POLITECNICO DI MILANO

Corso di Laurea in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

Supporto alla programmazione generica nel compilatore ILDJIT

Politecnico di Milano

Relatore: Chiar.mo Prof. Stefano Crespi Reghizzi

Correlatore: Ing. Simone Campanoni

Tesi di Laurea di:

Luca Rocchini, matricola 720908

Anno Accademico 2009-2010

A Claudia

Indice

Elenco delle figure

Elenco dei codici

Elenco delle tabelle

Sommario

L'interesse per nuovi paradigmi di programmazione e per la compilazione dinamica è cresciuto negli ultimi anni; in particolare, hanno avuto grande sviluppo i paradigmi derivanti dal concetto di meta-programmazione, dove la programmazione generica ne è un esempio.

La programmazione generica nasce come soluzione per sviluppare algoritmi o contenitori parametrici, ricercando i requisiti minimi che i parametri devono rispettare per un implementazione efficiente.

Con lo standard internazionale ECMA-335, introdotto nel 2006, si è definita l'infrastruttura Common Language Infrastructure (CLI) per la compilazione dinamica includendo il supporto alla programmazione generica semplificandone l'integrazione nei linguaggi di alto livello.

L'obbiettivo della tesi è integrare il supporto alla programmazione generica all'interno del compilatore ILDJIT, un implementazione dello standard ECMA-335, studiando gli effetti dell'integrazione su una macchina virtuale.

1 Introduzione

La realizzazione di software complessi, come ad esempio le applicazioni multimediali, presenta alti costi e prolungati tempi di sviluppo. L'interesse per nuovi paradigmi di programmazione e i nuovi modelli di compilazione che semplifichino lo sviluppo è quindi cresciuto considerevolmente a partire dal 1990.

Nell'ambito dei compilatori gli sviluppi maggiori sono stati sia la ricerca di tecniche di compilazione flessibili, come la compilazione dinamica, sia lo sviluppo di nuovi linguaggi o paradigmi di programmazione.

Le moderne tecniche di compilazione dinamica hanno permesso di ottenere una buona efficienza nell'esecuzione dei programmi mantenendo allo stesso tempo la portabilità su un ampio insieme di architetture senza richiedere complessi e costosi progetti di adattamento. Ad oggi il caso più noto è la Java Virtual Machine [17].

Parallelamente al tema della portabilità si sono sviluppati nuovi paradigmi nei linguaggi di alto livello. La ricerca si è orientata sempre più nello sviluppo di forme efficaci ed efficienti di polimorfismo e alla studio di meccanismi per permettere l'adattabilità del linguaggio. Il risultato di queste ricerche è stato l'inclusione del supporto nei moderni linguaggi, come Java [11] o C# [13], dei paradigmi di programmazione generica e riflessiva.

La programmazione generica nasce come soluzione per sviluppare algoritmi o tipi di dato parametrici, ricercando i requisti minimi che i parametri devono rispettare per un implementazione efficiente. Il paradigma riflessivo ricerca invece meccanismi per

l'interazione del programma con il linguaggio sorgente. Attraverso questi meccanismi un linguaggio riflessivo permette al programma di agire sul linguaggio adattandolo alle necessità del codice.

Con lo standard ECMA-335 [14], l'organizzazione internazionale European Computer Manufacturers Association (ECMA) ha pubblicato la specifica di un'architettura unificata che rispondesse alle richieste degli sviluppatori.

L'ECMA-335 definisce la struttura di un ambiente di esecuzione virtuale , chiamato Common Language Infrastructure (CLI), capace di eseguire programmi espressi nel linguaggio Common Intermediate Language (CIL); esso è un linguaggio sviluppato con l'obiettivo di fornire portabilità su diversi sistemi operativi ed architetture. Il CLI si astrae da uno specifico linguaggio di programmazione ad alto livello offrendo una ricca semantica, comprensiva del supporto alla programmazione generica e riflessiva, permettendo la traduzione di nuovi linguaggi di programmazione di alto livello verso essa. Oggigiorno svariati linguaggi di programmazione appartenenti a paradigmi differenti sono supportato dallo standard (e.g C, C++, Python, Java ecc..).

Lo standard ECMA-335 è implementato da diversi fornitori. L'implementazione commerciale più usata e famosa è quella di Microsoft. Nella comunità opensource esistono diverse soluzioni: Mono [19], DotGNU [8] e ILDJIT [12] (sviluppato al Politecnico di Milano). ILDJIT è un progetto innovativo pensato per la realizzazione di un compilatore dinamico fortemente parallelo e in grado di essere eseguito su più piattaforme hardware. La particolare struttura interna di ILDJIT, costruita attorno a thread paralleli, è capace di distribuire i compiti di ottimizzazione, gli permettono di sfruttare massicciamente il parallelismo hardware, ottenendo prestazioni molto buone su sistemi con più core.

Precedentemente al lavoro di questa tesi ILDJIT non era in grado di eseguire codice che richiedesse il supporto alla programmazione generica (secondo le specifiche ECMA-335). Questo lavoro ha introdotto il supporto a questo paradigma ed esteso il supporto il

supporto di ILDJIT alla riflessività. Nel raggiungere questo obbiettivo è stato necessario sia riprogettare alcuni sottosistemi del compilatore sia per gestire nuovi tipi di dati che le nuove convenzioni di chiamata che la programmazione generica richiede. In particolare sono il lavoro si è concentrato sulla revisione di un sottosistema di ILDJIT responsabile della descrizione della semantica del linguaggio CIL, chiamato Libiljitmm. Il testo della tesi è strutturato nel seguente modo: nel Capitolo 2 vengono descritto lo stato d'arte. Vengono innanzitutto presentati le principali tecniche di compilazione adottate oggi: la compilazione statica, dove la fase di traduzione e separata dalla fase di esecuzione, e la compilazione dinamica, dove lo stesso compilatore fornisce l'ambiente di esecuzione per il programma. Sono presentati di seguito i paradigmi di programmazione riflessiva e generica mostrando come queste caratteristiche sono incluse nei principali linguaggi imperativi oggi disponibili come il C++, Java e ECMA-335. Il capitolo si conclude con una confronto tra i due principali ambienti virtuali oggi disponibili: ECMA-335 e la Java Virtual Machine.

Il Capitolo 3 è dedicato alla presentazione della caratteristiche proprie delle maggiori implementazioni del CLI evidenziando in particolare le particolarità del compilatori ILDJIT rispetto alle macchine virtuali concorrenti come Mono e DotGNU.

Il Capitolo 4 descrive le tecniche che i compilatori CLI possono utilizzare per la gestione delle caratteristiche avanzate. Sono introdotti i concetti di specializzazione, di condivisione e di condivisione parziale. Viene inoltre presentata la libreria per la gestione meta-dati, Libiljitmm (il modulo del compilatore che ha subito le maggiori modifiche), spiegando le modifiche apportate e i vantaggi ottenuti. Nel corso del capitolo sono affronti altri argomenti come la corretta gestione delle chiamate a funzione virtuali (chiamate indirette proprie del paradigma Object-Oriented) e la gestione delle strutture dati necessarie per funzionamento efficiente.

Il Capitolo 5 presenta il risultati dei test che sono stati eseguiti sul compilatore modi-

ficato e descrive la loro interpretazione. I test riguardano sia il confronto prestazionale tra le due versione di ILDJIT, con e senza supporto alla programmazione generica, sia il confronto con il principale competitor di ILDJIT, Mono.

Il lavoro si conclude con il Capitolo 6 il trae alcune conclusioni relativamente ai risultato ottenuti dal lavoro.

2 Stato dell'arte

Obbiettivo di questo capitolo è l'analisi de concetto della meta-programmazione, facendo in particolare riferimento al paradigma della programmazione generica nei linguaggi imperativi. Nel corso del capitolo vengono anche presenti i modelli di compilazione usati nei moderni compilatori.

2.1 Tecniche di Compilazione

Con processo di compilazione si intende l'operazione di traduzione un programma da un linguaggio di alto livello a una rappresentazione semanticamente equivalente a più basso livello, tipicamente in codice nativo per la macchina target. Questo problema viene affrontato dai moderni compilatore attraverso due approcci: *compilazione statica* e la *compilazione dinamica* [1].

2.1.1 Compilazione Statica

In un compilatore statico, detto anche compilatore Ahead-of-Time, il problema della traduzione viene eseguito completamente in un tempo, detto tempo di compilazione, precedente all'esecuzione del programma. Solitamente nella compilazione statica i file sorgenti che compongono il programma vengono tradotti indipendentemente l'uno dall'altro in file oggetto. Questo file contiene, il codice macchina, i dati usati a tempo di esecuzione e informazioni utili al debugging. All'interno del file oggetto tutti i riferimenti

a risorse sono locali; qualora occorra fare riferimento ad una risorsa esterna il compilatore assegna un simbolo che in una successiva fase sarà risolto da un altro strumento, chiamato *linker*, che ha il compito di raccogliere tutti i file oggetto e combinarli in un unico programma eseguibile risolvendo tutti i simboli dichiarati.

Il file così prodotto dalla compilazione statica è quindi pronto per essere eseguito sulla macchina per cui è stato compilato: per esempio non è possibile eseguire un programma compilato per ambiente Windows su una macchina Linux.

La Compilazioni statica nonostante la mancanza della portabilità del codice e della complessità di progettazione del compilatore presenta comunque una seria di vantaggi e in particolare:

- efficienza del codice prodotto: durante la fase di compilazione si può spendere tempo in ottimizzazioni aggressive senza incorrere in costi durante l'esecuzione.
- Il programmi non richiedono complessi ambienti di esecuzione.

2.1.2 Compilazione Dinamica

La compilazione dinamica affronta in maniere diversa il problema della compilazione. Il codice, a differenza della compilazione statica, viene tradotto durante l'esecuzione stessa dell'applicazione. La compilazione Dinamica comprende due fasi note come *Traduzione Dinamica* e l*'Ottimizzazione Dinamica*.

Durante la traduzione dinamica il programma viene tradotto verso un rappresentazione di più basso livello tipicamente partendo da una rappresentazione indipendente dalla piattaforma per cui si sta compilando e generando codice direttamente sulla macchina che esegue il programma. Questo tipo di traduzione permette di quindi di distribuire programmi che siano portabili e di dimensione inferiore, grazie alla maggior semantica del rappresentazione usata rispetto al codice nativo. La traduzione dinamica viene anche applicate a contesti diversi dalla compilazione pura: spesso questa strategia è usata per

permettere l'esecuzione di programmi scritti per architetture non più disponibili o per simulare un'architettura diversa da quella su cui il compilatore è in esecuzione.

Anche al codice generato da un compilatore dinamico vengono applicate ottimizzazioni con lo scopo di ridurre il tempo di esecuzione del programma. L'ottimizzazione Dinamica aggiunge ai vantaggi della ottimizzazione statica la possibilità di sfruttare informazione altrimenti non disponibile a tempo di compilazione, come constanti note solo a tempo di esecuzione; durante l'esecuzione è infatti possibile analizzare codice per identificare queste informazioni e applicare quindi ottimizzazioni che dipendono da esse.

Sfortunatamente mentre sull'ottimizzazione statica non sono imposti vincoli di tempo, l'ottimizzazione dinamica deve essere particolarmente accorta ed efficiente essendo eseguita durante l'esecuzione stessa del programma. Per questo motivo l'ottimizzazione dinamica non è effettuata su tutto il programma ,ma solo su punti specifici, tipicamente i metodi che sono eseguiti più di frequente.

La compilazione dinamica può essere implementata principalmente attraverso due tecniche: l'interpretazione e la compilazione *Just-in-Time*.

L'interpretazione prevede che eseguire direttamente il codice sorgente istruzione per istruzione. Gli interpreti sono semplici da costruire, assicurano l'indipendenza dalla piattaforma e la sicurezza grazie ad un ambiente di esecuzione controllato, ma hanno un consumo di risorse elevato. Per ogni istruzione si ha un tempo di decodifica e esecuzione normalmente più elevato di un processore reale, a causa della complessità delle istruzione, oltre a richiedere delle risorse che vanno sommate a quelle usate dal programma.

L'approccio Just-in-Time nasce per risolvere gli svantaggi dell'interpretazione pura dei programmi. Questo approccio prevede due fase.

1. Il programma sorgente viene staticamente compilato in un formato intermedio indipendente dalla macchina target, chiamato *Byte-code*, che mantiene molte delle informazioni disponibili del sorgente del programma sotto forma di meta-dati.

2. Il Byte-code viene caricato da una software, chiamato Macchina virtuale, che è capace di interpretare il programma e tradurlo nel codice nativo della macchina. Questo processo è eseguito a tempo di esecuzione ed applicato solo alle porzioni di programma, unità di compilazione, effettivamente necessarie durante l'esecuzione. La dimensione dell'unità di compilazione dipende delle caratteristiche proprie della macchina virtuale spaziando da una linea di codice ad un intera funzione o ad altro ancora.

I compilatori Just-in-Time possono garantire le stessa caratteristiche di indipendenza e sicurezza dagli interpreti, i controlli di sicurezza sono imposti dalla macchina virtuale durante la traduzione, ma offrono prestazioni migliori: quando un unità di compilazione è tradotta può essere memorizzata in una cache dalla macchina virtuale dove è disponibile il codice per le future esecuzioni, avvantaggiandosi così le proprietà di località temporale dei programmi. Questa proprietà può essere ulteriormente sfruttata dalla macchina virtuale ricompilando e ottimizzando le unità eseguite più di frequente.

Sfortunatamente un compilatore così strutturato è però complesso da sviluppare e richiede un attento bilanciamento tra il tempo spesso nell'ottimizzazione del codice e il tempo necessario la sua esecuzione.

2.2 Meta Programmazione

Con il termine meta-programmazione si intende la capacità di scrivere programmi che generino o manipolino altri programmi, o loro stessi, come parte di un flusso di compilazione o durante l'esecuzione dello stesso programma. Il linguaggio compositivo di un meta-programma viene definito *meta-linguaggio*, mentre il linguaggio del programma manipolato è definito *linguaggio oggetto*.

Il grande successo del principio di meta-programmazione ha portato ad una sua standardizzazione in numerosi linguaggi di programmazioni, indipendentemente dal paradig-

ma utilizzato; sono infatti disponibili sia linguaggi logici (es. Prolog), sia funzionali (es. haskell o Lisp), che imperativi (es. C#, JAVA e C++). I principali paradigmi di meta-programmazione sviluppati sono: la programmazione riflessiva e la programmazione generica.

Si parla di programmazione riflessiva, o più semplicemente di riflessività, quando il meta linguaggio e il linguaggio oggetto coincidono. La Riflessività introduce la possibilità di osservare, processo di *Introspezione*, o alterare, *intercessione*, una rappresentazione del proprio stato durante l'esecuzione, usando un meccanismo offerto dal linguaggio che prende il nome di *reificazione*. La caratteristica che differenzia la reificazione dalla semplice rappresentazione dello stato come *First-Class Object* sta nella nella connessione causale tra rappresentazione e linguaggio; quando lo stato viene rappresentato come First-Class Object, questo può utilizzato senza che abbia effetti sul linguaggio, mentre la reificazione introduce un meccanismo che lega alla modifica della rappresentazione una modifica del linguaggio. Nel esempio riportato nel Codice 1 è mostrata la capita di riflessione del linguaggio Perl. Durante l'esecuzione di questo piccolo frammento è definito dinamicamente il nuovo tipo `Foo`; questo è manipolato aggiungendo il metodo `Hello`. Dopo queste operazioni il metodo `Hello` è invocabile su un istanza dal tipo come mostrato nell'ultima riga dell'esempio.

Codice 1: Nell'esempio in figura durante l'esecuzione viene definito il nuovo tipo `Foo` a cui è aggiunto il metodo `Hello` che è infine invocato su un istanza del tipo.

```
my $class  = Foo;
my $method = hello;
my $object = $class->new();
$object->$method();
```

Parallelamente alla riflessività ha avuto successo il paradigma della programmazione generica. La Programmazione generica [10] è stata introdotta con il linguaggio Ada nel

1983, ma ha visto il suo maggior sviluppo con il rilascio nel 1993 del C++ Standard Template library.

Mentre la riflessività esplora l'interazione tra esecuzione del programma e i meccanismi di interpretazione del linguaggio, la programmazione generica si focalizza nella ricerca di strumenti per esprimere il polimorfismo parametrico. In particolare nella programmazione generica, il polimorfismo parametrico, consiste nella ricerca delle somiglianze tra diverse implementazione di uno stesso algoritmo allo scopo di sintetizzare un'unica rappresentazione più generale e astratta; questo processo e noto anche come *Lifting*. Il processo di lifting individua i requisiti minimi, categorizzati in insiemi detti *Concepts*, che i tipi di dati utilizzati dall'algoritmo devono soddisfare affinché questo possa operare in modo corretto ed efficiente. Un concept è un insieme di vincoli, quali l'associazione tra tipi o garanzie di complessità, tipicamente non esprimibili nei linguaggi di programmazione classici. Nel esempi di Codice 2 3 è presentata la trasformazione che il processo di lifting opera. Nel frammento 2 contiene un semplice calcolo della somma de numeri di un array implementato sia per un vettore di numeri interi che per numeri in virgola mobile. Applicando il processo di Lifting i due frammenti vengono sintetizzati in un unica funzione, codice 3, strutturalmente simile ai precedenti ma in cui in tutte le informazioni sui tipi delle variabili sono astratti da parametro di tipo `T`. Questa sostituzione ha comportato una sostanziale modifica del codice: nei codice originale inizializzazione della variabile `result` contenente il risultato poteva essere espressa attraverso una costante, essendo tutti i tipi noti, mentre dopo il processo di lifting occorre sostituire questa costante con una meccanismo che dipenda del parametro di tipo, nel caso del esempio si ipotizza un costruttore. Le sostituzioni impongono che il tipo rappresentato dal parametro `T` sia soggetto a delle restrizioni: deve supportare tutti gli operatori usati nel codice come somma, assegnamento. Questi requisiti compongo il Concept. Quando il Concept non è esprimibile formalmente nel linguaggio usato, come accade nell'esempio,

è sostituito da commenti allegati al codice.

Codice 2: Esempio di una algoritmo adatto all'applicazione del *Lifting*.

```
int sum(int* array, int n) {
  int result = 0;
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}

float sum(float* array, int n) {
  float result = 0;
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Codice 3: Esempio dell'applicazione del processo di *Lifting*.

