;

    //CLASS MEMBERS
    //NAMESPACE MEMBERS
    //SCOPEBLOCK MEMBERS
    //FUNCTION MEMBERS
    //VARIABLE   MEMBERS
    //TYPEDEF MEMBERS
};
```

```
class TypeDef
{
public:
   TYPE_TYPE typeID;

   //types used by this one
   std::deque<TypePtr> typeList;

   //type modifiers
   bool isConst;
   bool isReference;
   bool isPointer;

   int arraySize;
   std::string name;
   SymPtr symbol;
};
```

Come si può vedere dalla dichiarazione, la classe *SymDef* rappresenta la definizione di un simbolo valido all'interno del programma da compilare. Ogni simbolo ha un puntatore al padre e contiene la lista dei simboli figli, ovvero la lista dei nomi definiti direttamente all'interno del suo scope[1]. Questo consente di creare una struttura ad albero che rappresenta direttamente la struttura gerarchica dei simboli e permette di ricercare un simbolo efficientemente. All'interno di un oggetto SymDef vengono inoltre immagazzinate tutte le informazioni utili riguardanti il simbolo. Nell'implementazione, al posto del vari commenti, sono presenti i membri della class SymDef rilevanti rispettivamente quando la specie del simbolo è `class`, `namespace`, `function`, `variable`, ecc. Questi campi contengono le informazioni più disparate, valide a seconda della specie del simbolo: per le variabili, ad esempio, se è parametro o membro di classe, per le funzioni se è virtuale e il suo eventuale indice all'interno della tabella delle funzioni virtuali, e così via.
Strettamente collegata alla *symbol table* e ai suoi record (gli oggetti SymDef) è la *type table* con i suoi record (costituiti da oggetti TypeRec). Analogamente alla precedente, tale tabella ha una struttura gerarchica rappresentata tramite un albero. I due alberi sono correlati tra di loro: ogni SymDef contiene un puntatore a un TypeRec che rappresenta il tipo del simbolo. Inoltre per quanto riguarda i tipi di dato definiti dall'utente (nel nostro caso, le classi) il TypeRec contiene un puntatore al simbolo che definisce l'UDT[2].

[1] I simboli definiti direttamente all'interno dello scope variano a seconda del simbolo padre: per una funzione sono i parametri e le variabili locali, per una classe sono variabili e funzioni membro, ecc..

[2] UDT: User Data Type, tipo di dato definito dall'utente

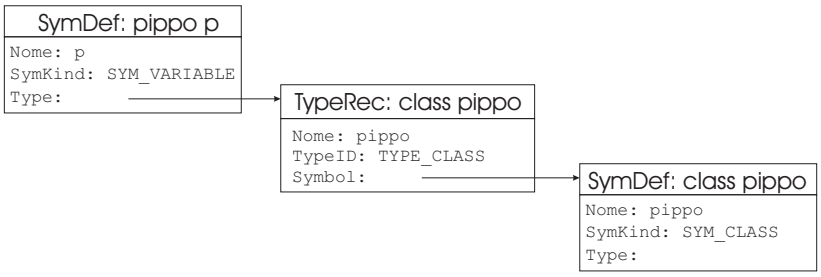

Figura 2: Esempio di relazione tra gli elementi della symbol table e della type table

Per esempio, il codice seguente:

```
class pippo
{
public:
   int i;
};