```
// Requirements:
//    T must have a default constructor T() that produces the identity value
//    T must have an additive operator +
//    T must have an assignment operator
T sum(T array[], int n) {
  T result = T();
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Sui concept definiti è possibile operare processi di *Concept refinement* o *specializzazione* rispettivamente allo scopo di offrire una descrizione più ricca dell'astrazione oppure fornire requisiti per versioni più efficienti dell'algoritmo. Il risultato delle due operazione è quindi una gerarchia di concept così definita: un concept $C2$ rifines $C1$ se include tutti

i requisiti di $C1$ e ne aggiunge di nuovi. Nel esempio di Codice 4 il processo di Concept refinement è applicato al Codice 3. L'algoritmo è stato ulteriormente astratto sostituendo il tipo array della variabile di ingresso con il parametro di tipo `Container`. Questa parametro rappresenta un generico contenitore che permetta l'accesso tramite indice ai suoi elementi. La sostituzione ha comportato come mostrato nel esempio l'aggiunta di un nuovo commento al codice.

Codice 4: Esempio dell'applicazione del *Concept refinement*.

```
// Requirements:
//     T must have a default constructor T() that produces the identity value
//     T must have an additive operator +
//     T must have an assignment operator
//     Container must have an indexing operator [] that returns a T
T sum(Container array, int n) {
  T result = T();
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

L'utilizzo della programmazione generica rende possibile l'esplorazione del *concept-based overloading*. Questa caratteristica prevede che il linguaggio di programmazione abbia la capacità di scegliere autonomamente l'implementazione più specifica ed efficiente di un algoritmo, tra quelle disponibili, in base ai requisiti soddisfacibili.

2.3 La programmazione generica nei moderni linguaggi imperativi

Come detto, molti linguaggi, includono nativamente il supporto alla programmazione generica anche se i principali linguaggi forniscono solo parzialmente la caratteristiche

fino a qui descritte. Nella presente sezione viene dunque descritta la programmazione generica nei principali linguaggi imperativi. Per l'analisi sono stati scelti tre linguaggi: il C++ tra linguaggi compilati staticamente, Java come rappresentate di un linguaggio compilato dinamicamente e lo standard ECMA-335, conosciuto anche come .NET framework, come esempio tra le moderne macchine virtuali.

2.3.1 C++ Template

Il C++ è un linguaggio multi paradigma, procedurale e object-oriented, sviluppato a partire dalla fine degli anni'70 come estensione del linguaggio C con lo scopo di offrire un linguaggio general-purpose, efficiente e portabile che non richiedesse né caratteristiche proprie di specifiche piattaforme ne sofisticati ambienti di esecuzione. Caratteristica distintiva del C++ rispetto ad altri linguaggi è la presenza di un sistema, detto *Template*, che permette di sperimentare il paradigma della meta-programmazione.

I template [23] permettono di costruire schemi di classi o funzioni parametriche rispetto a tipi di dato o a costanti di compilazione, parametri che vengono usati dal compilatore per la generazione automatica del codice. Nel esempio presentato nel Codice 5 quando lo schema è istanziato per una costante, nel esempio il numero 4, il compilatore valuta il fattoriale a tempo di compilazione e sostituisce allo schema il risultato del calcolo.

L'uso dei template semplifica lo sviluppo di un programma. I template permettono la scelta della migliore implementazione di una strategia, non più definita a design time, ma decisa in modo automatico a tempo di compilazione. Questa decisione è effettuata valutando per ciascun call-site i parametri associati agli schemi e scegliendo il più adatto tra quelli in overloading secondo la regola *Substitution failure is not an error*: in tutti gli schemi candidati vengono sostituiti ai parametri formali gli argomenti dedotti; iterativamente vengono poi esclusi tutti i candidati che generano un errore di compilazione in modo da giungere alla elezione di un unico candidato. Usando questa

regola e l'espansione ricorsiva degli schemi si riesce a definire strutture di controllo complesse, come la scelta condizionata o i cicli, che rendono il template un linguaggio turing-completo.

Nonostante la potenza espressiva dei template, la programmazione generica non è completamente supportata; pur supportando la definizione di algoritmi o contenitori generici non esiste alcun costrutto che corrisponda completamente alla semantica dei concept. Per esempio è possibile memorizzare meta-informazioni come constanti o tipi di dato, meccanismo noto come *traits*, ma non è tuttavia possibile descrivere alcun tipo di vincolo associato ai parametri di uno schema. Questo comporta la mancanza di un sistema completo di type-checking per itemplate, sostituito dal type-checking classico del compilatore dopo l'espansione dei template.

Il C++ è un linguaggio compilato staticamente con lo scopo di generare codice efficiente e che non richieda sofisticati supporti a tempo di esecuzione. Questo ha comportato che i template fossero implementati secondo lo schema dei *Two-Level Languages*. Questo concetto prevede una netta distinzione tra codice statico, valutato a compile-time, e codice dinamico, valutato durante l'esecuzione; in C++ i template sono considerati codice statico che viene eseguito espandendo ogni istanza dello schema in una copia di codice dinamico. Il principale vantaggio di questo approccio risiede nell'efficienza del codice, che non richiede supporto a tempo di esecuzione. Come effetti collaterali si hannno sia l'aumento del tempo necessario per la compilazione che l'aumento della dimensione finale del codice, causati dalla mancanza di meccanismi per la verifica della compatibilità tra le copie di codice generate.

2.3.2 .NET Framwork

Il .NET Framework è un infrastruttura software sviluppata da *Microsoft* alla fine del 1990. Questa infrastruttura offre allo sviluppatore numerose librerie con soluzione pre-

Codice 5: Un tipico esempio di template. A tempo di compilazione il template verrà espanso un una struttura contenente il reale valore del fattoriale.

```
// Induction

template <int N> struct Factorial {
  static const int value = N * Factorial<N - 1>::value;
};

// Base cases via template specialization:

template <>
struct Factorial<0> {
  static const int value = 1;
};

template <>
struct Factorial<1> {
  static const int value = 1;
};

// Factorial<4>::value == 24
```

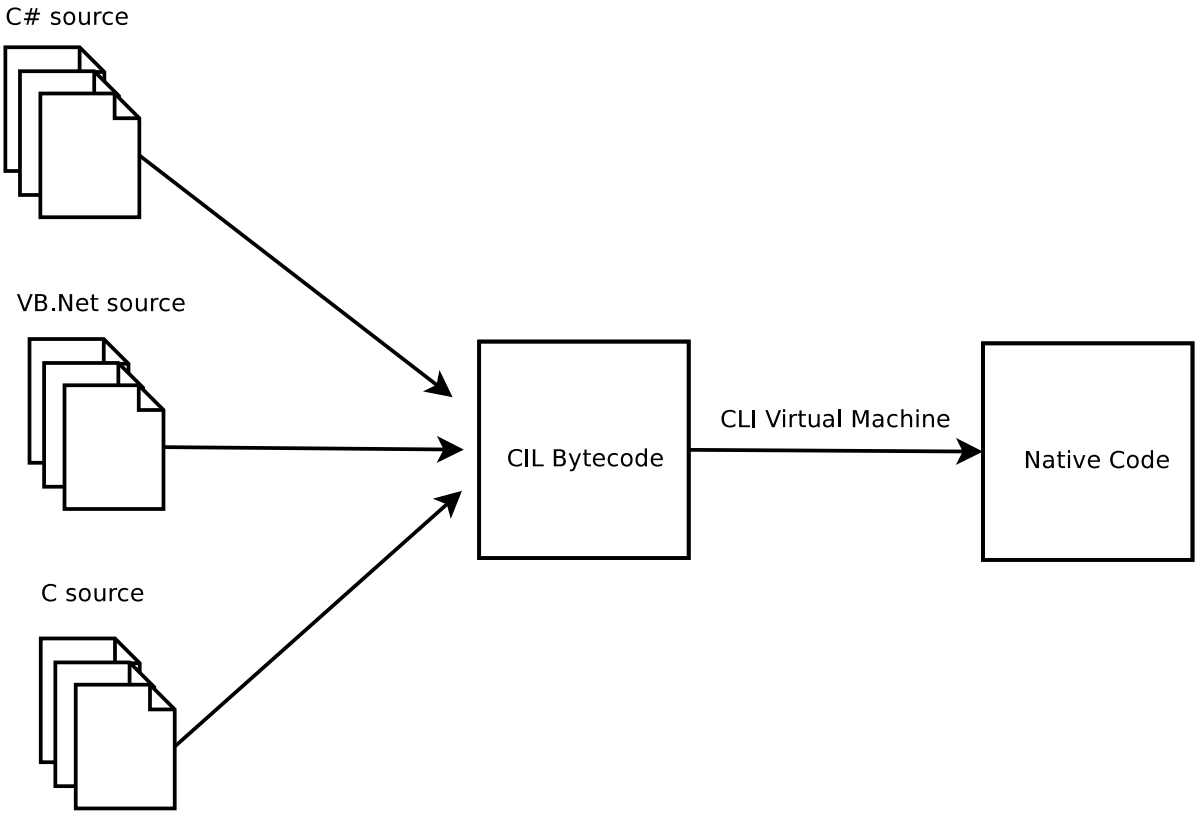
sviluppate e un ambiente di esecuzione per i programmi scritti appositamente per il Framework. La specifica di tale ambiente è descritta nel *Common Language Infrastructure* (CLI) definito dalla stessa Microsoft successivamente standardizzato e pubblicato come standard ECMA-335 [14]. L'ambiente descritto dallo standard, mostrato in figura 2.1 permette di utilizzare diversi linguaggi di alto livello per produrre programmi che siano eseguibili su diverse piattaforme senza la necessità di essere riscritti per una specifica architettura. il CLI è stato infatti progetta progettato per offrire le seguenti caratteristiche:

- Indipendenza dalla piattaforma reale di esecuzione.
- Indipendenza dal linguaggi di alto livello utilizzato.
- Type-safety e verificabilità statica del codice eseguito.
- Efficienza del codice generato.

Il nucleo del CLI è costituito dal sistema di tipi, detto Common Type System (CTS), che viene condiviso tra CLI e tutti gli strumenti che interagiscono con esso creando i presupposti per l'integrazione di applicazioni multi-linguaggio. Il CTS è composto attorno all'idea di contratto cioè la specifica dei requisti che implementazione di un costrutto del linguaggio deve rispettare.

La condivisione delle informazione tra tutti i componenti è realizzata con il meccanismo dei meta-dati, che memorizzando in maniera persistente i riferimenti e le descrizione dei contratti. L'ambiente di esecuzione specificato dal CLI prende il nome di Virtual Execution System (VES). Il VES è macchina astratta stack-based, che costruita attorno l'instruction set *Common Intermediate Language* (CIL) definito dal CLI stesso. Il compito del VES è di caricare ed eseguire il codice e di assistere il programma fornendo servizi di gestione delle memoria e della concorrenza.

Figura 2.1: Il *Common Language Infrastructure*.



Rispetto ad altre infrastrutture per la compilazione dinamica il CLI offre un sistema di tipi particolarmente ricco e complesso capace di supportare efficientemente numerosi paradigmi quali l'Object-Oriented, il funzionale o il procedurale. In particolare il CTS supporta sia la semantica dei linguaggi value-oriented, attraverso i Tipi Valore, che la semantica dei linguaggi Object-Oriented, attraverso il Tipi Riferimento:

Tipi Valore contengono direttamente i dati (es Interi o Floating-Point) e le istanze vengono allocate in stack o all'interno di oggetti. CLI ammette come tipi valore sia i tipi primitivi, forniti dal VES, che tipi definiti dal progettista. La Compatibilità tra due entità di tipi valore è determinata dalla compatibilità delle rappresentazioni. La definizione di un tipo valore avviene tramite la definizione di una classe.

Tipi Riferimento memorizzano un riferimento ad un'istanza del tipo solitamente allocato nello Heap. Per questi tipi la compatibilità dipende dalla funzionalità offerta. Il CLI suddivide i tipi riferimento in due categorie: i tipi oggetto e i tipi interfaccia. i tipi oggetto definiscono sia la struttura delle istanze del tipo che le operazioni permesse, mentre i tipi interfaccia non offrono informazioni relativamente alla struttura interna di tipo. Il CTS classifica gli Array come un particolare tipo oggetto.

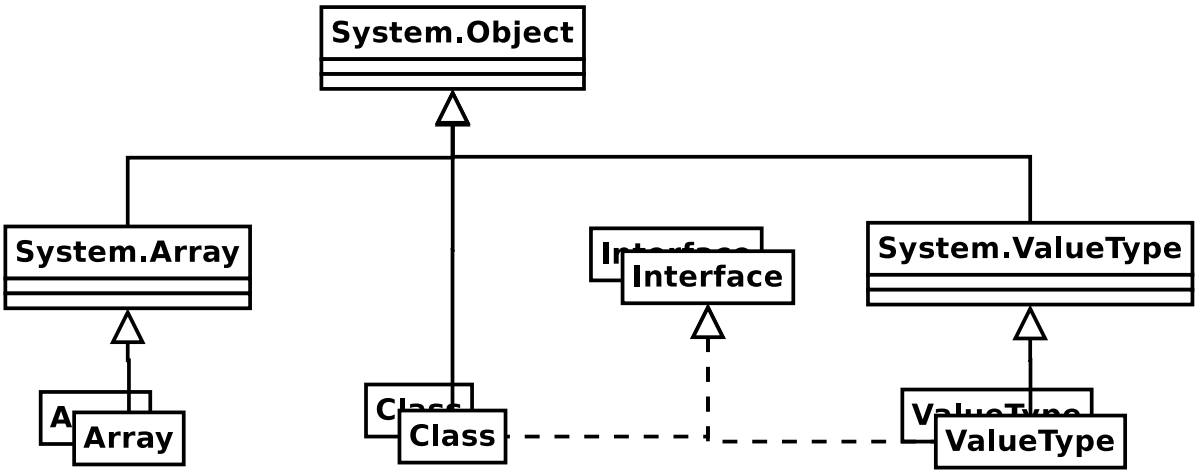
Ad eccezione dei tipi interfaccia, che non necessitano di una descrizione completa, e degli array, il cui tipo viene creato e gestito automaticamente dal VES, la definizione di un nuovo tipo avviene attraverso l'uso di classi e delle catene di ereditarietà. Devono inoltre essere rispettate le seguenti regole:

- tutti i tipi devono estendere in ultimo il tipo *System.Object* fornito dal VES.
- Tutti i Tipi valore devono ereditare da *System.ValueType* e non possono essere ulteriormente estese. Questa scelta permette di semplificare la gestione dello stack su cui le istanze dei tipi valore vengono memorizzate.

- Tutti i Tipi array estendono la Classe *System.Array*. Questa regola viene, come detto in precedenza, automaticamente imposta dal VES.
- Non è possibile l'ereditarietà multipla tra classi.

Il sistema di tipi risulta quindi unificato come si può vedere in figura 2.2.

Figura 2.2: Struttura del *Common Type System*



L'ereditarietà garantisce che il codice generato per funzionare su istanze di un tipo continui a funzionare anche quando sono passate istanze di un tipo derivato. Per garantire questa regola e permettere l'interoperabilità tra Tipi Valore e Tipi riferimento il CLI definisce due conversioni dette rispettivamente *Boxing* e *UnBoxing*. Concettualmente le due operazione corrispondo rispettivamente al casting di un tipo valore verso *System.Object* e viceversa. L'operazione di boxing converte un istanza di tipo valore in un istanza di tipo riferimento mantenendo la stessa struttura fisica. L'unboxing invece converte un istanza riferimento, il cui tipo dinamico è un tipo valore, in un istanze di tipo valore.

Il CTS supporta nativamente il polimorfismo parametrico, attraverso la definizione di definire tipi di dati e algoritmi parametrici, detti *Generics*. I Generics sono una

estensione naturale e uniforme del CTS, fino a qui descritto, e sono stati progettati per rispettare i seguenti requisiti:

Ortogonalità i tipi generici possono essere utilizzati in qualunque contesto dove i tipi CTS sono permessi. Sono infatti permessi sia tipi valore che tipi riferimento generici.

Indipendenza dal linguaggio non sono fatte assunzioni sul linguaggio sorgente usato. Per questo motivo i Generics del CLI devono garantire una buona espressività limitata solamente al Type-Safety che il CLI impone. Questa restrizione non permette di supportare completamente alcuni meccanismi, come la varianza dei tipi generici, tipica dei linguaggi derivati da Eiffel.

Indipendenza dall'implementazione l'implementazione del CLI è lasciata al progettista della macchina virtuale.

Efficienza le operazioni su tipi generici non devono fornire prestazione inferiori rispetto alle analoghe su tipi regolari.

Il Sistema dei Metadati

Tutti i contratti del CTS vengono memorizzate attraverso il sistema dei meta-dati. Il CLI definisce si lo schema logico che il formato fisico con cui i dati vengono salvati. Per ciascun contratto non gestito dal VES i compilatori memorizzano la completa descrizione attraverso i meta-dati; queste informazione sono accessibili sia dal VES che al programma in esecuzione attraverso la libreria di Riflessitiva. La riflessività permette al programma di navigare attraverso le descrizione dei contratti ottenendo informazione su di essi. Il sistema dei metadati è utilizzato anche dalle istruzione CIL. Attraverso simboli che accompagnano le istruzioni, noti *metadata tokens*, è determinanata la semantica delle operazioni senza dover conoscere il reale layout degli oggetti che manipolano.

ECMA-335 Generics

Il polimorfismo parametrico in ECMA-335 prevede la possibilità di definire famiglie di tipi o metodi caratterizzati dalla presenza di uno o più parametri formali di tipo detti *generic parameter*. Il tipo attuale usato per istanziare un tipo o un metodo generico è definito *generic argument*. Il CLI associa a ciascun generic parameter un nome e un insieme, eventualmente vuoto, di vincoli che un generic argument deve rispettare. Questi vincoli sono divisi in due categorie:

Vincoli di tipo richiede che generic argument estenda un altro tipo.

Vincoli speciali richiede che generic argument appartenga a una famiglia dei tipi valore o dei tipi riferimento. Un tipo generico deve comunque rispettare devi vincoli imposti dal CLI a priori: in particolare non sono infatti consentiti come generic argument ne i tipi di dati riservati al VES ne i tipi, come i *ByRef*, il cui uso è consentito solo in certi costrutti.

Molti linguaggi ad alto livello permettono di esprimere la varianza dei parametri formali. Per mantenere la compatibilità con questi linguaggi il CLI permette specificare la proprietà di varianza per i generic parameter; questi sono generalmente non varianti ma sono comunque permesse la proprietà di covarianza e la proprietà di contro-varianza. La type-safety impone ai generic parameter varianti le seguenti restrizioni:

- La proprietà di covarianza è consentita solo per i generic parameter assegnati a punti in cui un tipo è prodotto (es. come valore di ritorno di un metodo).
- La proprietà di contro-varianza è consentita solo per i generic parameter assegnati a punti in cui il tipo è consumato (es. come tipo per un parametro di ingresso di un metodo).

Attraverso il sistema dei meta-dati viene memorizzato solo il contratto relativo alla definizione dei Generics mentre attraverso i metadata tokens è possibile indirizzare sia la definizione che la una singola istanza del tipo.

Il supporto ai Generics nel CLI è stato progettato in modo da semplificare la logica dei compilatori per l'architettura CIL. Quasi tutte istruzioni le CIL sono in grado di operare indifferentemente sia su tipi valore chee su tipi riferimento, garantendo una visione uniforme sul sistema di tipi. Questa caratteristica permette al compilatore di generare codice generico di trascurare la categoria di un generic argument e lasciarne la gestione al VES a tutto vantaggio della dimensione e della efficienza del codice.

2.3.3 Java

Java [11] è un linguaggio di programmazione classificato come general-purpose, concorrente, object-oriented progettato per essere eseguito attraverso *Java Virtual Machine* (JVM). Lo sviluppo di Java è caratterizzato dalla ricerca di un linguaggio semplice ma dotato di buona espressività; queste peculiarità sono ben visibili nel sistema di tipi per esempio. Java definisce solo due categorie di tipi: i tipi primitivi, corrispondenti ai tipi forniti dalla JVM, e i tipi riferimento. A questa ultima categoria appartengo i tipi definiti dallo sviluppatore attraverso il meccanismo delle classi o delle interfacce.

Dalla versione 5 il linguaggio ha incluso il supporto al polimorfismo parametrico, noto con il nome di Generics, come una estensione del sistema di tipi.

Java Generics

I Java Generics sono schemi di tipi riferimento che possiedono uno o più parametri formali di tipo, detti *type variable*. I Generics non sono tipi concreti ma occorre che siano istanziati sostituendo ai parametri formali di tipo gli argomenti, ottenendo così un

tipo parametrico. Un tipo parametrico è assimilabile ad un tipo concreto ma è soggetto alle seguenti restrizioni:

- Non può essere usato nella creazione di un array.
- Non può essere usato nella gestione delle eccezioni.
- Non può essere usato come letterale.
- Non è consentito l'uso in espressione che richiedono il controllo del tipo dinamico di un oggetto.
- Non è possibile usare tipi primitivi come parametri di tipo. I tipi primitivi devono essere sostituiti dai loro corrispettivi boxed type.

I limiti descritti sono strettamente legati alla strategia di gestione dei Generics che i sviluppatori del linguaggio hanno deciso di adottare. In Java i Generics esistono solo come costrutti del linguaggio e non sono reificati dall'ambiente di esecuzione. Un tipo è detto reificabile se a tempo di esecuzioni sono disponibili informazioni complete sul tipo. Negli esempio riportati nel codice 6 e nel Codice 7 si può vedere come il frammento prima della traduzione, Codice 6, venga trasformato dal compilatore nel Codice 7 prima di essere tradotto in bytecode. Dopo questa fase ogni riferimento ai parametri di tipo è cancellato. La mancanza di delle informazioni sui parametri impedisce all'ambiente di esecuzione di garantire la correttezza e la type-safety di alcune operazione e di conseguenza limita l'espressività dei Generics in Java.

Il processo trasformazione spiegato prende il nome di *type erasure*. Durante la trasformazione compilatore mappa i tipi parametrici in un unica rappresentazione chiamata *raw type*. Il processo è ottenuto in tre fasi:

Eliminazione dei Parametri di tipo questa fase rimuove ogni riferimento ai parametri di tipo sostituendoli con il tipo *Object*.

Eliminazione degli argomenti di tipo ogni tipo parametrico usato viene rimosso e sostituito con il corrispondente raw type.

Inserimento dei controlli di cast sono inseriti controlli di Cast espliciti nei punti di ritorno delle chiamate a garanzia della type safety.

Codice 6: Un esempio di una definizione di una classe generica in Java.