pippo p;
```

porta alla struttura in figura 2

3 Il linguaggio assembly e la VM (Virtual Machine)

Il compilatore SMC attualmente ha come linguaggio target un linguaggio assembly per una macchina virtuale costruita ad hoc. Questo consente di trattare in modo più semplice aspetti come funzioni virtuali, gestione della memoria dinamica, ecc. concentrandosi sulle funzionalità.

Il linguaggio assembly e la macchina virtuale sono stati pensati per lavorare insieme; le loro principali caratteristiche sono:

- unica forma per i dati, sia in memoria che su disco: gli operandi delle istruzioni, gli elementi dello Stack, i registri della macchina virtuale, le variabili membro degli oggetti sono tutti tipi di dato `struct Operand`

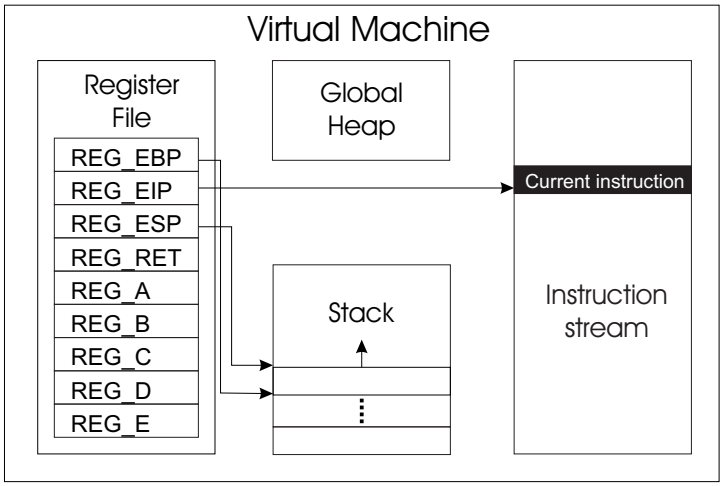

Figura 3: La Virtual Machine

- la macchina virtuale è costruita secondo lo schema in figura 3: ha 9 registri (4 riservati, 5 disponibili all'utente), uno spazio di memoria dinamico (heap), uno stack e una CPU virtuale che esegue il programma in input passo dopo passo, completando per ogni istruzione le seguenti fasi: decodifica dell'istruzione, reperimento degli operandi, esecuzione, memorizzazione del risultato.

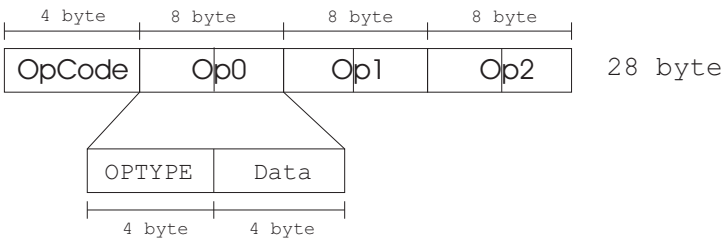

Figura 4: Struttura delle istruzioni e degli operandi

- per ragioni di semplicità, le istruzioni hanno lunghezza fissa e struttura a *three operands*, come in figura 4

La struttura che consente di immagazzinare ogni dato all'interno del file eseguibile e nella memoria della VM è illustrata in figura 4.
Questa struttura, chiamata comunemente *tagged union*, permette di memorizzare al suo interno tipi di dati diversi; il significato del dato contenuto è poi interpretato in maniera diversa a seconda del valore del tag. Nella seguente tabella sono illustati i vari tipi di dati memorizzabili da una struttura `Operand`. I vari tipi di Indirizzamento nella terza colonna sono illustrati in figura 5

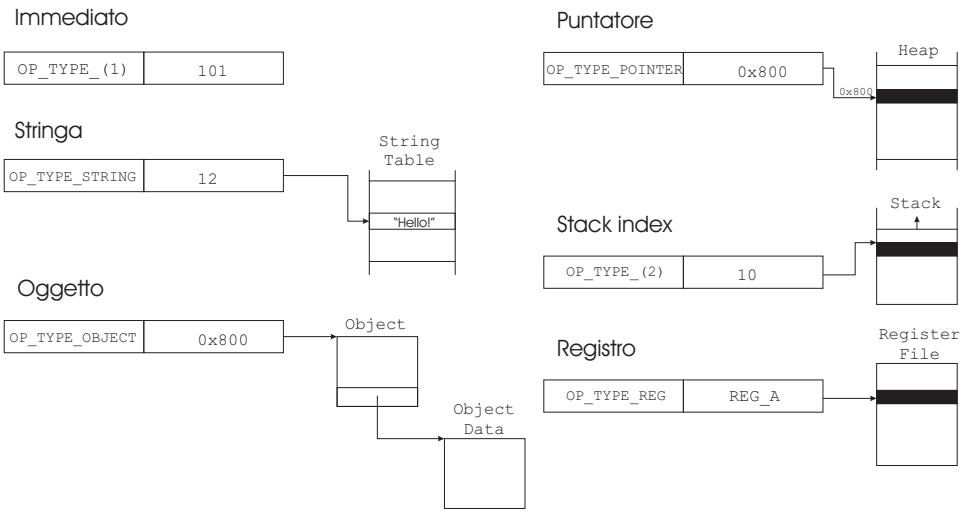

Figura 5: I vari tipi di indirizzamento. Per quelli nella colonna di destra, in nero è evidenziato il dato (sempre `Operand`) su cui vengonono effettivamente svolte le operazioni. `OP_TYPE_(1)` sta per `INT, FLOAT, INSTR_INDEX, FUNC, OBJECT_TAG` e `TYPE_TAG`; `OP_TYPE_(2)` sta per `ABS_STACK_INDEX` e `REL_STACK_INDEX`.

Tag	**Tipo del dato**	**Indirizzamento**
`OP_TYPE_INT`	integer (32-bit)	Immediato
`OP_TYPE_FLOAT`	floating point (32-bit)	Immediato
`OP_TYPE_STRING`	stringa	Stringa
`OP_TYPE_ABS_STACK_INDEX`	integer (32-bit)	Stack index
`OP_TYPE_REL_STACK_INDEX`	integer (32-bit)	Stack index
`OP_TYPE_OBJECT`	Object*	Oggetto
`OP_TYPE_POINTER`	pointer	Puntatore
`OP_TYPE_ARRAY`	ObjectArray*	Oggetto
`OP_TYPE_INSTR_INDEX`	integer (32-bit)	Immediato
`OP_TYPE_FUNC`	integer (32-bit)	Immediato
`OP_TYPE_REG`	integer (32-bit)	Registro
`OP_TYPE_OBJECT_TAG`	tag	Immediato
`OP_TYPE_TYPE_TAG`	tag	Immediato

Le istruzioni assembly con la descrizione del loro significato e degli operandi sono presentate nella seguente tabella:

Istruzione	No. op	Tipo op	Descrizione
`ADD, SUB, MUL, DIV, MOD`	3	`INT`	Eseguono rispettivamente addizione, sottrazione, moltiplicazione, divisione, modulo tra gli operandi 1 e 2 e salvano il risulatato nell'operando 0. Esempio: `ADD REG_A REG_B REG_C: op[REG_A] := op[REG_B] + op[REG_C]`
`INC, DEC`	1	`INT`	Incrementano o decrementano di un'unità l'operando 0.
`MOV`	2	any	Recupera l'elemento nell'operando 1 e lo assegna al dato *puntato* dall'operando 0. Esempio: `REG_A = {OP_TYPE_POINTER, 0x300}` `MOV REG_A 10: heap[0x300] := 10` `REG_A` resta invariato.
`SET`	2	any	Recupera l'elemento nell'operando 1 e lo assegna all'operando 0. Esempio: `REG_A = {OP_TYPE_POINTER, 0x300}` `SET REG_A 10:` `REG_A = {OP_TYPE_INT, 10}` `heap[0x300]` resta invariato.
`PUSH`	1	any	Mette il dato indicato nell'openado 0 sullo stack. Nel caso di un operando di tipo `OBJECT` incrementa il contatore di riferimenti.
`POP`	1	any	Prende il dato indicato in cima allo stack e lo muove nella locazione indicanta dall'openado 0, rimuovendolo dallo stack.
`TOP`	1	any	Mette il dato indicato in cima allo stack nell'openado 0. Lo stack rimane invariato.