```
interface Comparable <T> {
 public int compareTo(T that);
}

class Value<T> implements Comparable<Value<T>>{
 private T value;

public Value (T value) {
  this.value = value;
 }
 public T getValue() {
  return value;
 }
 public int compareTo(T that) {
  return this.value.equals(that.value);
 }
}
```

2.3.4 Java Generics Vs. ECMA-335 Generics

Le scelte operate dai due progetti nell'ambito del supporto ai Generics sono molto diverse: l'introduzione dei Generics in java è stata realizzato con l'obbiettivo di garantire la *migration compatibility*, ossia la completa interoperabilità con il codice prodotto anche con versioni precedenti di Java. Lo standard ECMA-335 ha introdotto i Generics con il solo supporto alla *backward compatibility*, ossia offrendo la possibilità di eseguire

Codice 7: L'esempio mostra il Codice 6 dopo l'applicazione della type erasure

```
interface Comparable {
 public int compareTo(Object that);
}

class Value implements Comparable{
 private Object value;

public Value (Object value) {
  this.value = value;
 }
 public Object getValue() {
  return value;
 }
 public int compareTo(Object that) {
  return this.value.equals(that.value);
 }
}
```

codice precedente sulle nuove piattaforme, ma non il viceversa. Il risultato ha portato a differenze apprezzabili nella espressività che sono sintetizzate nella tabella sottostante:

Tabella 2.1: Confronto tra i Java Generics e i Generics ECMA-335

	Java Generics	**ECMA-335 Generics**
Type Check	La type erasure introduce dei controlli e cast eseguiti a tempo d'esecuzione a svantaggio delle prestazioni	La type safety è garantita senza la necessità di inserire ulteriorri controlli o cast a tempo di esecuzione
Parametri di tipo	sono consentiti solo tipi riferimento	Sono consentiti tutti i tipi. L'uso dei tipi primitivi non richiede conversione boxing/unboxing

Tabella 2.1: Confronto tra i Java Generics e i Generics ECMA-335

	Java Generics	**ECMA-335 Generics**
Supporto alle eccezioni	Le eccezioni generiche non sono supportate	le eccezioni generiche sono supportate
Membri statici	non sono supportati membri statici per i tipi parametrici. Durante la type erasure tutte le istanze sono mappate su un unica classe.	Ogni realizzazione di una tipo generico è un tipo concreto che può quindi definire membri statici.
Riflessività	sono necessarie estensione alla libreria di riflessività per ottenere le informazioni complete sui tipi parametrici	Le realizzazione di tipi generici sono concrete e preservano tutte le informazioni a tempo di esecuzione.

2.3.5 Java Virtual Machine Vs. ECMA-335 Virtual Machine

La Java Virtual Machine [17] e lo standard ECMA-335 [9] [7] sono de-facto gli standard per l'ambienti di esecuzione virtuale. Entrambe le infrastrutture sono caratterizzate dall'adozione di macchine a stack programmate attraverso un ricco instruction set. A differenza delle cpu reali, questi ambienti contengono istruzioni con semantiche più ricche, ad esempio istruzioni per l'accesso o la manipolazione di oggetti; queste istruzioni sono caratterizzate dal fatto che il codice in esecuzione non dispone delle informazioni sul reale layout degli oggetti che manipola ma bensì usa riferimento ad informazione simboliche sulla loro struttura.

Nonostante l'apparente similarità tra le due infrastrutture esistono delle profonde differenze progettuali riportate nella tabella sottostante:

Tabella 2.2: Confronto tra la Java Virtual Machine e la macchina virtuale ECMA-335.

	Java Virtual Machine	**ECMA-335**
Sistema di Tipi	Non-Unificato.	Unificato
Supporto multi linguaggio	Progettata per come target il solo linguaggio Java.	Progettata espressamente per l'indipendenza dai linguaggi di alto livello
Supporto alle *tail call*	Non previsto dalla JVM	é parte dello standard. Le tail call permettono di per supportare efficientemente i linguaggi funzionali.
Supporto ai linguaggi Dinamici	Non realizzabile. Non è possibile implementare tecniche di dispaching differenti da quelle già previste dalla JVM	é possibile implementare meccanismi di dispaching arbitrari usando un sotto insieme apposito dell'instruction set.
Packing del codice	Ogni classe è distribuita a se stante. Il supporto agli archivi di classi stato introdotto a posteriori	il formato per la distribuzione comprende caratteristiche avanzate come gestione della versioning e gestione dell'internazionalizzazione

Tabella 2.2: Confronto tra la Java Virtual Machine e la macchina virtuale ECMA-335.

	Java Virtual Machine	**ECMA-335**
Interoperabilità con il codice nativo	Solo attraverso un interfaccia proprietaria Java Native Interface	Supporto le invocazioni a codice nativo attraverso un livello di compatibilità binaria.
Verificabilità del codice	Complessa richiede analisi DataFlow	Ogni istruzione CIL specifica un semplice insieme di regole.
Supporto alla riflessività	é necessario l'intervento dei servizi offerti dall'ambiente di esecuzione	é possibile gestire la riflessività direttamente nel codice CIL grazie ad apposito un set di istruzioni.

3 ECMA-335 Virtual Machine

Questo capitolo introduce lo stato dell'arte delle macchine virtuali che implementano lo standard ECMA-335, rivolgendo una particolare attenzione verso il progetto ILDJIT scelto come infrastruttura implementativa per questo lavoro di tesi.

3.1 Common Language Runtime

Il Common Language Runtime è il componente centrale dell'infrastruttura .NET di Microsoft nonché l'implementazione di riferimento dello standard CLI. Sul Common Language Runtime esistono solo poche informazioni data la natura closed source del prodotto. Il VES implementa un compilatore Just-in-Time a cui si affianca il servizio *Native Image Generator*(NGEN) per la generazione di codice Ahead-of-Time. L'unico sistema supportato dal Common Language Runtime è Windows.

3.2 DotGNU

L'iniziativa DotGNU è nata con lo scopo di fornire un sostituto open source al Common Language Runtime. All'interno di questa iniziativa sono stati sviluppati diversi progetti:

Portable.NET ha come obiettivo l'implementazione completa dello standard ECMA-335. Oltre ad un ambiente di esecuzione, questo progetto ha realizzato una

Base Class Library, compatibile con la quella standard offerta dalla soluzione di Microsoft, e un compilatore C#.

DotGNU Execution Environment è un server per la gestione dei webservice progettati su piattaforma .NET.

libJIT è una libreria riutilizzabile che implementa un compilatore Just-in-Time. Questa libreria è stata progettata per offrire un generico motore JIT indipendente da un particolare byte-code. Attualmente libJIT è usata come backend sia dal progetto Portable.NET che dal progetto ILDJIT.

Lo sviluppo di DotGNU ha subito un forte rallentamento a partire dal 2007, anno in cui è stato rilasciata l'ultima versione stabile. Attualmente alcune parti dallo standard CLI, come i generics, non sono state ancora completate.

3.3 Mono

Mono è la maggiore piattaforma software open source compatibile con il .NET Framework. Nato nel 2001, ha visto il suo primo rilascio stabile nel 2004 con Mono 1.0. Questo è stato possibile grazie agli sforzi congiunti sia della comunità open source che della casa software Novell. L'attuale versione stabile di mono, 2.6.4, è composta da un impressionante numero di moduli e componenti che garantiscono la quasi completa compatibilità con l'infrastruttura di CLR; tra questi i principali sono:

C# Compiler è un compilatore C# compatibile con i profili 1.0, 2.0 (standardizzato dall'ECMA), e parzialmente con C# 3.0 ..

Mono Runtime è una completa implementazione dello standard CLI. L'ambiente d'esecuzione prevede sia la compilazione Just-in-Time che quella Ahead-of-Time.

Base Class Library fornisce una libreria di classi compatibile con il Base Class Library del CLR.

Mono Class Library è un insieme di classi che estende le funzionalità della Base Class Library. La Mono Class Libary include molte funzionalità che aiutano lo sviluppo di applicazioni in ambiente Linux.

Lo sviluppo di Mono è stato caratterizzato da numerose rivoluzioni localizzate soprattutto nel Mono Runtime. Con l'introduzione nell' ottobre 2008 della seconda major release, è stato completamente riprogettato il motore Just-in-time per sostituire la vecchia rappresentazione intermedia basata su alberi con una più moderna rappresentazione Single-Static-Assignment. Nello stesso anno fu riprogettata anche la gestione dei generics, introducendo per la prima volta la tecnica di condivisione parziale e, parallelamente, venne rilasciata una prima versione del compilatore Ahead-of-Time il cui sviluppo è ad oggi tuttora in corso. L'ultima versione di Mono ha introdotto un nuovo backend Just-in-Time, basato sul progetto LLVM, con lo scopo di aumentare l'efficienza del codice nelle applicazioni *computationally intensive*.

Dato il numero di piattaforme (X86, Arm, Sparc ecc..), di sistemi operativi supportati (Windows, Linux, Mac Os X, Solaris, ecc..) e il livello di compatibilità con il CLR, Mono, ad oggi, è da considerasi l'implementazione di riferimento per il .NET Framework nel mondo opensource.

3.4 ILDJIT

ILDJIT, acronimo di *Intermediate Language Distributed Just In Time* , è un compilatore sviluppato dal *Compiler Group* [5] presso il *Politecnico di Milano*. Il suo sviluppo è iniziato nel 2005 da *Simone Campanoni* che ad oggi è il leader del progetto e sviluppatore principale. ILDJIT è un ambiente compatibile con il .NET framework che include un

implementazione del CLI, sviluppata internamente al progetto ILDJIT, e la Base Class Library del progetto *DotGnu*.

Lo scopo del progetto ILDJIT è lo studio di una nuova generazione di VES per ambienti distribuiti e paralleli capace di adattarsi velocemente alle caratteristiche delle future piattaforme di calcolo. Gli obbiettivi di ILDJIT si possono quindi riassumere nella ricerca dell'adattabilità e modularità senza compromettere le performance.

3.4.1 Architetture del Software

ILDJIT è un software modulare [3]. I moduli sono divisi in due gruppi: il gruppo principale include il Pipeliner, il CLI Manager, l'Ottimizzatore, il Garbage Collector e la Macchina virtuale IR, mentre il secondo gruppo comprende i moduli di supporto all'infrastruttura quali il Profiler, il sistema di inizializzazione ed altre strutture dati e librerie di supporto. Nel dettaglio:

Pipeliner gestisce la pipeline software necessaria per la traduzione, l'ottimizzazione e l'esecuzione del codice CIL.

CLI Manager implementa l'architettura descritta dal CLI e provvede alla traduzione del codice CIL nella rappresentazione intermedia interna a ILDJIT

Macchina virtuale IR ha il compito di tradurre la rappresentazione interna in codice macchina semanticamente equivalente.

Profiler raccoglie statistiche sulle performance sia dei moduli di ILDJIT che sul codice generato.

L'implementazione di ciascun modulo può seguire vari approcci a seconda dell'utilizzo tipo: i moduli che richiedono la maggiore flessibilità sono implementati attraverso plugins (librerie condivise caricate dinamicamente) permettendo di caricare, cambiare

e rimuovere moduli anche durante l'esecuzione. Per i moduli che hanno una vita utile più lunga ma sono utilizzati da più sottosistemi si è scelto l'approccio della libreria condivisa caricata staticamente; questo approccio permette di ottenere un buon grado di flessibilità ad un costo inferiore rispetto al caricamento dinamico. Tutti i componenti che invece richiedono la massima efficienza sono stati progettati come moduli interni e non sono condivisi tra i sottosistemi.

3.4.2 Modello di Compilazione

I compilatori Just-in-Time tradizionali adottano una politica di compilazione *lazy*: quando il flusso di controllo determina l'invocazione di un metodo non ancora pronto, l'applicazione viene sospesa finché questo non è tradotto. ILDJIT applica invece una politica diversa:

1. Il pipeliner, mostrato in Figura 3.1, è progettato per avvantaggiarsi del parallelismo derivante sia dalla compilazione ed esecuzione simultanea del codice che dal parallelismo intrinseco delle fasi di compilazione e di ottimizzazione; tutti gli stadi della pipeline software sono quindi eseguiti in parallelo e a ciascuno stadio sono assegnati più thread. Il pipeliner prende in ingresso un metodo CIL da tradurre ed in base allo stato di traduzione del metodo, il pipeliner sceglie in quale dei cinque stadi della pipeline inserirlo. Un metodo nello stato CILCODE deve ancora essere tradotto nella IR ed è inserito all'ingresso del pipeliner; al termine del primo stadio il codice è in formato IR (stato IRCODE) ed è in attesa dell'ottimizzazione ad alto livello. Quando il metodo è ottimizzato è pronto per essere tradotto in codice macchina, entrando così nello stato (MACHINECODE), ma tuttavia prima di poter essere eseguito occorre risolvere tutte le dipendenze di memoria statica del codice; il metodo quindi entra nel quarto stadio della pipeline che inizializza la

memoria statica necessaria portando il metodo nello stato (EXECUTABLE). Da questo momento il metodo è pronto per andare in esecuzione.

2. ILDJIT implementa una tecnica nota come *Dynamic Lookahead compilation* per evitare stalli dovuti alla compilazione durante l'esecuzione del programma. Questa tecnica stima probabilità che un metodo m_i sia invocato durante l'esecuzione del metodo m_j attraverso la misura delle distanza tra le due unità sul grafo delle chiamate; tutte le unità per cui la distanza dall'attuale unità in esecuzione è inferiore ad una soglia D sono buone candidate e sono quindi avviate alla traduzione. ILDJIT osserva lo stato di esecuzione del programma, come riportato in figura 3.2, e calcola il *lookahead boundary*, ossia la porzione del callgraph che include i prossimi metodi candidati alla traduzione. Il lookahead boundary è un insieme dinamico il cui contenuto dipende sia dal parametro di soglia impostato che dal flusso di esecuzione del programma. In figura 3.2 si può vedere come il lookahead boundary sia formato dai metodi m1, m2, m3 durante l'esecuzione del metodo main mentre quando l'esecuzione arriva al metodo m5 l'insieme contiene solo i metodo m6 e m7.

Per garantire buone performance occorre minimizzare l'overhead sul sistema. ILDJIT implementa quindi una strategia adattativa per impostare le soglie di attivazione e il numero di thread attivi.

3.4.3 Macchina virtuale IR

La Macchina virtuale IR è uno di componenti più complesso dell'intero VES. Il compito delle macchina virtuale è gestire la traduzione da IR a codice macchina e la successiva fase di esecuzione rispondendo alle richieste del pipeliner. Data la complessità di questo sottosistema si è scelto di progettare la macchina virtuale attraverso il design pattern della delega: durante l'esecuzione il componente principale di questo sottosistema orche-

Figura 3.1: Il diagramma mostra il modello della pipeline di compilazione: sulla sinistra sono mostrati gli stadi della traduzione mentre sulla destra i componenti dell'architettura interessati. Nei box è mostrato lo stato di traduzione del codice al ingresso di ogni stadio.

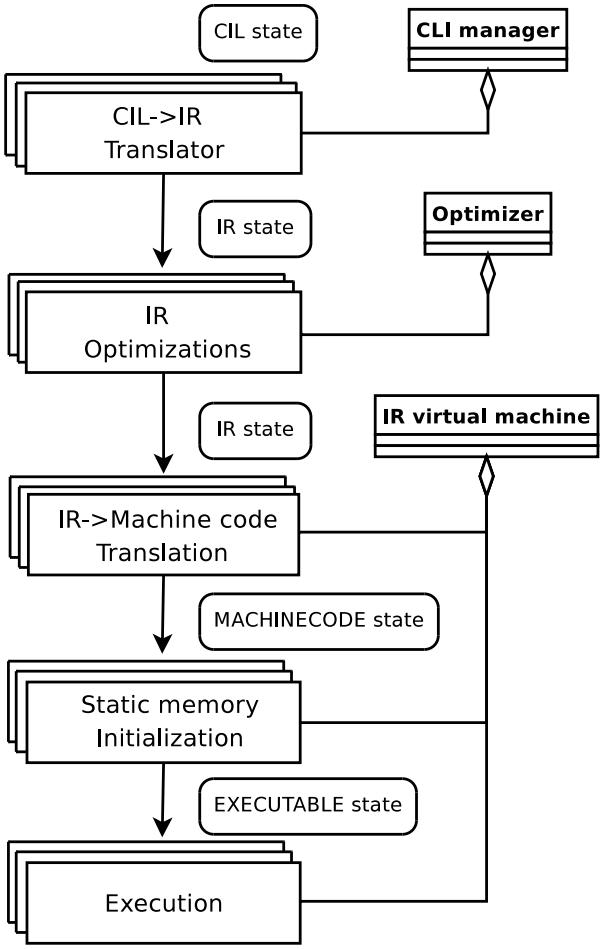
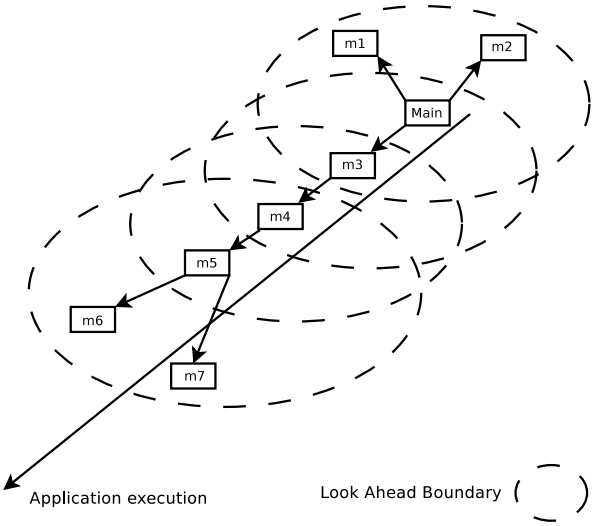


Figura 3.2: L'immagine mostra l'adattamento dalla lookahead boundary durante l'esecuzione del programma. L'ordine di esecuzione del programma è main, m3, m4, m5.