`AND, OR, XOR, LSH, RSH`	3	`INT`	Eseguono rispettivamente le operazioni bit a bit di and, or inclusivo, or esclusivo, shift a sinistra e shift a destra tra gli operandi 1 e 2, salvando il risulatato nell'operando 0.
`NEG`	2	`INT`	Esegue la negazione bit a bit dell'operando 1 salvando il risulatato nell'opernado 0.
`FADD, FSUB, FMUL, FDIV`	3	`FLOAT`	Eseguono rispettivamente addizione, sottrazione, moltiplicazione e divisione tra gli operandi 1 e 2 e salvano il risulatato nell'operando 0.
`FSQRT`	3	`FLOAT`	Esegue l'estrazione di radice quadrata sul numero referenziato da op1 salvando il risulatato in op0.

Istruzione	No. op	Tipo op	Descrizione
`CALL`	2	`TAG, INT`	Chiama la funzione con ordinale in op1 membro dell'oggetto con ID in op0.
`VCALL`	1	`INT`	Chiama la funzione virtuale con ordinale in op0 membro dell'oggetto corrente.
`RET`	1	`INT`	Ritorna dalla funzione. Il valore in op0 indica il numero di posizioni sullo stack da liberare.
`I2F, F2I, STR2I, STR2F, I2STR, F2STR`	2	vari	Istruzioni di conversione rispettivamente da (op1) a (op0): `INT -> FLOAT` `FLOAT -> INT` `STRING -> INT` `STRING -> FLOAT` `INT -> STRING` `FLOAT -> STRING`
`UCAST, DCAST`	3	`OBJECT, OBJECT, TAG`	Upcast e downcast dell'oggetto in op1. Op2 contiene l'ID della classe target. `DCAST` può generare un'eccezione se il tipo non è corretto.
`STORE`	2	`POINTER`, any	Memorizza nella locazione di memoria puntata da op0 il dato referenziato da op1.
`RELEASE`	1	`OBJECT`	Rilascia un riferimento a un oggetto. Se il reference count per l'oggetto arriva a 0, questo viene cancellato.
`PADD, PSUB`	3	`POINTER, POINTER, INT`	Aggiunge o sottrae un offeset specificato in op2 dall'indirizzo di memoria in op1, memorizzando il risultato in op0.
`LEA`	2	`POINTER`, vari	Carica l'indirizzo di memoria di op1 in op0. Equivalente a `op0 = &op1`
`LEAA`	2	`POINTER, ARRAY, INT`	Carica l'indirizzo di memoria dell'elemento dell'array in op1 con indice in op2 nell'operando referenziato in op0. Equivalente a `op0 = &(op1[op2])`
`NEW`	2	`OBJECT, TAG`	Crea un nuovo oggetto del tipo specificato in op1.
`NEWA`	3	`OBJECT, TAG, INT`	Crea un array con elementi del tipo specificato in op1 e dimensione specificata in op2.
`CLONE`	2	`OBJECT`	Clona l'oggetto referenziato dall'operando 1. Un clone condivide l'area dati dell'oggetto, ma ha un diverso riferimento e, possibilmente, un diverso tipo.
`HALT`	0		Arresta la VM (ultima istruzione eseguita).

Istruzione	No. op	Tipo op	Descrizione
`JMP`	1	`INT`	Salto incondizionato all'istruzione in op0.
`JNEG JZ JNZ`	2	`INT or FLOAT, INT`	Salto all'istruzione in op1 se $op0 < 0$, $op0 = 0$, $op0 \neq 0$
`JE, JNE, JLE, JGE, JL, JG`	3	`INT or FLOAT, INT or FLOAT, INT`	Salto all'istruzione in op2 se $op0 = op1$, $op0 \neq op1$, $op0 \leq op1$, $op0 \geq op1$, $op0 < op1$, $op0 > op1$,

4 Peculiarità e difficoltà riscontrate

Di seguito sono illustrati gli aspetti di design e di implementazione che distinguono un compilatore per un linguaggio imperativo da un compilatore per un linguaggio ad oggetti; inoltre sono discussi gli aspetti che si sono rivelati più problematici o più interessanti.

4.1 TYPENAME vs. IDENTIFIER

Facendo il parsing di qualsiasi linguaggio C/C++ like che permetta la definizione di tipi da parte dell'utente si presenta un interessante problema[4]. Anzitutto bisogna notare che esistono due forme distinte di 'identificatori' che possono essere letti dal lexer durante la fase di analisi lessicale, identificati entrambi come simboli terminali nella mia grammatica. Questi token terminali sono chiamati IDENTIFIER e TYPENAME; questa distinzione è necessaria a causa di un elemento fondamentale dei linguaggi C-like. Un lexeme di tipo TYPENAME si presenta come un identificatore standard, ma è definito nella symbol table come un tipo di dato dichiarato in precedenza (class, nel nostro caso, ma anche typedef, struct, enum). Tutti gli altri lexeme che compaiono come identificatori e che non sono keywords del linguaggio sono tokenizzati come IDENTIFIER.
Perché questa distinzione è necessaria? Si consideri il codice seguente:

```
class A;
class B;

A f(B); //dichiarazione di funzione f:  B -> A
                //B : TYPENAME
```

Come si puo' capire facilmente, `A f(B);` è una dichiarazione di funzione. Ma se cambiamo il *contesto*:

```
class A;
int B = 10;

A f(B); //dichiarazione di variabile f oggetto della classe A
            //con un costruttore che richiede un intero
            //B: IDENTIFIER
```

Ora `A f(B);` è la dichiarazione di una variabile. Come fa un parser LALR(1)

come quello generato da Yacc a distinguere i due casi? Da questo esempio possiamo capire che non si può eseguire il parsing dello statement `A f(B);` in modo indipendente dal contesto.

La soluzione standard per l'ambiguità precedente è quella di permettere al lexer di costruire token differenti basandosi su informazioni contestuali. L'informazione contestuale usata è la risposta alla domanda: "l'identificatore dato è un nome di tipo al punto corrente del parsing?". Questo feedback loop (con il parser che genera le informazioni di contesto inserendole nella symbol table e il lexer che ne fa uso per generare nuovi token), è conosciuta come *lex hack*. Grazie al lex hack il frammento di codice `A f(B);` viene presentato dal lexer al parser come

```
IDENTIFER IDENTIFIER '(' TYPENAME ')' ';'
```

oppure come

```
TYPENAME IDENTIFIER '(' TYPENAME ')' ';'
```

due casi che il parser distingue senza difficoltà.

4.2 Accesso a variabili e funzioni membro

Come già affermato in precedenza, una delle caratteristiche più complesse in un liguaggio ad oggetti è la gestione corretta dello scope. Questo succede sia a compile-time (per il binding della variabile/funzione corretta, come specificato nella prima sezione e nelle due sezioni successive), sia a run-time, per quanto riguarda l'accesso alle variabili membro e ai metodi dell'oggetto corretto. Si consideri il seguente esempio:

```
class C {
public:
   int c;
}

class B {
public:
     C a();
}

int main() {
   A b;
   b.a().c = 1;
   print((string)b.a().c);
}
```

In corrispondenza di un operatore di accesso ai membri (.) e dell'uso di un identificatore, sia esso variabile o nome di funzione, il compilatore deve generare le corrispondenti corrette istruzioni assembly.

```
b.a().c = 1
```

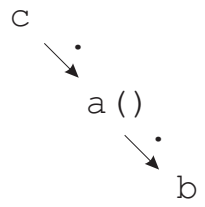

Figura 6: Parse tree generato per l'espressione `b.a().c`

Come premessa, è bene ricordare che il parser del compilatore SMC è un parser generato con YACC/Bison, ed è quindi un *recursive descendent parser*. Il parse tree generato dal codice di esempio è quindi il codice in figura 6.

Ecco come il processo funziona nei vari casi:

- Se l'identificatore è una **variabile**:
  - se tale variabile si trova in posizione **iniziale** (cioè non ha figli nel parse tree) ,ed è una variabile globale o un parametro o una variabile locale (cioè se si trova sullo stack), recupero l'indirizzo della variabile e lo metto nel regitro `REG_D`:
    ```
    SET REG_D, identifier
    ```
  - se tale variabile si trova in posizione **iniziale** (cioè non ha figli nel parse tree), ed è una variabile membro della classe corrente bisogna considerare il riferimento `this` implicito: l'espressione *expr* deve essere considerata come `this.`*expr*.
    Devo quindi recuperare il riferimento a `this` dallo stack e caricarlo nel registro `REG_D`, aggiungendo poi l'offset della variabile come in figura 7:
    ```
    LEA REG_D, this;
    PADD REG_D, REG_D, identOffset;
    ```
  - se tale variabile si trova in posizione **intermedia** o **finale** (cioè se ha figli nel parse tree), è sicuramente membro di un oggetto: nel registro `REG_D` ricevo già l'indirizzo base dell'oggetto e ottengo l'indirizzo dell'area dati e quindi del membro specifico con la coppia di istruzioni
    ```
    LEA REG_D, REG_D;
    PADD REG_D, REG_D, identOffset;
    ```
- Se l'identificatore è una **funzione** quello che devo fare è generare il codice necessario a mettere sullo stack il puntatore a `this` corretto per la funzione