stra l'esecuzione degli altri sotto-moduli responsabili della gestione delle eccezioni, dei thread,ogni oltre che della traduzione ed esecuzione vera e propria della IR.

La rappresentazione intermedia si basa su un macchina a registri programmata da un linguaggio assembly RISC-like. Il Linguaggio IR definisce il propri tipi di dati, identificati dalla tripla $\langle valore, tipoformale, tipointerno \rangle$. I dati possono essere delle costanti, tipi semplici (es. interi, virgola mobile) o tipi complessi (riferimenti a classi, metodi, campi ecc) la cui dimensione può corrispondere sia alla dimensione nativa della piattaforma, soprattutto per i tipi semplici, o essere indipendente da quella della macchina.

IR definisce sette gruppi di istruzioni:

- Istruzioni aritmetiche (e.g `ADD`, `SUB`).
- Istruzione per l'accesso alla memoria (e.g `LOAD`, `STORE`, `GETADDRESS`).
- istruzione per l'allocazione degli oggetti (e.g `NEWOBJECT`, `NEWARRAY`).
- Istruzioni di invocazione a funzione (e.g `CALL`, `INDIRECTCALL`).

- Istruzione per comparazione (e.g `GT`, `LT`, `EQ`).
- Istruzione di salto (e.g `BRANCH`, `BRANCHIF`).
- istruzione per la gestione delle eccezioni (e.g `THROW`).

La istruzione aritmetiche sono ulteriormente divise in due categorie, a seconda che notifichino o meno l'overflow. Questa divisione è essenziale per garantire la gestione di eventuali side-effects. Le istruzioni per per la gestione degli oggetti e della memoria, normalmente non presenti nella rappresentazione intermedia, sono state inserite per permettere l'implementazione di analisi e ottimizzazioni aggressive. Infine anche le istruzioni per la gestione delle eccezione sono divise in due sotto-categorie: la prima categoria comprende le istruzioni che generano le eccezioni, mentre la seconda raccoglie tutte le istruzioni per la cattura delle stesse. Introducendo queste informazioni nell'IR ILDJIT può riconoscere le parti di codice che gestiscono le eccezioni non ottimizzandole in quanto eseguite raramente.

3.4.4 Ottimizzazione del codice

L'ottimizzazione del codice è una fase critica del processo di compilazione sopratutto in un compilatore dinamico: non è infatti garantito che l'applicazione di una trasformazione sul codice porti a benefici al metodo, anzi è possibile che le prestazione degradino. Alcune ottimizzazione come il loop interchange o il loop unrolling tentano di migliorare le performance riorganizzando o trasformando il codice del corpo del ciclo. I risultati di queste trasformazioni generano spesso un aumento delle dimensioni del codice, nel loop unrolling, o una riorganizzazione degli accessi a memoria, nel loop interchange, che possono degradare le prestazioni delle cache. Occorre quindi bilanciare i benefici che l'ottimizzazione del codice può portare con i costi, in termini di tempo di compilazione, per realizzarla.

ILDJIT applica ottimizzazione sia a livello di IR che a livello di codice macchina. L'applicazione di un'ottimizzazione ad alto livello risulta infatti semplice da applicare grazie alle informazioni sulla semantica del programma disponibili. Tuttavia molte trasformazioni richiedono informazioni sul tipo di macchina e quindi devono essere realizzate o completamente o parzialmente a livello di codice nativo.

Il Trade-off costo-benefici richiede che il compilatore fornisce un infrastruttura specifica per applicare la migliore strategia di ottimizzazione; ILDJIT ha scelto quindi di definire quattro categorie di plugins:

Interfaccia IR fornisce ai plugins di ottimizzazione gli strumenti per manipolare la IR aggiungendo, rimuovendo o modificando istruzioni.

Interfaccia di profiling Questa interfaccia permette di raccogliere informazioni, quali il grafo delle chiamate o statistiche sul valore dei parametri, durante l'esecuzione di un programma controllando nel contempo la granularità dell'informazione: un granularità troppo piccola rende costosa la raccolta a causa dei costi di instrumentazione del codice mentre una granularità troppo grossa comprometterebbe la qualità dell'informazione raccolta a causa della mancanza di dettaglio.

Optimization Policy plugins hanno il compito di prendere decisioni in merito alle ottimizzazioni da applicare interagendo con l'interfaccia di profiling e gli altri moduli di ILDJIT. Attualmente la politica attiva in ILDJIT si basa su un sistema di priorità a quattro livelli: ai metodi che hanno alta priorità è applicato il livello base di ottimizzazione per minimizzare la latenza. I livelli superiori sono invece associati in base alla complessità del metodo; all'aumentare della grandezza e del numero di loop il livello di ottimizzazione aumenta passando al livello assegnato per metodi semplici a singolo loop fino al livello massimo assegnato ai metodi contenenti un numero elevato di loop.

Interfaccia di Ottimizzazione questo contratto permette agli plugins di ottimizzazione di dichiarare all'Optimization Policy plugins la dipendenza da altre analisi sul codice o ottimizzazioni.

3.4.5 Moduli

ILDJIT è stato progettato per offrire una struttura flessibile e estensibile, basata su moduli e plugins. Ogni Modulo di ILDJIT può essere modificato o sostituito indipendentemente dagli altri, finché le interfaccia che espone non è modificata. Ad oggi sono stati sviluppati è seguenti moduli:

Iljit é il componente principale di ILDJIT. Lo scopo di Iljit è inizializzare correttamente tutti i moduli del sistema in base ai parametri ricevuti dalla linea di comando. Questo modulo è responsabile della gestione di tutti i thread necessari all'esecuzione di ILDJIT.

Preparato l'ambiente di esecuzione Ilji carica il codice CIL e lo traduce nella sua rappresentazione IR. Durante la traduzione è necessaria l'interazione con il gestore delle chiamate interne; questo modulo interno ad il Iljit ha il compito di implementare i metodi che non possono essere direttamente gestiti dalla Base Class Libary poiché dipendenti dalla piattaforma. Ad oggi il modulo iljit supporta la compilazione Just-in-Time, Ahead-of-Time e Dynamic-Lookahead, tramite il modulo esterno Libdla.

Decoder : ha lo scopo di caricare è interpretare il formato del codice dato in input ad ILDJIT. Attualmente è fornito un modulo *iljit-ECMAsoft* compatibile con il codice CIL definito nell'ECMA-335.

Garbage Collector (GC) è responsabile del servizio di gestione della memoria per i programmi in esecuzione in ILDJIT. Per ogni oggetto richiesto questo sottosi-

stema alloca la memoria richiedendola al sistema operativo o riusando memoria precedentemente allocata e non più utilizzata.

Esistono numerose tecniche per la garbage collection. Per questa motivo si è scelto fornire ILDJIT di un interfaccia che permetta di supportare diversi Garbage Collector tramite plugins. Questa scelta permette di rendere disponibile in ogni momento diversi GC rimandando la scelta della versione da utilizzare all'utente.

Libiljitir contiene la descrizione della rappresentazione intermedia usata da ILDJIT e fornisce l'interfaccia per la manipolazione della stessa.

Libirvirtualmachine fornisce il servizio di traduzione della IR nel codice macchina nativo. Attualmente questo modulo è costruito attorno al progetto *Libjit*.

Libiljitiroptimizer implementa Optimization Policy base di ILDJIT. Questo modulo prende decisione in merito alle ottimizzazione da applicare ai metodi come descritto nella Sezione 3.4.4.

Libiljitirprofiler Implementa il modulo profiler di ILDJIT.

Libiljitu è la libreria più piccola di ILDJIT. Essa contiene le definizione e gli strumenti base utili al resto dei componenti del compilatore.

Libiljitmm implementa il servizio di accesso e navigazione dei meta-dati. Questa modulo sarà descritto in dettaglio successivamente.

Optimizer plugins ILDJIT fornisce una insieme di plugins dedicati all'analisi dataflow e all'ottimizzazione del codice. Questi plugins lavorano sulla reppresentazione IR del programma producendo nuovo codice ottimizzato equivalente a quello fornito. Le ottimizzazioni possono essere eseguite sia localmente che remotamente attraverso un *Optimizer-server* apposito.

XanLib fornisce strutture generiche per la gestione dei dati utilizzate nel compilatore come tabelle di Hash, liste, alberi ecc.. .

4 Progettazione della supporto ai Generics in ILDJIT

Nel corso del capitolo vengono introdotte le tecniche che un compilatore compatibile con CLI può utilizzare per la gestione dei Generics; vieni quindi spiegata nel dettaglio l'implementazione scelta per ILDJIT analizzando la scelte progettuali fatte.

4.1 Tecniche per la gestione dei Generics

L'implementazione del polimorfismo parametrico descritto nello standard CLI deve rispettare i seguenti vincoli:

- I tipi generici devono essere esatti. Tutti gli oggetti devono, a tempo di esecuzione, portare l'informazione completa sul loro tipo permettendo due di distinguere due realizzazioni della stessa definizione generica. Queste informazioni sono necessarie per garantire la type safety e per permettere l'introspezione sugli oggetti. Il sistema di tipi CLI infatti non è interamente statico è richiede controlli a tempo di esecuzione per la correttezza.

- L'ambiente di esecuzione deve gestire tutte le realizzazione compatibili con i vincoli imposti dalla definizione. La gestione delle differenti rappresentazioni in memoria è a carico della macchina virtuale.

- L'implementazione deve supportare la ricorsione dei generic parameter.
- Non devono esistere restrizioni sull'uso dei metodi virtuali polimorfici. Il compilatore deve accettare l'overriding di metodi generici nelle sotto classi e la loro specifica in interfacce o in classi astratte.

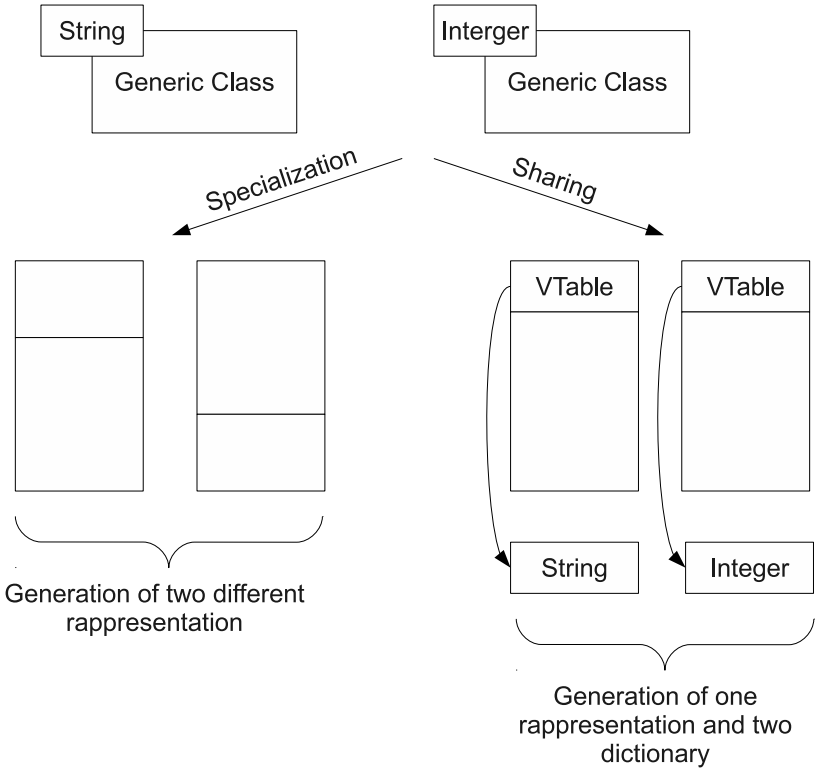
Tradizionalmente esistono due metodi per la generazione di codice, mostrati in Figura 4.1, a partire da definizioni parametriche:

Specializzazione del codice e dalla rappresentazione per ciascuna realizzazione di un componente polimorfico sono generate una rappresentazione dati e codice specifiche per quell'istanza. Questa strategia ha come vantaggi le prestazioni del codice generato e la relativa semplicità dell'implementazione, ma tuttavia è responsabile dell'aumento della dimensione del codice generato [16].

Condivisione del codice e della rappresentazione Il compilatore genera un'unica rappresentazione dati e codice per tutte le realizzazioni. Questa strategia ha come principale beneficio la ridotta dimensione del codice generato che tuttavia penalizza le prestazione del programma. Un compilatore .NET per poter gestire in modo uniforme generic argument sia di tipo valore che di tipo riferimento deve inserire nel codice conversioni di boxing e unboxing. Queste conversione richiedono molte operazioni di allocazione di oggetti o copia in memoria causando che causano che rallentano il programma [16].

In un contesto come quello .NET la strategia di condivisione classica è però poco efficiente; è stata quindi sviluppata una nuova tecnica, detta di condivisione parziale [20], che unisce i vantaggi della strategia di specializzazione con quelli della strategia di condivisione. Questo tecnica lavora su questo schema:

Figura 4.1: Modelli di implementazione di classi Generiche

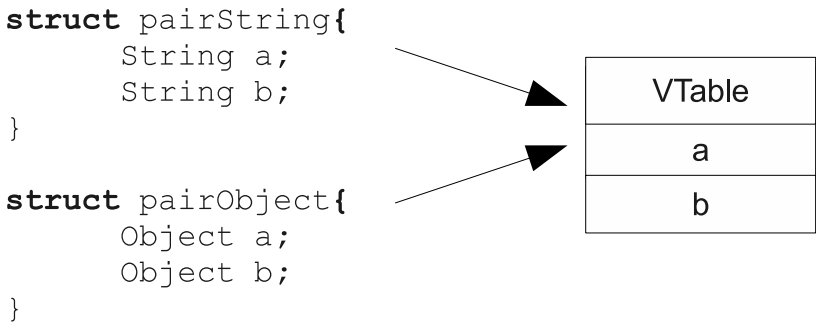


- Quando viene richiesta una particolare instanza di un tipo generico il compilatore verifica se esista un'istanza compatibile. Se questa non esiste viene costruita una nuova rappresentazione che sarà successivamente condivisa.

- Quando viene invocato l'istanza di un metodo parametrico si verifica l'esistenza di una versione compatibile già tradotta. Se questa non esiste viene costruita una nuova versione del metodo.

Due realizzazioni vengono dichiarate compatibili se la loro compilazione genera lo stesso codice, a meno delle informazioni di tipo, e la stessa rappresentazione in memoria. In particolare nel CTS tutti i tipi riferimento sono reciprocamente compatibili mentre i tipi primitivi generano istanze non compatibili tra di loro, anche a parità di memoria occupata. Per i tipi valori definiti dal programmatore la compatibilità è dichiarata caso per caso in base al layout reale della rappresentazione. Nel esempio di Codice 8 sono mostrati due tipi di dati valore: il tipo `PairString` e il tipo `PairObject`. I due tipi di dati descrivo una coppia di riferimenti rispettivamente al tipo `String` per il tipo `PairString` e a un oggetto generico `PairObject`. Il Layout dei due tipi è compatibile; infatti nel CLI la dimensione dei riferimenti è fissa è dipendendo esclusivamente dalla macchina su cui il VES è eseguito.

Codice 8: Esempio di tipi valore compatibili.



Quando si applica la strategia di condivisione occorre introdurre un meccanismo per memorizzare in ciascun oggetto delle informazioni sui generic argument.

Tutti gli oggetti, gestiti dal compilatore, sono rappresentati in memoria da un'intestazione seguita dal contenuto dell'oggetto. L'intestazione contiene informazioni utili al funzionamento tra cui il riferimento alla *VTable* del tipo dell'oggetto. La VTable è un dizionario usato per associare la corretta implementazione durante un'invocazione virtuale di un metodo; quando durante l'esecuzione occorre invocare un metodo virtuale il codice accede a questa struttura dinamica per determinare il puntatore del codice a cui deve saltare. Si consideri adesso il caso in cui un'instruzione CIL richieda di conoscere riferimento ad un tipo, un campo o un metodo che dipende da un generic parameter: nel caso della completa specializzazione di codice il riferimento è risolta durante la traduzione, mentre nel caso di condivisione del codice non è possibile conoscerne il valore a tempo di compilazione. Usando la tecnica analoga alle VTable è possibile fornire l'accesso alle informazioni sul valore corretto a tempo di esecuzione consentendo di gestire i Generics con strategia di condivisione. La VTable per le espressioni di tipo è popolata in due fasi:

1. A tempo di compilazione sono determinate staticamente le istruzioni CIL che richiederanno l'accesso alla VTable. Per queste istruzioni viene riservato uno slot nella VTable.

2. Quando un oggetto è allocato in memoria la VTable del tipo dell'oggetto è costruita.

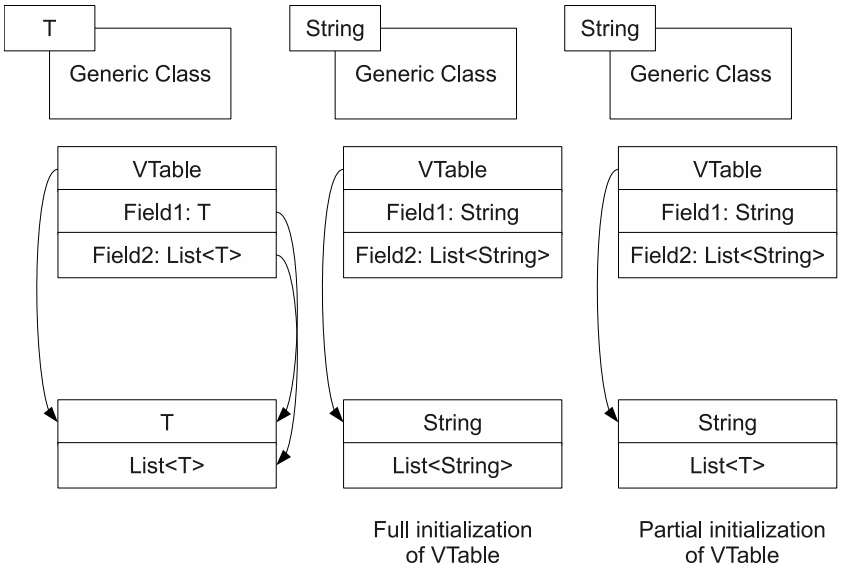
L'inizializzazione della VTable può avvenire in due modalità: inizializzazione totale o parziale:

Inizializzazione totale quando l'oggetto è allocato la VTable viene completamente popolata. L'accesso al contenuto degli slot è diretto.

Inizializzazione parziale quando l'oggetto è allocato sono popolati solo gli slot relativi ai generic argument. Quando un'istruzione richiede uno slot verifica se questo è pronto; se lo slot non è inizializzato viene invocato un servizio del compilatore incaricato di risolvere l'espressione.

In Figura 4.2 è mostrato un esempio dei due tipi di inizializzazioni. Nella inizializzazioni totale (parte centrale della figura) tutti i riferimenti vengono risolti quando la VTable è costruita. Nella inizializzazioni parziale (parte destra della figura) i riferimento al tipo `List<T>` viene risolto solo quando è richiesto l'accesso al campo `field2`.

Figura 4.2: Differenza tra l'inizializzazione totale e l'inizializzazione parziale delle VTable



L'inizializzazione parziale è preferibile quando la VTable ha grandi dimensioni e la probabilità di accesso ad uno slot è bassa. In queste condizioni il costo dell'indirezione nell'accesso alle informazioni è inferiore al costo della dell'inizializzazione totale.

Lo standard CLI offre la possibilità di definire metodi generici che il VES deve gestire. Per la gestione dei metodi è utile considerare separatamente le invocazioni dirette dalle

invocazioni virtuali.

Quando si applica la strategia di specializzazione la gestione di metodi non virtuale è immediata; ad ogni realizzazione richiesta corrisponde una versione del corpo; non è quindi richiesto alcuna struttura dinamica per contenere le informazioni sul tipo di istanza. L'applicazione della condivisione parziale richiede invece di risolvere due problemi: (1) individuare la versione corretta del metodo da invocare e (2) fornire un sistema che permetta al metodo di accedere alle informazioni di tipo. Il primo problema è automaticamente risolto dalla regola di compatibilità definita in precedenza mentre il secondo può essere affrontato passando alla funzione un dizionario, costruito dal metodo chiamante, come argomento aggiuntivo del metodo.

La gestione dell'invocazione a metodi virtuali è più problematica e richiede di affrontare il problema dell'individuazione della corretta implementazione da invocare a tempo di esecuzione. Quando si applica la strategia di specializzazione o di condivisione parziale non esiste un unico puntatore al codice adatto ad essere collocato nella VTable. Per questo motivo la gestione efficiente di questo tipo di invocazione richiede di introdurre nuove strutture dinamiche descritte nella Sezione 4.5.

Per lavoro di tesi la scelta sulla tecnica da adottare è ricaduta sulla specializzazione del codice per i seguenti motivazioni:

- La strategia di specializzazione è un estensione naturale del compilatore e richiede un numero minore di strutture di controllo.

- La generazione di codice con la tecnica di condivisione richiede un numero maggiore di controlli che aumentano la latenza tra richiesta di un metodo e la disponibilità del codice macchina.

- Il codice IR prodotto usando la strategia di specializzazione contiene maggiori informazioni sulla semantica dei tipi utilizzati. Queste informazioni sono utili

in fase di ottimizzazione per ottenere analisi dataflow più precise a vantaggio di trasformazioni più aggressive.

- L'architettura parallela di ILDJIT unita all'uso della tecnica Dynamic-Look-Ahead maschera il ritardo introdotti dalla generazione di grandi quantità di codice.

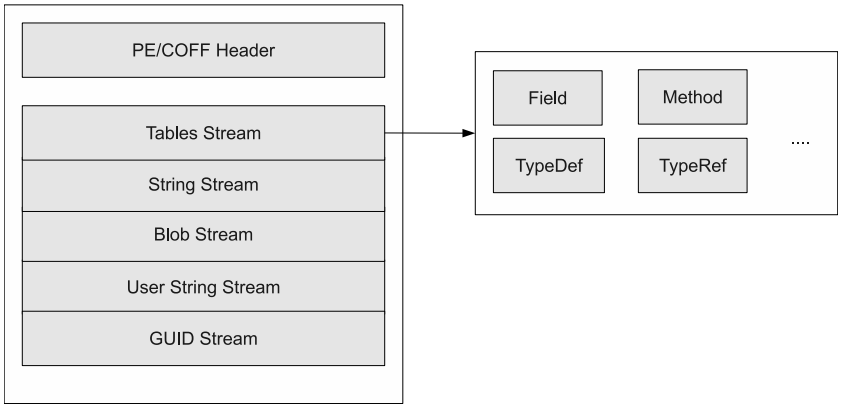
4.2 I meta-dati

Il Sistema di meta-dati è fondamentale nel CLI. Attraverso questo sistema viene codificata gran parte della semantica di un programma e in particolare vengono memorizzati i contratti del CTS. L'accesso ai meta-dati è quindi un'operazione critica per il compilatore sia nella fase di traduzione, accompagnando e specificando la semantica delle istruzioni, sia nella fase di esecuzione quando deve dare accesso ai meta-dati attraverso il supporto alla riflessività. Occorre quindi che la gestione da parte del compilatore dei meta-dati sia molto efficiente. Nel compilatore ILDJIT l'accesso ai meta-dati è gestito dalla modulo Libiljitmm approfondito nella sezione seguente; prima di descrivere questa libreria è utile chiarire la struttura e la semantica dei meta-dati.

Il sistema dei meta-dati agisce a tutti gli effetti come un base di dati distribuita in cui sono memorizzate le descrizioni dei contratti CTS. Questo meccanismo permette ai componenti di essere raccolti e distribuiti attraverso un partizionamento sia logico, offerto dagli *Assembly*, che fisico, attraverso i *Module*. Tramite gli assembly è possibile gestire il versioning, l'internazionalizzazione e la sicurezza delle risorse che vengono infine distribuite con i module, ovvero un singolo file nel formato *Portable Executable/Common Object File Format* (PE/COFF).

Il module è un contenitore complesso, come mostrato in Figura 4.3, che comprendere sia il codice CIL che i meta-dati memorizzati in due strutture dati: le *tabelle* e gli *heap*. La tabelle corrispondono alle relazioni di un comune database relazionale ma il

Figura 4.3: Struttura di un *Module*



cui numero e schema sono fissati dallo standard. Le tabelle possono essere composte solo da due tipi di campi: campi costanti, detti *bitmask*, o indici ad altri oggetti del module.

Le tabelle sono logicamente divise in due categorie: La prima categoria raccoglie tutte le tabelle che definiscono i contratti (tipi, campi, proprietà, ecc..) mentre la secondo categoria include tutte le tabelle che permettono di riferirsi ai componenti. Questa seconda categoria è necessaria per consentire alle istruzioni CIL di indirizzare contratti per cui non esiste una definizione esplicita all'interno del module come:

- tipi automatici del VES, array e puntatori.
- instanze dei Generics.
- contratti definiti un altri module.

Alla prima categorie appartengo le tabelle *TypeDef*, *Field*, *Method* e *GenericParam* mentre la seconda categoria comprende le tabelle *TypeRef*, *TypeSpec*, *MemberRef* e *MethodSpec*.

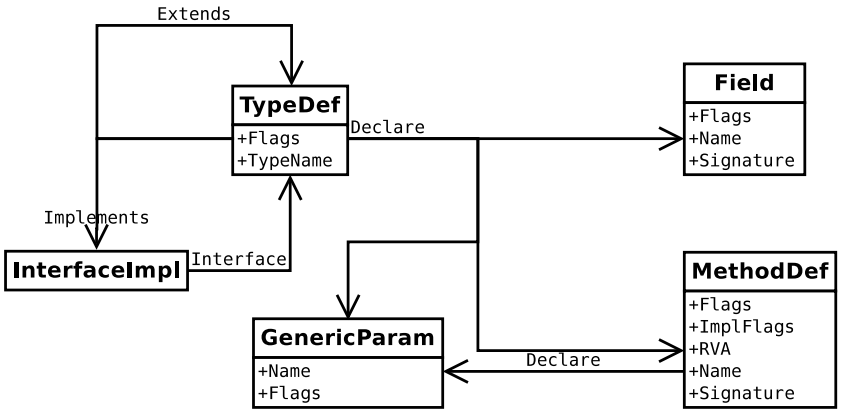
Gli heap memorizzano invece oggetti di grandi dimensione o di dimensione non fissa. Esistono quattro tipi di heaps: gli *string*, i *blob*, gli *user string* e i *globally unique identifier* (GUID). I Primi tre tipi corrispondo ad array di byte mentre gli oggetti GUID sono degli

identificatori a dimensione fissa, 16-byte. Attraverso gli Heap, è in particolare con i blob, il compilatore ha la possibilità di codificare le signature dei metodi o tipi particolarmente complesse.

Tutti gli oggetti del sistema di meta-dati possono essere estesi attraverso i *custom attributes*. I custom attributes sono annotazioni definite dallo sviluppatore usati per memorizzare a tempo di compilazioni informazioni specifiche per un'applicazione. Il CLI definisce un insieme standard di annotazioni come annotazioni sui permessi di sicurezza.

Vengono adesso illustrati le principali tabelle, il cui schema è mostrato in Figura 4.4, e i principali heap definiti dallo standard CLI:

Figura 4.4: Diagramma delle dipendenze delle tabelle



La Tabella TypeDef le tuple della tabella Typedef corrispondo alla definizione di classi o interfaccie. Ogni tupla é costituita dei seguenti campi:

TypeName punta allo string heap contenente il nome del tipo

Extends è un indice complesso che punta ad un altro Tipo; punta in particolare a *TypeDef*, *TypeRef* o *TypeSpec*

FieldList punta alla tabella *Field*. Esso segna il primo campo che il tipo definisce.

MethodList punta alla tabella *MethodDef*. Esso segna il primo metodo che il tipo definisce.

Flags è un bitmask. Codifica varie informazioni come semantica del tipo(Class, Sealed Class, Abstract Class o Interface) e la modalità di layout.

La definizione del tipo offerta da questa tabella è incompleta. La informazioni sulle interfacce implementate e l'eventuale elenco dei Generic Parameter sono raccolti rispettivamente dalla tabella *InterfaceImpl* e dalla tabella *GenericParam*.

La Tabella Field Ogni tupla definisce un campo. Un campo ammette uno e un solo tipo proprietario definito dalla tabella TypeDef dello stesso Module. Ogni tupla é costituita dei seguenti campi:

Flags è un bitmask. Codifica l'accessibilità del campo e altre informazioni sulla semantica.

Name è un indice allo string heap contenente il nome del campo.

Signature è un indice al blob heap contenente la Signature del campo. La Signature contiene le informazione sul tipo che del campo.

La Tabella MethodDef Ogni tupla definisce un metodo. Un campo ammette uno e un solo tipo proprietario definito dalla tabella TypeDef dello stesso Module. Ogni tupla é costituita dei seguenti campi:

ImplFlags e Flags sono un bitmask. Codifica l'accessibilità del metodo, il tipo di metodo (di instanza, virtuale o statico) e il tipo di implementazione (implementato in CIL, metodo interno o Pinvoke).

RVA memorizza l'indirizzo inizio nel codice CIL del metodo, quando necessario, all'interno del module.

Name punta allo string heap contenente il nome del campo.

Signature è un indice al blob heap contenente la Signature formale del metodo. Questa signature differisce da quella attuale quando si invocano metodi con numero di argomenti variabile o si fa riferimento a instanze di metodi generici.

La definizione del metodo offerta da questa tabella è incompleta. La informazioni sull'eventuale elenco dei generic parameter è raccolto dalla tabella *GenericParam*.

La Tabella GenericParam Memorizza la definizione dei generic parameter definiti da classi o metodi. Per ciascuna tupla è definito questo insieme di campi:

Name è indice allo string heap contenente il nome del parametro.

Owner è un indice complesso. Punta alla classe o al metodo al quale il generic parameter si applica.

Flags è un bitmask. Codifica i vincoli speciali e la proprietà di convarianza per il generic Parameter.

L'elenco dei Vincoli di tipo definiti dal generic parameter sono memorizzati separatamente nella tabella *GenericParamConstraint*.

La Tabella InterfaceImpl Memorizza per ogni classe l'elenco delle interfacce implementate esplicitamente. Per ciascuna tupla è questo insieme di campi:

Class è indice alla tabella *TypeDef*. Punta alla classe a cui la tupla si applica.

Interface è un indice complesso. Punta all'interfaccia a cui la tupla si riferisce.

La Tabella TypeRef Questa tabella permette di indirizzare tipi definiti in una tabella TypeDef esterna al module corrente. Ogni tupla è composta dai eseguenti campi:

ResolutionScope è un indice complesso. Punta alla descrizione del module che contiene la definizione.

TypeName è indice allo string heap contenente il nome del tipo indirizzato. dell'instanza.

La Tabella MemberRef Questa tabella permette di indirizzare in maniera generica i membri di un tipo. Ogni tupla è composta dai eseguenti campi:

Class è un indice complesso. Punta al tipo che definisce il membero. In particolare l'indice punta ad una riga della tabella *TypeDef* o *TypeRef* quando si riferisce ad un tipo regolare o alla tabella *TypeSpec* quando ci si riferisce ad una realizzazione di un Generics.

Name è indice allo string heap contenente il nome del membro indirizzato.

Signature è indice allo blob heap contenente la signature del membro. Per i metodi Il blob indirizzato codifica la signature attuale del metodo. dell'instanza.

La Tabella TypeSpec La tabella *TypeSpec* possiede solo il campo Signature. Questo campo è che è un indice a un blob heap. faccendo riferimento a questa tabella è possibile specificare tutti i tipi codificati negli heap quali array, puntatori o istanze di tipi generici.

La Tabella MethodSpec Questa tabella fornisce i metadata tokens per l'indirizzamento di istanze di metodi generici. Ciascuna tuple definisce i seguenti campi:

Method è un indice alla tabella Method o MemberRef. Questo Campo specifica il definizione di metodo generico che si intende instanziare.

Instantion è un indice al Blob Heap contentente i generic argument dell'instanza.

Heap Type Questo oggetto codifica un generico riferimento ad un tipo particolarmente complessi. Permette di memorizzare riferimenti a istanze di tipi generici, array o altri tipi che non sono definiti tramite la tabelle definiti spiegate in precedenza.

4.3 Implementazione precedente della libreria Libiljitmm

Durante il processo di traduzione il compilatore deve poter esaminare la descrizioni dei contratti per poter produrre codice corretto. Occorre quindi che esista un modulo responsabile della decodifica dei metadata tokens, che accompagnano le istruzioni, e un sistema che permetta di esaminare le proprietà del contratto che il simbolo codifica. In ILDJIT questo compito è affidato alla libreria Libiljitmm che più precisamente fornisce tre funzionalità:

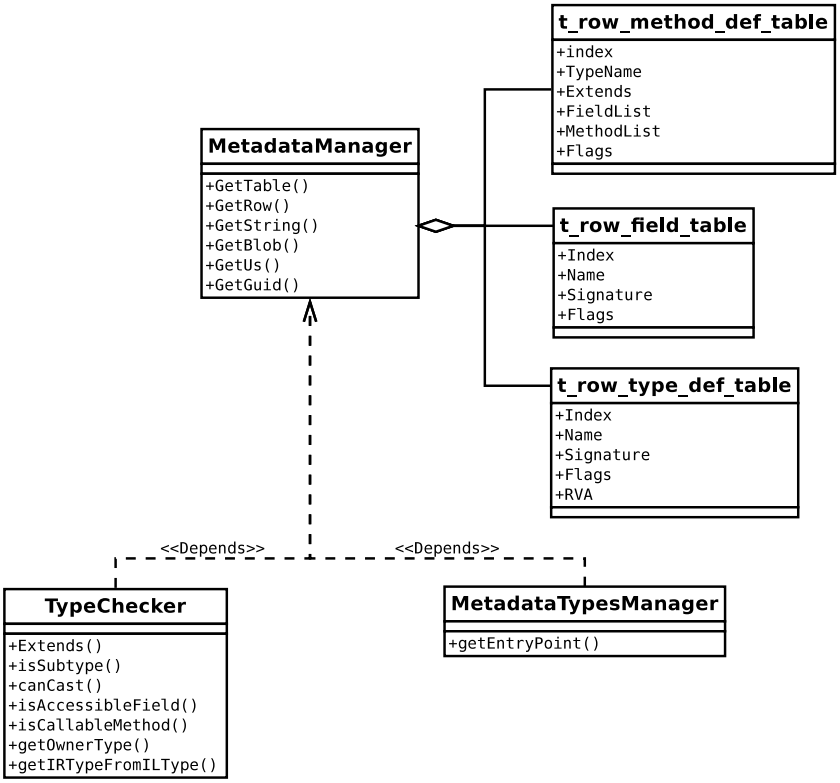
- Fornisce l'accesso alle tabelle e agli heap attraverso opportune rappresentazioni in memoria delle strutture.
- fornisce un meccanismo per la decodifica di un metadata tokens un identificatori globale del contratto.
- fornire un interfaccia per la navigazione delle proprietà dei contratti.

Come mostrato in Figura 4.5 la libreria fornisce tre interfaccie:

MetadataManager permette l'accesso alle tabelle e agli heap.

MetadataTypesManager fornisce il metodo `getEntryPoint` per la decodifica di un metadata tokens in un identificatore interno del compilatore. In particolare ogni simbolo è decodificato nella tripla $\langle$ Module, Tabella, Riga della tabella$\rangle$ che in ILDJIT è rappresenta dal puntatore del record. Facendo riferimento alla figura ogni tipo,campo o metodo sono decodificati rispettivamente in un puntatore al tipo `t_row_type_def_table`, `t_row_type_field_table` o `t_row_method_table`.

Figura 4.5: Il diagramma mostra il modello della libreria Libiljitmm



TypeChecker permette dato un record di estrarre alcune proprietà del contratto. Come mostrato in figura questa interfaccia per esempio permette di determinare la classe padre di un tipo (`Extends`) verificare l'accessibilità di un campo da un metodo (`isAccessibleField`) o effettuare altre semplici operazioni. Il `TypeChecker` non supporta tutti i contratti CTS ma solo un limitato un suo sotto insieme necessario durante per la traduzione del codice CIL in codice IR.

Questa implementazione offre poche funzionalità rendendo necessario ad altri moduli l'accesso diretto ai meta-dati per poter ottenere tutte le informazione necessarie. Per esempio l'accesso alla lista dei campi dichiarati da una classe, operazione necessaria sia nel calcolo del layout che utile nella libreria riflessiva, è implementato separatamente dai due sottosistemi. Questa scelta appare poco efficiente non permettendo la condivisione delle informazione tra i due moduli.

Esistono oltretutto un problema nella scelta del identificatore per i contratti. La scelta di utilizzare un identificatore di basso livello genera limita fortemente i contratti identificabili:

- CTS permette di dichiarare tipi quali i puntatori o array per cui non esiste una definizione esplicita in alcun module ma devono essere gestiti in automatico dal VES.

- Le realizzazioni dei Generics non sono descritte da record in tabelle ma attraverso blob heap. In aggiunta la stessa realizzazione può essere descritta da più blob heap senza violare alcuna regola del CTS.

La conseguenza di questa strategia nella gestione dei contratti emerge anche nella type-safety. Prediamo per esempio l'esecuzione del codice 9. L'esecuzione corretta genera un eccezione `InvalidCastException` quando si prova ad assegnare un array ad un oggetto `string`. Usando questa implementazione della libreria libiljimm il tipo `string[]` è erro-

neamente decodificato nello stesso identifcatore della classe `string`. Quando è richiesto il type-checking come conseguenza viene dichiarata la compatibilità tra i due tipo.

Codice 9: Il test deve generare un eccezione `InvalidCastException`. Il controllo di compatibilità dai tipi richiesto dall'assegnamento non è gestito correttamente dalla precisamente implementazione della libreria Libiljitmm.

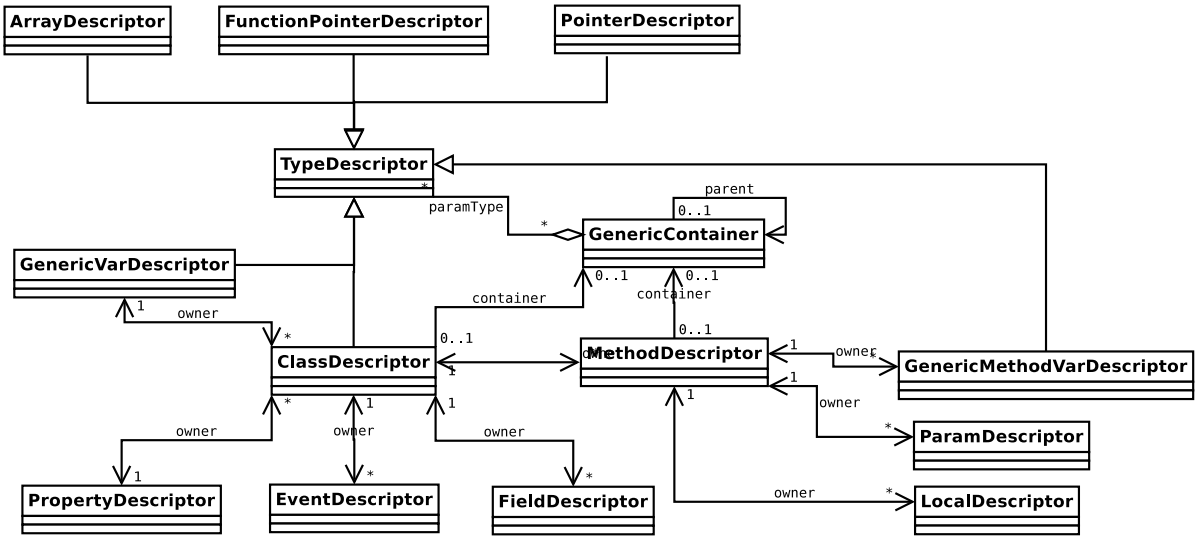
```
using System;