```
class C
{
public:
  int n;
  A b;
  void f()
  {
    b.a().c = 1;  ⟶  this.b.a().c = 1;
  }
};
```

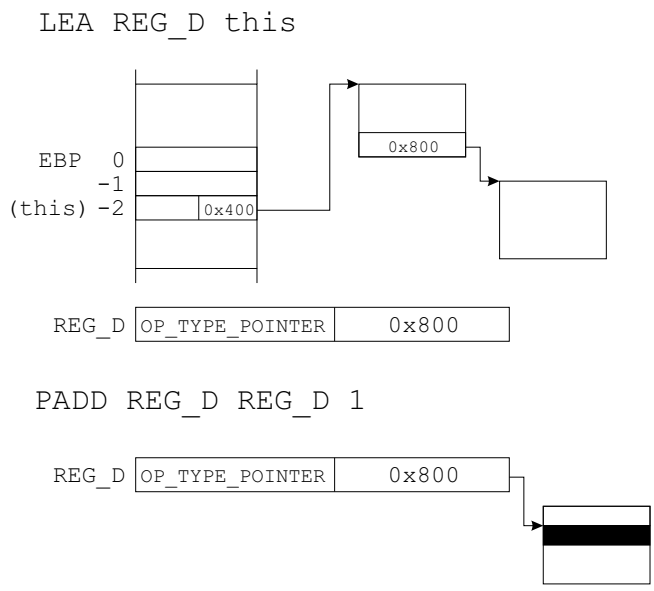

Figura 7: Parse tree generato per l'espressione `b.a().c`

chiamata, dopodiché altri nodi del parse tree si occuperanno di generare il codice necessario a fare il push degli altri parametri e la chiamata a funzione. Anche in questo caso, bisogna distinguere due casi:

- se il nodo rappresentante la chiamata nel parse tree non ha figli, si tratta di una chiamata di funzione in prima posizione che può ritornare un lhs; ad esempio: `a().b`. In questo caso, trattandosi di una chiamata nello scope corrente, quello che il compilatore deve generare è il codice necessario a mettere sullo stack un riferimento a `this` analogo a quello corrente:

  ```
  CLONE REG_D, this;
  PUSH REG_D;
  ```

- se il nodo rappresentante la chiamata nel parse tree ha figli, si tratta di una chiamata di funzione con scope; esempio: `a.b()`. In questo caso,

il `this` all'interno del corpo della funzione `b()` è proprio un riferimento all'oggetto `a`. Poniamo quindi che il codice generato in precedenza porti ad avere un riferimento ad `a` nel registro `REG_D`:

```
CLONE REG_D, REG_D;
PUSH REG_D;
```

4.3 Overloading e Overriding

La ricerca di un simbolo all'interno dello scope corrente è piuttosto semplice: è sufficiente un cammino all'interno dell'albero di simboli costituente la symbol table, descritto nella sezione *Symbol Table e Type Table*. Tuttavia, per quanto riguarda le funzioni, la faccenda è complicata da un altro fattore: l'overloading. In caso di più nomi definiti per una stessa funzione, infatti, il problema è diverso: bisogna trovare la funzione "più adatta" per svolgere il compito. Il valore che si calcola per ogni funzione, per capire quanto sia "adatta" a essere invocata in un particolare contesto, si dice *fitness value* ed è una funzione che prende come input il tipo dei parametri attuali (con i quali la funzione viene chiamata), il tipo dei parametri formali della funzione candidata e una tabella di costi. L'algoritmo reperisce tutti i record presenti nella *symbol table* aventi lo stesso nome e tipo `TYPE_FUNCTION` e per ognuna di queste funzioni candidate controlla la visibilità nello scope corrente e ne calcola il fitness value usando la seguente procedura:

```
function BakerDitchfield( this_name : name, args : list<set<tree>>)
   : tree_list
begin
   result_trees : set<tree> = {}
   for each interpretation (formals) corresponding to this_name
      if formals.length == args.length then
         new_tree : tree
         new_tree.type = return_type
         new_tree.cost = 0
         for i = 1 to formals.length
            if there exists j in args[i] such that j.type can be converted
               to formals[i].type then
               find k in args[i] such that conversion_cost(k.type,
                  formals[i].type) + k.cost is minimized
               new_tree.child[i] = k
               new_tree.cost += k.cost
               new_tree.cost += conversion_cost(k.type,
                formals[i].type)
            else
               skip to next interpretation
               (i.e. this interpretation is not consistent with the
               possible argument types)
          end if
```