public class test
{
 public static int Main(){
  object obj1=new string[1];
  string obj2=(string)obj1;
  return 0;
 }
}
```

4.4 Implementazione attuale della libreria Libiljitmm

Per risolvere i problemi dell'implementazione precedente e supportare i Generics la libreria Libjitmm ha subito una profonda revisione. Come primo obbiettivo si è individuato un nuovo meccanismo per l'identificazione dei contratti. La scelta è ricaduta sul design pattern dei descrittori [18]. Un descrittore è un indice ad una struttura dati interna alla libreria che contiene i dettagli del contratto. Questo design pattern è tipicamente utilizzato nelle basi di dati object-oriented o nel object-relational mapping. Nel contesto delle basi di dati, ottenere il valore di un attributo richiede operazione complesse che coinvolgo anche più sotto sistemi; attraverso il descrittori è possibile mascherare questa complessità eliminando ogni legame diretto tra l'informazione richiesta è il formato con cui questa è memorizzata.

La nuova implementazione fornisce quindi solamente l'interfaccia `MetadataTypesManager`. Questa interfaccia fornisce la decodifica di un simbolo attraverso il metodo due metodi.

Figura 4.6: Il diagramma della classi UML del sistema di descrittori della libreria libjitmm



getEntryPoint permette di ottenere il descrittore della funzione principale (e.g *Main Function*) del programma.

fetchDescriptor decodifica i simboli che accompagnano le istruzione rispetto al `GenericContainer` di un altro descrittore. Il `GenericContainer` usata dalla libreria per descrivere i generic argument usati da un istanza di un Generics. Questa struttura è utilizzata dal modulo di decodifica per l'espansione ricorsiva delle definizione generiche. Al termine della decodifica si possono ottenere tre differenti tipi di descrittori corrispondenti ai diversi tipi di simboli che si possono trovare: un `TypeDescriptor`, se il simbolo si riferisce ad una descrizione di un tipo, mentre un `FieldDescriptor` o un `MethodDescriptor` quando il simboli si riferisce rispettivamente ad un campo o un metodo di una classe.

La struttura dei descrittori è in realtà più complessa, come mostrato in figura 4.6. La semantica dei contratti di tipo nel CTS è molto varia; non tutti i tipi permetto di definire li stessi membri o ammettono le stesse operazioni: ad esempio a differenze delle classi

non è permesso agli array definire esplicitamente campi o metodi. A ciò si aggiunge le differenti codiche che sono adottata per memorizzare le descrizione nei module. Questi caratteristiche hanno portato alla scelta di progettare diversi tipi di descrittori che meglio si adattassero alle caratteristiche di un singolo contratto. Per Garantire comunque una visione uniforme su descrittori di tipo è stato definito un descrittore astratto, chiamato `TypeDescriptor`, che generalizza tutti i tipi e fornisce un insieme di operazioni comuni a tutti i descrittori di tipo. Di seguito sono descritte nel dettaglio i principali descrittori e la loro funzione.

TypeDescriptor è un descrittore astratto per i tipi CTS. Fornisce un insieme di operazioni uniforme a tutti i descrittori di tipo. Per ogni descrittore di tipo esiste uno e un solo `TypeDescriptor` associato e viceversa. Quando è richiesta un operazione questo descrittore la richiesta è rediretta sul descrittore associato. Il `TypeDescriptor` è introdotto per disporre nel compilatore di un identificatore uniforme per i contratti di tipo.

ArrayDescriptor questo descrittore corrisponde al contratto array del CTS. Ogni descrittore `ArrayDescriptor` mantiene un descrittore costituito dalla coppia formata dal `TypeDescriptor` del tipo degli elementi e dal forma del Array.

PointerDescriptor corrisponde a un puntatore a un tipo del CTS. viene internamente identificato dal `Typedescriptor` del tipo puntato.

FunctionPointerDescriptor corrisponde al contratto di puntatore a funzione. Ogni `FunctionPointerDescriptor` corrisponde ad un diversa signature del metodo.

ClassDescriptor è l'estensione del `TypeDescriptor` per i per descrizione di classi o interfacce un'interfaccia o di una classe. L'identificatore di questo descrittore all'interno alla libreria è più complesso dei casi precedenti. Quando ci si sta riferendo a un

tipo non generico o a una definizione generica l'identificatore mantenuto comprende solo la riga della tabella TypeDef. Se si sta riferendo ad un istanza di un tipo generico si aggiunge all'identificatore un contenitore (`GenericContainer`) con i generic argument utilizzati. Questa modalità di indirizzamento di accedere alle informazione in modo uniforme sia per i tipi concreti che per le definizione di tipo generiche.

GenericVarDescriptor questo descrittore corrisponde a un generic parameter dichiarato in una definizione di una classe generica. Questo descrittore non corrisponde ad alcun tipo concreto, quindi non è mai esaminato durante la traduzione, le sue informazioni possono essere esaminate attraverso la libreria di riflessività.

GenericMethodVarDescriptor è l'analogo del `GenericVarDescriptor` per i generic parameter definiti da un metodo.

In aggiunta ai descrittori di tipo la libreria Libiljitmm definisce un insieme di descrittori per i contratti dei membri di un tipo, come metodi o campi. Tutti i descrittori di questa categoria sono caratterizzati da un identificatore interno costruito a partire dal `ClassDescriptor` che dichiara il membro. Di seguito sono presentati i due descrittori principali di questa categoria.

MethodDescriptor corrisponde al contratto di un metodo. Per descrivere istanze di un metodo generico il `MethodDescriptor` aggiunge all'identificatore un contenitore con i generic argument utilizzati, come accade nei `ClassDescriptor` per le istanze di classe generiche . L'operazione più importante permessa da questo descrittore è il controllo dell'accessibilità del metodo.

FieldDescriptor per ciascun campo definito in un classe concreto questo descrittore fornisce l'accesso alla proprietà. è identificato dalla ClassDescriptor che dichiara il campo e dalla riga della tabella Field.

Su tutti i descrittori sono sempre possibile delle operazioni di base qauli locking e di accesso a un dizionario contestuale. L'operazioni di locking (`lock` e `unlock`) sono necessarie per garantire la thread-safety essendo ILDJIT un compilatore intrinsicamente parallelo. Il dizionario contestuale permette ad altri moduli del compilatore di memorizzare informazioni contestuale al contratto (es. layout del tipo) ma non gestite direttamente dalla libreria Libiljitmm avvantaggiandosi delle proprietà di località spaziale dei dati riducendo così la pressione sulla cache del processore.

Con l'introduzione dei descrittori l'esportazione dell'interfaccia `MetadataManager` non risulta più necessario in quanto la delle navigazioni dei contratti avviene attraverso solo le operazioni che i descrittori offrono. Con l'introduzione di questo nuovo livello di astrazione ci si aspetta un'aumento del costo d'accesso. Per questo motivo vengono introdotte dei meccanismi di caching con lo scopo di prevenire la decodifica di simboli precedentemente già analizzati. Quando un simbolo è decodificato la libreria tiene traccia in un indice del descrittore e dei generic argumen usati nella decodifica. Se il simbolo è nuovamente richiesto si verificano le seguenti condizioni:

- se il simbolo non richiede generic argument per la decodifica si ritorna in descrittore ottenuto in precedenza.

- se il simbolo richiede uno o più generic argument si verifica che esiste un istanza già decodificata compatibile con il GenericContainer passato; e in questo caso si ritorna il descrittore ottenuto in precedenza. Due GenericContainer sono compatibili i generic argument usati nella decodifica sono inclusi nel GenericContainer passato.

Per minimizzare il costo l'accesso fisico ai meta-dati nella nuova implementazione sono state aggiornate le primitive per l'accesso alla tabelle. La decomposizione delle relazioni in tabella, prevista dal CLI, non è sempre ottimale, esistono infatti casi in cui l'accesso a proprietà dei contratti risultato gravosa. Prendiamo per esempio il caso in cui si voglia conoscere l'elenco delle interfacce implementate da una classe, operazione necessarie

per esempio durante il calcolo della layout della classe. Lo standard CLI memorizza le informazioni sulle interfacce nella tabella `InterfaceImpl` a cui è associato il seguente schema:

- Class: è un indice alla tabella TypeDef. Indica la classe su cui si vuole implementare un interfaccia.
- Interface: è un indice alla tabella TypeDef, TypeRef o TypeSpec. Punta alla interfaccia che si vuole implementare.

Partendo da un record della tabella TypeDef è necessario eseguire $TypeDef(index) \bowtie InterfaceImpl(Class)$. Questa operazione può essere accelerata se sono disponibili degli indici sulle chiavi di join. Nel caso di CLI la gestione degli indici è facilmente realizzabile in quanto le tabelle oggetti immutabili durante l'esecuzione del VES. Quando è richiesta per la prima volta un indice su una tabella ILDJIT crea l'indice che è mantenuto valido per tutta l'esecuzione del programma. Il costo della creazioni dell'indice è proporzionale alle dimensione della tabella ma quando l'indice è create le operazioni di associative sono eseguite in tempo costante. Quindi durante la fasi di traduzioni quando si fa riferimento a contratti memorizzati in module di grandi dimensione e frequentemente acceduti, come quello della Base Class Library, il costo di accesso medio risulta sensibilmente inferiore.

4.4.1 Effetti della nuova libreria

L'introduzione della nuova libreria a permesso una notevole semplificazione dei moduli del compilatore. Prendiamo l'esempio della libreria di riflessività. La riflessività offre al programma oggetti che incapsulano le informazioni sui module e sui tipo. Tramite la riflessione un programma può dinamicamente creare istanze di un tipo, ottenere il tipo di oggetto esistenza e invocare o a accedere a metodi o campi da esso definiti.

La classe `System.Type` è la classe principale dalla libreria di riflessione. Questa classe rappresenta un generico tipo del CTS. Attraverso questa classe è possibile conoscere

per esempio il nome del tipo, campi, metodi o altri contratti dichiarati ma eseguire operazione più complesse. A partire da un tipo è possibile costruire nuovi tipi, come array o puntatori, o ottenere nuove istanze di una classe generica.

Queste operazioni richiedono in ultimo l'interazione con il compilatore attraverso un insieme di chiamate interne che esplorano, modificano o aggiungono nuovi contratti nel compilatore. Nella precedente implementazioni molte delle funzionalità previste dalla dalla libreria di riflessività erano non disponibili o parzialmente supportate. La mancanza di un identificatore uniforme per tutti i tipi di dati nella precedentemente implementazione limitava infatti il numero dei tipi accessibili dalla libreria di riflessività: in particolare era possibile esaminare solo classi o interfacce. Funzionalità avanzate come la definizione di nuovi non erano invece supportate non potendo costruire un identificatore per un contratto non definito in un module.

La nuova libreria risolve molti di questi problemi. La visione uniforme sui tipi adottata dalla libreria di riflessività coincide con quella realizzata dai descrittori permettedo di mappare le operazioni richieste dal programma sul `System.Type` su operazioni sui descrittori. Quando è richiesto la creazione di un nuovo tipo è facile per il compilatore costruire un descrittore adatto non dovendo fare necessariamente riferimento alla struttura fisica con i cui i dati sono memorizzati.

4.5 Invocazione virtuale e ad interfaccia per i metodi generici

La gestione delle ridefinizioni di metodi in classi derivate previsto dal modello ad oggetti del CTS richiede che parte del supporto sia garantito da codice eseguito a tempo di esecuzione. Prendiamo come esempio il codice 10, durante la traduzione del metodo `main` il compilatore non può determinare staticamente il tipo dell'oggetto puntato dalla

variable `c` ma conosce solo il suo tipo formale. Non è quindi possibile generare staticamente codice che invochi direttamente uno dei metodi scelto tra `A.f`, `B.f` o `C.f`. La risoluzione deve essere effettuata quindi a tempo di esecuzione quando si conosce il tipo dinamico di un oggetto. Per garantire la massima efficienza del codice la gestione dei metodi virtuale dipenda dal tipo di metodo considerato. ILDJIT in particolare gestisce separatamente l'invocazione a metodi virtuali di classe, a metodi virtuali generici di classe e a metodi virtuali (sia generici che non) definiti in interfacce.

4.5.1 Invocazione a metodo virtuale

L'invocazione di metodi virtuali di classe è il caso più semplice di chiamata virtuale. Il problema è risolto facendo ricorso alle *VTable*, in figura 10. Per ciascun tipo è calcolata staticamente una tabella a cui e possibile riferirsi attraverso un puntatore mantenuto nell'intestazione di ogni oggetto. La VTable è popolata nel modo seguente:

- Per ciascun nome di un metodo si riserva uno slot della tabella contenente il puntatore alla corretta implementazione del metodo.
- Quando una classe estende una classe dal padre la tabella è creata a partire da quella del padre aggiungendo i nuovi metodi e aggiornando i puntatori al codice per i metodi ridefiniti.

Conoscendo quindi il nome del metodo e un tipo formale è possibile a tempo di compilazione stabile quale slot conterrà il puntatore del metodo che si vuole invocare. Il compilatore genera quindi le seguenti istruzioni per eseguire una chiamata virtuale:

- carica la VTable dall'intestazione del oggetto
- carica il puntatore al codice del metodo memorizzato in una posizione nota staticamente della tabella.

- invoca il metodo attraverso una chiamata indiretta.

L'esecuzione di una invocazione virtuale risulta più complesse di una chiamata semplice; ci si aspetta quindi un peggioramento delle prestazioni. L'analisi delle prestazioni è effettuata nel capitolo 5.

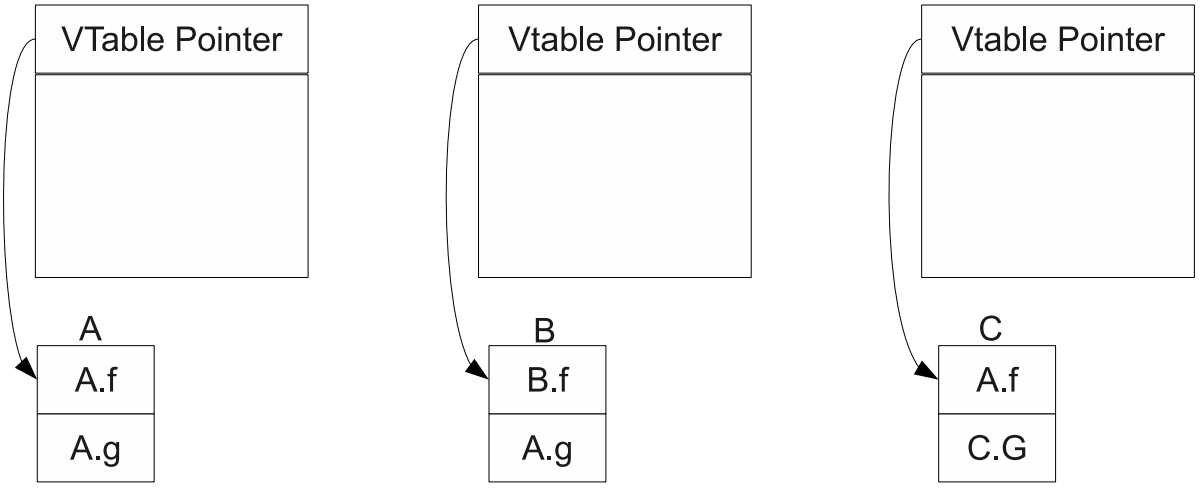
Codice 10: Esempio di invocazione virtuale di metodi

```
class A{
 virtual int f() { ... }
 virtual int g() { ... }
}

class B extends A {
 virtual int f() { ... }
}

class C extends A {
 virtual int g() { ... }
}

void main(){
 A c;
 ...
 c.f();
 ...
}
```

4.5.2 Invocazione a metodo virtuale generico

L'invocazione di metodi generici richiede l'estensione del meccanismo delle VTable. In un compilatore dinamico infatti non è possibile conoscere tutte le istanze di un metodo generica prima del calcolo della VTable stessa. In ILDJIT la VTable è stata modificata, come in figura 4.8, introducendo un livello secondario; durante la costruzione per tutte le definizioni di metodi generici è riservato uno slot che contiene il puntatore ad una

Figura 4.7: Struttura delle VTable per il Codice 10



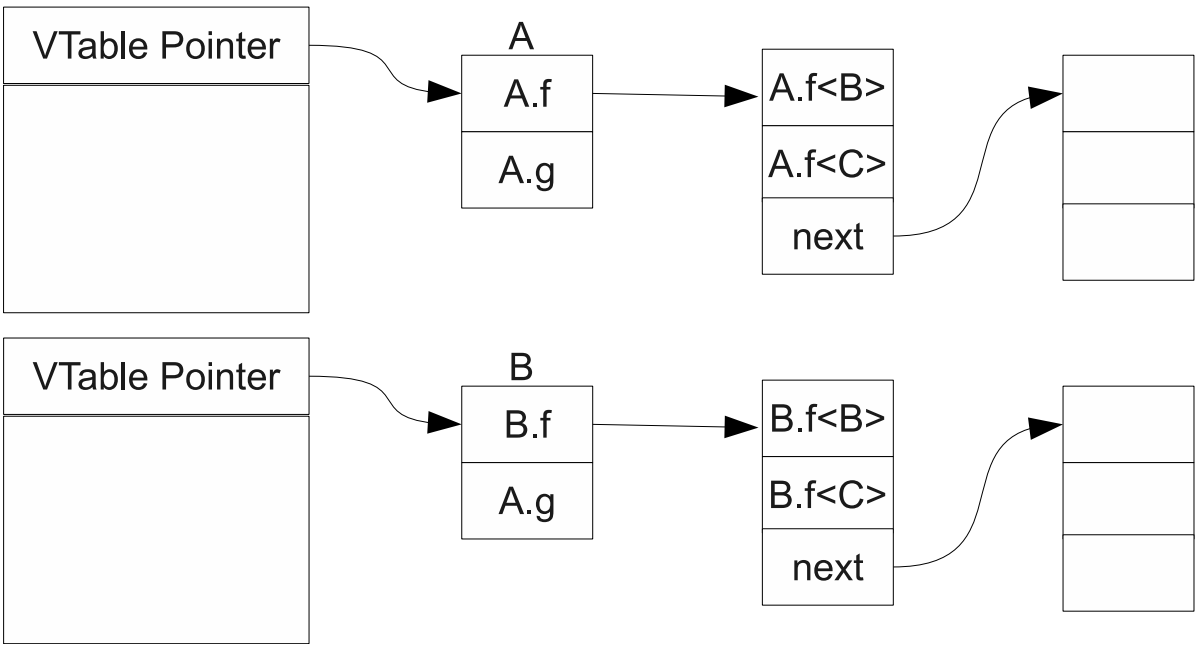
VTable secondaria. Ogni slot della VTable secondaria è riservato ad un'istanza del metodo. Come si nota in figura la struttura secondaria è realizzata su più livelli per rendere estendibile dinamicamente la tabella. Quando il numero di richieste satura le pagine a disposizione viene creato un nuovo livello. Quando il compilatore deve eseguire una chiamata virtuale ad un'istanza genera quindi il seguente codice:

- carica la VTable dall'intestazione del oggetto.
- carica il puntatore contenuto nello slot assegnato alla definizione del metodo.
- raggiunge la pagina, il cui numero è noto staticamente, che contiene lo slot assegnato all'istanza.
- carica il puntatore contenuto nello slot.
- invoca il metodo attraverso una chiamata indiretta.

Codice 11: Esempio di invocazione virtuale di metodi generici

```
class A{
 virtual int f<T>() { ... }
 virtual int g() { ... }
}

class B extends A {
 virtual int f<T>() { ... }
}

class C extends A {
 virtual int g() { ... }
}

void main(){
 A c;
 ...
 c.f<B>();
 c.f<C>();
 ...
}
```

Figura 4.8: Struttura delle VTable per il Codice 11



4.5.3 invocazione a metodo di interfaccia

Nel caso di linguaggi che permettono l'ereditarietà multipla, come per le interfacce nel CLI, l'identificazione del codice del metodo da invocare risulta più complessa rispetto ai casi precedenti. Non è possibile utilizzare l'algoritmo di costruzione della VTable spiegato nella sezione 4.5.1. Nell'esempio mostrato nel Codice 12, non è infatti possibile per per la classe `E` mantenere contemporaneamente lo slot ereditati da `A` e `B` in prima posizione.

La soluzione classica del problema è analizzare staticamente tutte le classi e riservare gli slot per queste in un unico passaggio usando la tecnica di colorazione del grafo. Si può infatti costruire un grafo così fatto: tutti i nomi di metodi costituiscono i nodi mentre gli archi sono costituiti da coppie di metodi che devo coesistere nella stessa VTable. I colori nel modello rappresentano gli slot della tabella. Nei compilatori come ILDJIT in cui è possibile caricare nuove classi mentre il sistema e in esecuzione l'approccio descritto non è realizzabile.

Invece di basarsi sulla colorazione del grafo ILDJIT applica quindi una soluzione diversa basata sul tabelle di hash, vedi Sezione 4.6, conosciuta con il nome *interface method table* (IMT). A tempo di compilazione per ciascun nome di metodo viene inserita una voce in una tabella di hash con il puntatore al codice da eseguire. Questa tabella sarà raggiungibile a tempo di esecuzione grazie a un puntatore presente nell'intestazione del oggetto. Quando un il compilatore deve invocare un metodo sull'interfaccia genera quindi il seguente blocco di codice:

- carica L'IMT dall'intestazione dell'oggetto.
- carica il blocco corrisponde all'hash del nome del metodo.
- verifica che non ci sia stata collisione. Se il test ha successo si procede all'invocazione indiretta altrimenti si risolve il conflitto secondo la politica *Separate chaining*

with list heads, descritta in sezione 4.6.2.

Data la natura dinamica della tabella di hash è facile introdurre nuove chiavi anche a tempo di esecuzione. Questa caratteristica è sfruttata per gestire le chiamate ad istanze di metodi generici. Quando durante l'esecuzione viene richiesta un'istanza di un metodo generico non è ancora disponibile nella IMT [2], il codice generato invoca il servizio del compilatore `loadVirtualMethod` che si prende carico di tradurre il metodo e aggiornare l'IMT con la nuova voce rendendo disponibile la chiave per le prossime invocazioni.

L'accesso all'IMT è più complesso rispetto a quello relativo alla VTable ma il costo è comunque paragonabile; infatti in mancanza di collisioni il numero di accessi a memoria è uguale. Bisogna considerare che la funzione di hash è calcolata solo durante la traduzione; questo permette di scegliere funzioni di hash anche complesse per ridurre la probabilità di collisione.

4.6 Miglioramenti alla libreria Xanlib

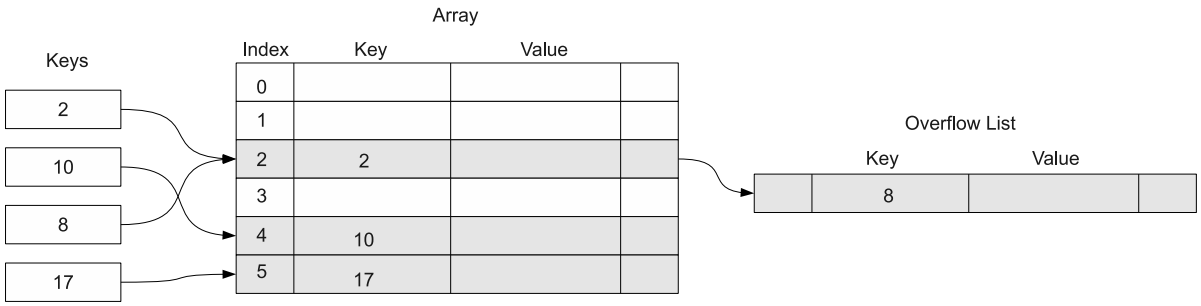
Molti moduli di ILDJIT,e in particolare della libreria Libiljitmm, richiedono strutture ad accesso calcolato o array associativi su cui costruire indici o cache. La struttura più nota per gestire l'associatività è la *tabella hash*. Questo tipo di struttura dati è usato per mettere in corrispondenza una chiave ad un valore arbitrario attraverso una funzione detta di hash che trasforma la chiave in un indice di un array in cui è memorizzato il valore corrispondente. Idealmente la funzione di hash è iniettiva cioè fa corrispondere a chiavi diverse indici diversi garantendo nel contempo che tutti gli slot dell'array abbiano la stessa probabilità di essere riempiti. In realtà una funzione di hash perfetta non è realizzabile. Occorre quindi gestire il caso in cui l'applicazione della funzione di hash su chiavi distinte produca lo stesso indice, situazione detta di collisione, attraverso politiche di risoluzione. Esistono due principali categorie di tecniche di risoluzione: open hashing, in Figura 4.9 e il closed hashing, in figura 4.10. L'open hashing memorizza le collisioni

Codice 12: Esempio di invocazione virtuale di interfaccia