```
          end for
          if this interpretation is valid then
              if there exists i in result_trees such that i.type ==
new_tree.type
              then
                  if i.cost == new_tree.cost then
                      i.ambiguous = true
                  else if i.cost < new_tree.cost then
                      remove i from result_trees
                      add new_tree to result_trees
                  ( else throw away new_tree )
                end if
              else
                  add new_tree to result_trees
            end if
          end if
       end if
   end for
   for each tree in result_trees
       if tree.ambiguous then
          remove tree from result_trees
       end if
   end if
   return result_trees
end
```

Se più di una funzione ha lo stesso fitness value, viene generato un errore.

Ancora diverso è il caso di *function overriding*: in questa situazione, una funzione viene nascosta da una funzione con nome **e** parametri identici. Questa situazione si verifica quando ridefiniamo una funzione presente in una classe base nella classe derivata; per esempio:

```
class A {
public:
   void f() { ... }
};

class B : public A {
public:
   void f() { ... }
   void g() { f(); /*B::f() hides A::f()*/ }
};
```

Nel caso in cui siano coinvolte funzioni virtuali, vengono innescati meccanismi più complessi per produrre un comportamento corretto. Questi meccanismi sono

discussi nella sezione seguente.

4.4 Chiamate a funzione virtuale: polimorfismo

Se si fa l'*overriding* di una funzione normale, quello che si ottiene quando si chiama tale funzione è una chiamata a funzione con binding statico. In pratica, compilando con SMC il seguente programma, l'output che si ottiene è "Instrument::play"
Esempio: Instrument

```
#include output.hs
class Instrument
{
public :
   void play ()
   {
      print ( Instrument::play );
   }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : Instrument
{
public :
   // Redefine interface function:
   void play ()
   {
      print ( Wind::play );
   }
};

void tune ( Instrument & i )
{
   i . play ();
}

int main ()
{
   Wind flute ;
   tune ( flute ); // Upcasting

   return 0 ;
}
```

Ma se si aggiunge la keyword **virtual** davanti al metodo *Instrument::play*, il comportamento cambia: l'output è "Wind::play" grazie al fatto che la keyword virtual dice al compilatore che c'è bisogno di un comportamento *polimorfo* e di usare quindi il binding dinamico. La funzione da chiamare è decisa a run-time in modo efficiente tramite l'istruzione assembly VCALL e la cosiddetta v-table, la tabella delle funzioni virtuali collegata ad ogni oggetto. Vediamo come funziona tutto il meccanismo, cosa succede in fase di compilazione e cosa in fase di esecuzione.

4.4.1 Fase di compilazione

In fase di compilazione, durante la dichiarazione di una funzione virtuale f come membro di una classe **A**, il compilatore esegue le seguenti operazioni:

- come nel caso di funzioni non virtuali, il compilatore costruisce il simbolo per la funzione f (oggetto SymDef);
- se la funzione f è la prima ad essere dichiarata come virtuale per la classe **A**, il compilatore costruisce la v-table per tale classe. La v-table, tabella delle funzioni virtuali, è un array di simboli (oggetti SymDef) che viene associato alla classe di cui la funzione è membro;
- il simbolo della funzione f viene aggiunto alla v-table: ci sono ora due ulteriori possibilità:
  - non siamo in presenza di un *override*: nessun simbolo per f è già stato definito in una classe antenata di **A** (oppure **A** è alla base della gerarchia):
    alla v-table di **A** viene aggiunto un nuovo record con il simbolo per f;
    al campo `virtualIndex` del simbolo viene assegnato l'indice di questo nuovo record;
  - siamo in presenza di un *override*, e il simbolo per f definito in precedenza è anch'esso dichiarato come **virtual**:
    nella v-table di **A** viene reperito l'indice del precedente simbolo per f;
    a questa posizione viene inserito il nuovo simbolo per f sovrascrivendo il vecchio.

Affinché le v-table siano costruite correttamente è inoltre necessario che quando viene dichiarata una nuova classe **B** derivata da una classe precedente **A**, se **A** ha una v-table, questa sia copiata identica in **B**.
I passi di questo algoritmo sono riassunti ed esemplificati nella figura8.

Durante la chiamata a funzione, il compilatore vede se la funzione chiamata è virtuale o meno, e in tal caso genera un'istruzione `VCALL virtual_function_index` al

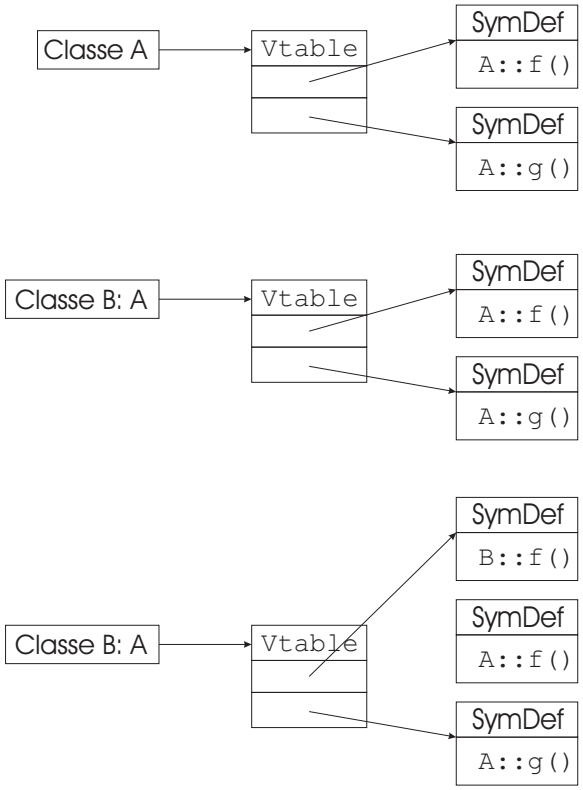

Figura 8: Dichiarazione di funzioni virtuali: in alto la v-table di **A** con due funzioni virtuali, *f* e *g*, al centro la v-table di **B** subito dopo la dichiarazione della classe, in basso la v-table di **B** dopo l'overriding della funzione virtuale *f*

posto di una canonica `CALL object_id function_index`. Il perché non sia necessario l'object_id nel caso di una chiamata virtuale sarà chiaro in seguito, quando analizzeremo il comportamento in fase di esecuzione.

Infine, durante la generazione del codice eseguibile, il compilatore scrive dopo l'header dell'eseguibile e la string table un'ulteriore tabella, detta object table. Tale tabella contiene informazioni su tutte le classi presenti nel programma generato; tra queste informazioni ci sono l'offset all'interno della sezione di codice eseguibile di tutte le funzioni della classe e la sua v-table.

4.4.2 Fase di esecuzione

Per capire meglio cosa accade in fase di esecuzione, vediamo il codice generato dal compilatore per il programma di esempio Instrument, con dichiarazione di funzione normale e con dichiarazione di funzione virtuale:

Chiamata normale

```
27:  PUSH REG_D
```

```
28:  CALL object no:3338408 func #0
```

Chiamata virtuale

```
27:  PUSH REG_D
28:  VCALL func #0
```

Come descritto nella sezione “Il linguaggio assembly”, nella macchina virtuale per SMC ogni tipo di dato è incapsulato in un Operand, che contiene il valore e un tag per identificare il tipo del dato correntemente ospitato nella struttura.

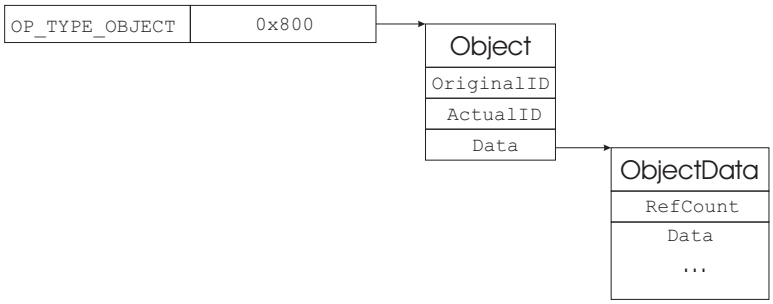

Figura 9: Qualsiasi oggetto viene rappresentato nella memoria della VM in questo modo. La grandezza della sezione `Data` dipende dalla dimensione dell’oggetto

Nel caso in questione, il registro D della macchina virtuale contiene un tipo di dato `OP_TYPE_OBJECT`: il parametro **this** per la funzione chiamata. Al tipo di dato `OP_TYPE_OBJECT` corrisponde la struttura descritta in figura 9 I campi OriginalID e ActualID contengono l’ID univoco della class a partire dalla quale è stato creato l’oggetto. In particolare il campo OriginalID non viene mai modificato; in questo modo anche se passo l’oggetto tramite un riferimento alla classe base, come accade nell’esempio, la VCALL sa sempre qual’è la classe originale.

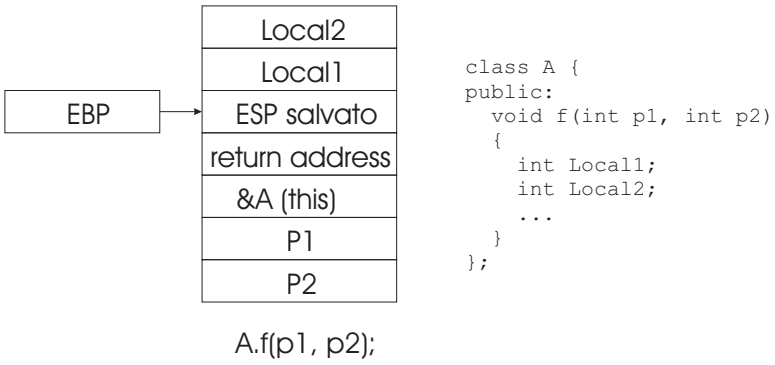

Figura 10: La configurazione dello stack al momento dell’esecuzione della prima istriuzione della funzione

Quindi nel momento in cui si esegue una chiamata a funzione il puntatore a **this** è

sempre presente sullo stack in posizione prefissata; un esempio di configurtazione dello stack al momento della chiamata è rappresentata in figura 10.
I passi di una chiamata virtuale si riassumono quindi ai seguenti (figura 11):

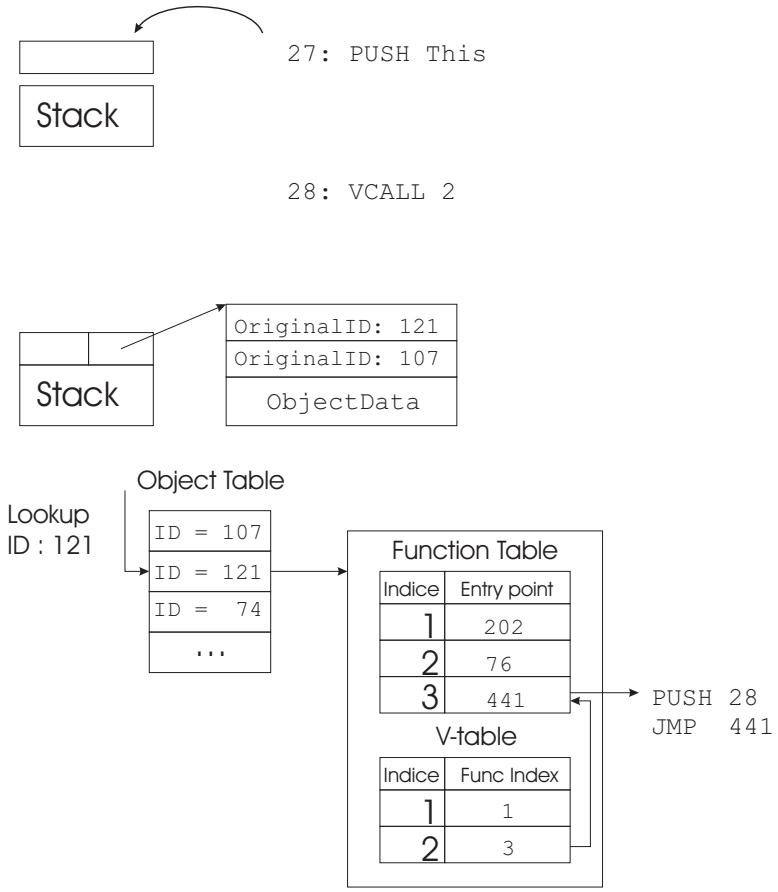

Figura 11: Una chiamata a funzione virtuale: usando il parametro implicito This messo sullo stack, si ricava l'ID della classe originale dell'oggetto, e da questo tutte le informazioni necessarie alla chiamata della funzione.

- reperimento di **this** sullo stack;
- da questo, otteniamo l'OriginalID dell'oggetto;
- usiamo l'ID come chiave per trovare le informazioni sulla classe corrispondente nella Object Table;
- tra queste vi sono la v-table e le l'entry point di tutte le funzioni definite in quella casse;
- con queste informazioni e il virtualIndex, l'operando della VCALL, si reperisce l'entry point corretto;

- si esegue un salto incondizionato a questo indirizzo, salvando il return address sullo stack.

Per quanto riguarda una chiamata a funzione standard, il procedimento è più semplice e più veloce: l'ID della classe è già noto al tempo di compilazione ed è quindi inserito come primo operando dell'istruzione CALL. Inoltre l'indice della funzione, specificato come secondo operando, non è un indice per la v-table, ma direttamente un indice per la Function Table di quella classe.

A Appendice: gli esempi

Di seguito sono elencati e brevemente illustrati gli esempi forniti a corredo

Nome	**Descrizione**
`array.cs`	illustra l'utilizzo degli array e il passaggio di questi a una funzione.
`instrument1.cs`	esempio ereditarietà: no polimorfismo.
`instrument2.cs`	esempio ereditarietà: polimorfismo con funzioni virtuali.
`instrument3.cs`	esempio ereditarietà a più livelli con polimorfismo.
`ObjectObject.cs`	oggetti membro di oggetti, funzioni che ritornano riferimenti a oggetti membro.
`perlin.cs`	generatore di perlin noise: test delle principali capacità aritmetiche del linguaggio e della VM, test sulle chiamate native.
`prova1.cs`	semplice test di scope sulle variabili.
`string.cs`	test sulle stringhe.
`switch.cs`	test del costrutto switch.
`while.cs`	test del costrutto while.

B Appendice: sorgenti di ispirazione

Molti libri mi hanno ispirato nel design del linguaggio: in particolare [6] per la struttura della VM, [3] per come accedere ai mebri di una classe in linguaggio assembly, [2] per l'implementazione e il funzionamento delle chiamate virtuali e [5] per la grammatica di un linguaggio con classi.

Riferimenti bibliografici

[1] S. Bilas. Fubi: Automatic function exporting for scripting and networking. Available online at http://www.drizzle.com/ scottb/gdc, 2001.

[2] B. Eckel. *Thinking in C++ 2nd Ed.* MindView, 2002.

[3] J. Robbins. *Debugging Applications*. Microsoft Press, 2000.

[4] J. A. Roskind. C++ 2.1 grammar, and the resulting ambiguities. 1991.

[5] B. Stroustrup. *The C++ programming languge, 3rd Ed.* Addison Wesley, 1997.

[6] A. Varanese and A. LaMothe. *Game Scripting Mastery*. Premier Press, 2003.