```
interface A{
 virtual int f<T>();
}

interface B {
 virtual int g();
}


class C implements A {
 virtual int f<T>() { ... }
}

class D implements B {
 virtual int g() { ... }
}

class E implements A, B{
 virtual int f<T>() { ... }
 virtual int g() { ... }
}
```

esternamente alla tabella mentre il closed hashing individua un altro indice nell'array in cui salvare la coppia chiave-valore. Il costo d'accesso ad una tabelle di hash è idealmente

Figura 4.9: L'immagine mostra l'inserimento di quattro chiavi nella tabella di hash. La funzione di hash scelta è *key*modulo6. Quando la chiave 8 è inserita dopo la chiave 2 è costruita una lista per risolvere la collisione.



costante ma nella realtà è fortemente legate oltre che alla qualità della funzione di hash anche al cosiddetto fattore di carico, calcolato come il rapporto tra celle libere e elementi presenti. All'aumentare del fattore di carico il costo dell'accesso alla tabella di hash peggiora fino ad proporzionale al dimensione dell'array.

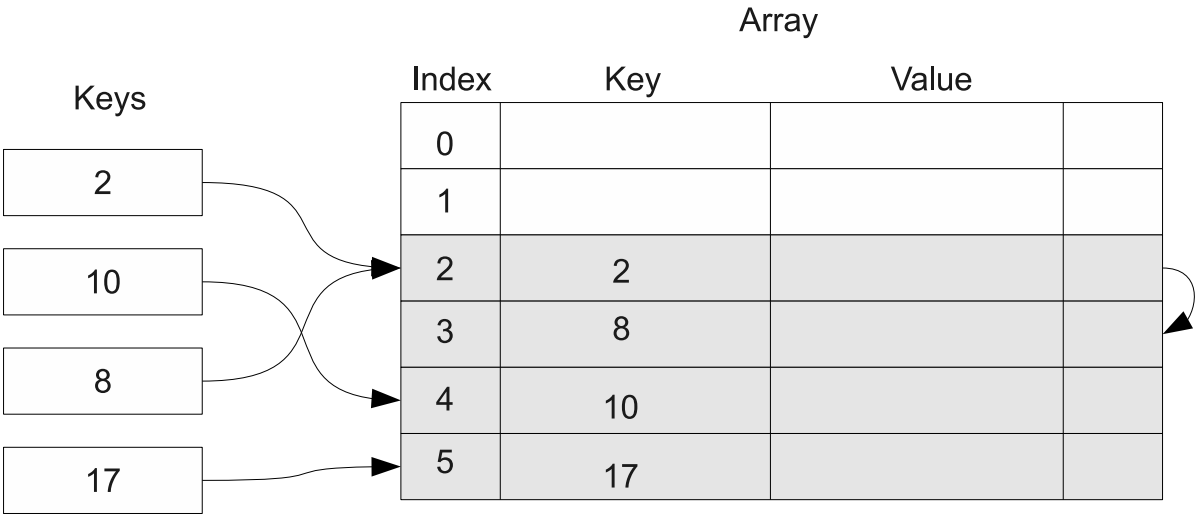
4.6.1 Implementazione precedente

Le caratteristiche della tabella di Hash precedente al progetto sono le seguenti:

Gestione della collisione con politica della scansione lineare Questa tecnica rientra nella categoria del closed hashing. Quando durante l'inserimento si incontra una collisione non si fa altro che utilizzare l'indice successivo a quello che collide, sino a che non si trova un indice libera.

Hashing estendibile Quando il fattore di carico supera una soglia fissata la dimensione della tabella è aumentata di una percentuale fissata e le chiavi vengono riorganizzate.

Figura 4.10: L'immagine mostra l'inserimento di quattro chiavi nella tabella di hash. La funzione di hash scelta è *key*modulo6. Quando la chiave 8 è inserita dopo la chiave 2 la collisione è risolta inserendo la chiave nel indice libero, in particolare nel esempio l'indice 3.



La funzione di hash è a carico del modulo che utilizza la tabella. Quando non viene specificata alcuna funzione è applicato un semplice operatore di modulo per mappare la chiave all'indice.

La fase di test condotta durante lo sviluppo del lavoro ha portato alla luce i limiti della implementazioni utilizzata. Il problema riscontrato è stato un evidente fenomeno noto come clustering primario. A causa della bassa qualità della funzione di hash è della politica di gestione delle collisione si ha la formazione di addensamenti di chiavi in gruppi continui, detti appunto clusters. La formazioni di cluster fa si che la probabilità di accesso all'indice successivo sia più alta dipendendo non solamente dalla funzione di hash ma anche dalla probabilità che il cluster si ingrandisca. Il clustering primario ha effetto quindi sulle prestazioni dell'accesso alla tabella di hash rendendo il costo dell'accesso proporzionale alla dimensione media dei cluster. La formazione dei cluster in questa implementazione è da ricondursi alla funzione di hash non adatta alla struttura tipica

della chiavi utilizzate. Le chiave sono tipicamente indirizzi a memoria, allocata dinamicamente, caratterizzati da una struttura regolare: in particolare i bit più significativi presentano bassa variabilità, essendo tutti gli indirizzi nello stesso segmento di memoria, mentre i bit meno significativi devono rispettare le regole di allineamento richiesto dal processore.

4.6.2 implementazione corrente

Per garantire migliori prestazioni di accesso alla tabella di hash questa ha subito una profonda revisione. Per proteggere la tabella da funzioni di hash deboli si è fatto ricorso a due soluzioni note in letteratura [22]:

- Si è inserito un nuovo livello di hashing basata sulla rotazione dei bit della chiave. Questo nuovo livello è puramente combinatorio per garantire il minimo impatto sulle prestazioni.

- Per minimizzare la probabilità di collisione è utile che la dimensione della tabella sia sempre fissata ad un numero primo. La nuova implementazione ,quando la soglia del fattore di carico è superata, incrementa la dimensione della tabella al numero primo più vicino al doppio di quello usato in precedenza.

Le tabelle di hash usate dal compilatore raggiungono spesso grandi dimensioni arrivando a contenere anche migliaia di elementi. Per garantire su grandi tabelle buone prestazioni, usando la tecnica del closed hashing, bisogna mantenere il fattore di carico a livelli molto bassi causando un consumo di memoria eccessivo. Si è deciso quindi di sostituire la politica di risoluzione delle collisioni passando a una tecnica di open hashing, nota come *Separate chaining with list heads*, caratterizzata da una caduta più dolce delle prestazioni d'accesso all'aumentare del fattore di carico. La tecnica utilizzata, mostrata in Figura 4.9, memorizza le collisioni in liste concatenate esterne alla tabella,

ma a differenza di altre tecniche basate sul concatenamento mantiene il primo elemento all'interno dell'array per favorire la efficienza a livello cache.

5 Valutazione delle prestazioni

Questo capitolo mostra i risultati ottenuti dalla introduzione della nuova libreria e dalla inclusione del supporto ai generics. Sono quindi presentati i test effettuati, l'ambiente di valutazione e infine i risultati ottenuti.

5.1 I benchmark

Per verificare le prestazioni della nuova libreria sono stati selezionati test da diverse sorgenti: sono stati inclusi test inclusi del *Java Grande Forum Benchmark suite*(JFG) [15], test sintetici caratterizzati dal consumo elevato di risorse, dalla versione C# della suite *Scimark* [21], che comprende test su algoritmi per il calcolo numerico e scientifico, dalla *Collective Benchmark* (CBench) [4], che comprende applicazioni di uso reale, e infine i *Micro-Benchmark*, un insieme di test scritti ad-hoc per stressare alcuni parti della macchina virtuale. I risultati così ottenuti sono riassunti nei grafici riportati successivamente; le due versioni della macchina virtuale ILDJIT, con e senza supporto ai generici, sono rappresentate da due barre i cui valori sono stati normalizzati al tempo di esecuzione del test sulla piattaforma Mono.

5.1.1 L'ambiente di test

Tutti i test sono stati eseguito su un unica macchina equipaggiato con un processore intel I5 750, con quattro core. Ogni core possiede una cache di primo e secondo livello

private e una cache di terzo livello è condivisa fra tutte le CPU.

Il sistema esegue Ubuntu GNU/Linux 10.04 [24] con il kernel aggiornato alla version 2.6.32. Sulla macchina di test sono stati eseguiti con Mono, version 2.6.4 l'ultima disponibile maggio 2010, e con le versione di ILDJIT con e senza supporto ai Generics.

Ogni benchmark è stato eseguito venti volte impostando i parametri ottimizzazioni al minimo consentito del compilatore (es. in ILDJIT questo equivale all'opzione `-O0`). Dai dati raccolti sono estati estratti la media e la deviazione standard usati come indicatori delle prestazioni. I risultati così ottenuti sono stati rappresentati nei istogrammi riportati successivamente normalizzando i valori dell'esecuzione delle tue versioni di ILDJIT rispetto al tempo di esecuzione sulla piattaforma Mono.

5.1.2 Micro-Benchmark

Questi test sono stati sviluppati durante il lavoro per stressare i componenti interni della macchina virtuale e per potere valutare quali componenti del sistema dovessero essere ottimizzati. Sono stati in particolari sviluppati due categorie di test:

Call Test comprende i test per verificare il latenza introdotta nell'invocazione di un metodo. Per ogni tipo di invocazione è stato creato un test apposito.

Test su contenitori Questo test prevede l'inserimento e la cancellazione di oggetti da un contenitore, nel caso specifico uno stack. Per questo test sono state previste tre tipi di contenitori: un contenitore specializzato a meno per un tipo di dati, un contenitore polimorfico e un contenitore generico. Questo test permette di valutare il costo delle operazioni di boxing e unboxing quando si lavora con il contenitore polimorfico e valutare la prestazione tra il codice specializzato dal compilatore e il codice specializzato a mano.

Nelle figure 5.1 e 5.2 sono riportati i risultati ottenuti da Micro-Benchmark. Come si può vedere non è tutti stato possibile eseguire tutti i test sulla versione precedente di ILDJIT per la mancanza del supporto ai Generics.

Da grafici in Figura 5.1 sono mostrati i risultati dei test sulla prestazione di invocazione. L'introduzione dei Generics infatti non introduce nessun peggioramento nella gestione di metodi non generici. L'introduzione della IMT, vedi Figura 5.3, ha permesso una notevole riduzione del costo delle chiamate ad interfaccia. La decisione in sede di progetto di differenziare la gestione delle chiamate di interfaccia generiche da quella delle chiamate virtuali generiche permesso di ottenere tempi inferiori del 7% nell'invocazione di queste ultime. I costi dell'invocazione sono riassunti in tabella sottostante

Tabella 5.1: Tempi di invocazione di un metodo. I tempi sono normalizzati al tempo necessario per una chiamata diretta nel compatore ILDJIT senza supporto ai Generics

	Metodo virtuale	**Metodo generico virtuale**	**Metodo d'interfaccia**
ILDJIT+Generics	1.007	1.014	1.08
ILDJIT	1.006	-	20.98

Le prestazione nei test sui contenitori in figura 5.2 mostrano una situazione simile. L'uso dei contenitori polimorfici annulla i vantaggi prestazionali che ci si aspetterebbero dall'uso dei tipi valore. Nel caso di contenitori polimorfici l'utilizzo dei tipi valore peggiora le prestazione a causa delle operazioni di copia in memoria che le conversioni boxing e unboxing richiedono. L'introduzione dei Generics permette di ottenere due vantaggi. In primo luogo non sono necessarie le conversioni cast permettendo un miglioramento delle prestazioni rispetto ai contenitori polimorfici anche quando si utilizzano tipi riferi-

Figura 5.1: Risultati dei Micro-Benchmark per le invocazioni a metodi

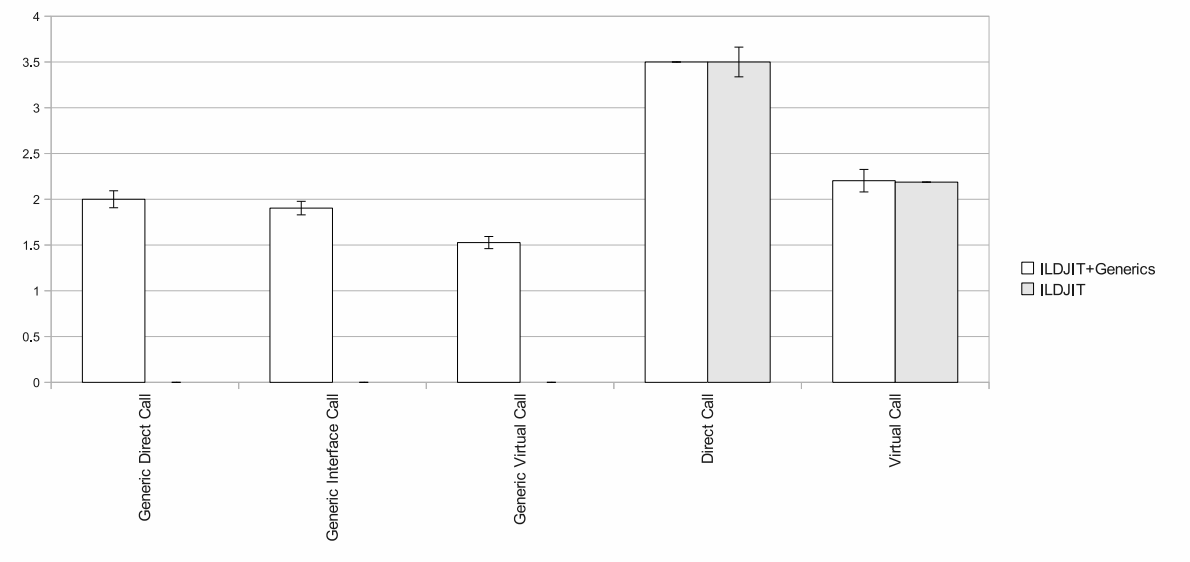
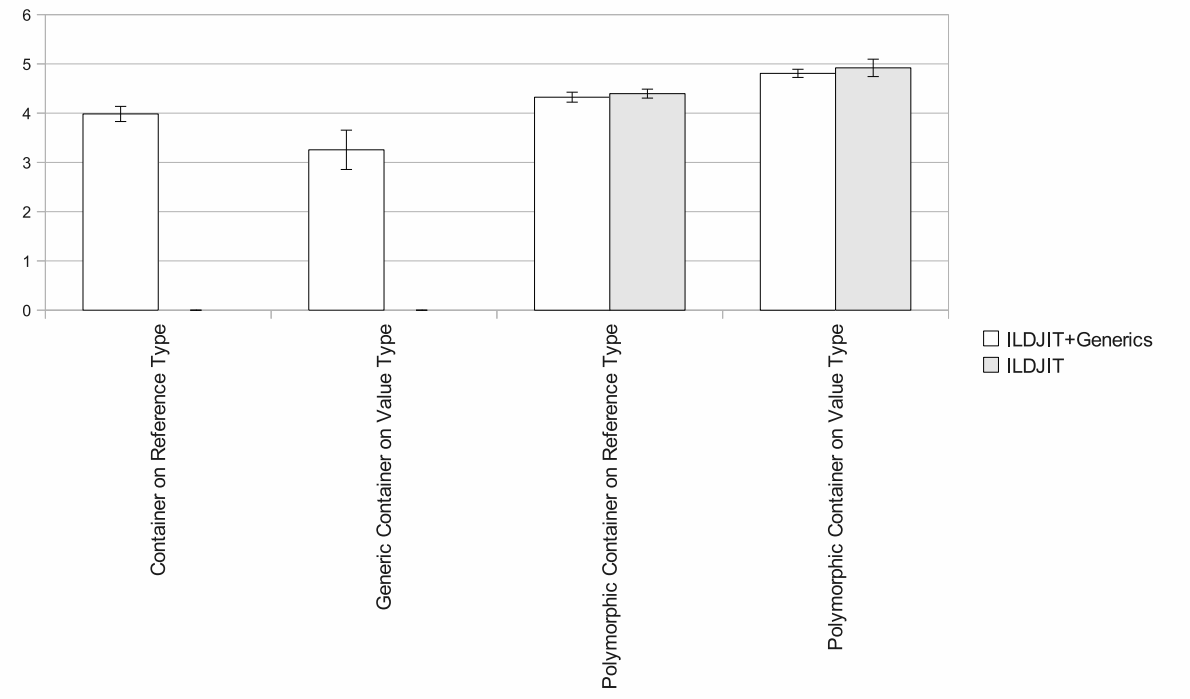
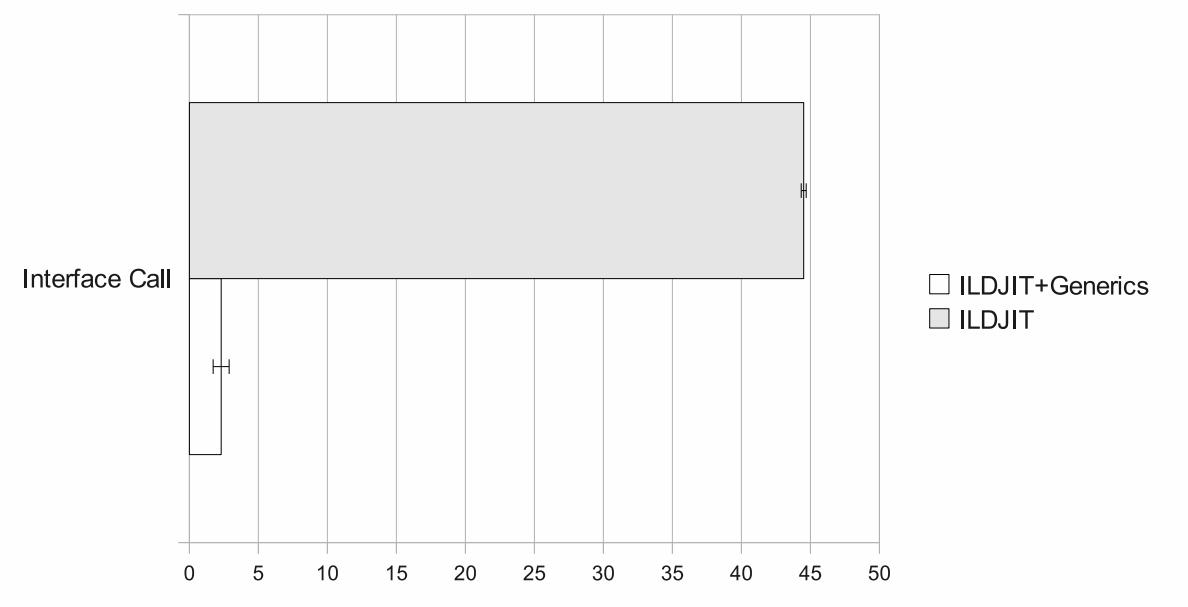


Figura 5.2: Risultati dei Micro-Benchmark per i contenitori



mento. In Secondo luogo la specializzazione del codice non richiede l'introduzione delle conversione boxing con un conseguente speed-up del 30%.

Figura 5.3: Risultati dei Micro-Benchmark per le invocazioni a metodi d'interfaccia



5.1.3 Java Grande Foundation benchmark e Scimark

la JGF è un insieme di test progettati per valutare le performance della macchina virtual su applicazioni che richiedono grandi quantità di risorse di calcolo e di memoria. Da questa suite sono stati estratti i benchmark sequenziali che si dividono in tre sezioni:

Section 1 comprende tutti i test che valutano funzionalità di basso livello come semplici operazione matematica, l'invocazione di metodi o le conversione cast. Questa sezione comprende i test `JGFArithBench.exe`, `JGFLoopBench.exe`, `JGFCastBench.exe`, `JGFAssignBench.exe` e `JGFCreateBench.exe`.

Section 2 comprende kernel numerici usati frequentemente in applicazioni reali. I kernel scelti da questa sezione sono `JGFheapsort.exe`, `JGFFFT.exe` e `JGFSparseMatmult.exe`

Section 3 contiene applicazioni reali di calcolo scientifico. la complessità di questi benchmark è maggiore rispetto alle due sezioni precedenti. Sono stati scelti in particolare i test `JGFRayTracerBench.exe`, `JGFRayTracerBenchModified.exe` `Euler.exe` e `Moldyn.exe`.

La suite Scimark comprende un buon numero di kernel di calcolo numerico. I test sono stati progettati per la valutare in maniere approssimativa il numero di MFlop (milioni di istruzione in virgola mobile al secondo) raggiunti dalla macchina virtuale. I kernel selezionati dal progetto sono FFT (`SciMarkFFT.exe`), Gauss-Seidel relaxation (`SciMarkSOR.exe`), Sparse matrix-multiply (`SciMarkSPARSEMATMULT.exe`), Monte Carlo integration (`SciMarkMONTECARLO.exe,`) e dense LU factorization (`SciMarkLU.exe`).

I benchmark estratti dal JFG e da Scimark sono stati eseguiti con due insieme di dati di diversa complessità per analizzare il comportamento su diverse situazioni di carico; i risultati dei test per i dataset piccolo e dataset grandi sono riportati rispettivamente in Figura 5.4 e in figura 5.5. Nei kernel matematici i servizi che la libreria Libiljitmm offre sono invocati con minor frequenza che in altri tipi di test e normalmente solo in fase di traduzione. Ciò nonostante come si vede in figura 5.4 molti test eseguiti sul dataset piccolo mostrano che in diversi casi come `JGFAssignBench.exe` e `JGFCreateBench.exe`, che richiedono maggiormente l'accesso ai meta-dati, sia stato possibile ottenere uno speedup positivo. I test su cui il guadagno è maggiore sono i test come `Euler.exe` che fanno grande uso dei vettori. nel CLI l'accesso hai vettori è sempre controllato per garantire sia la type-safety che il bound-safety. La nuova implementazione permette di avere controlli di tipo più veloci permettendo di guadagnare anche il 15% nei tempi di esecuzione.

Figura 5.4: Risultati dei Benchmark JGF e scimark sul dataset piccolo

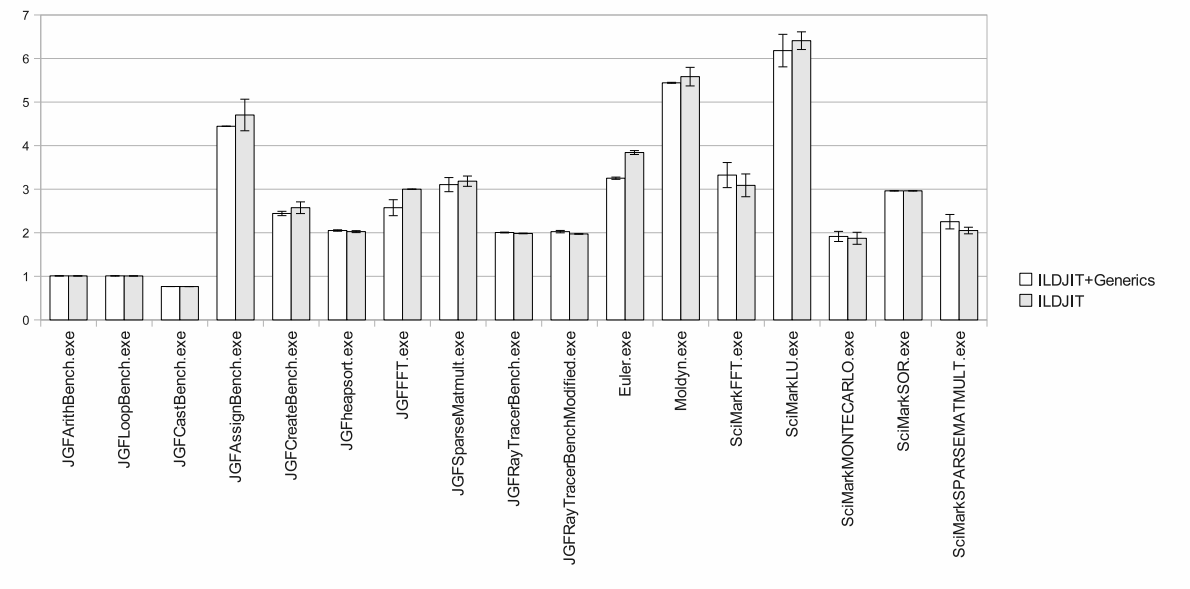
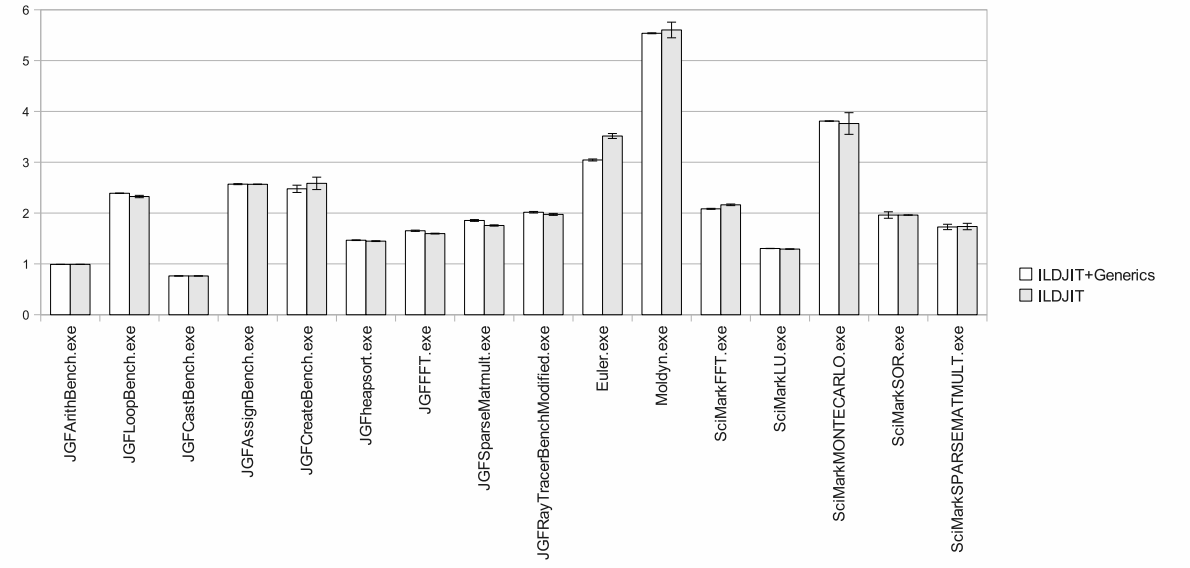


Figura 5.5: Risultati dei Benchmark JGF e scimark sul dataset grande



Passando al dataset grandi, in Figura 5.5, si nota che la differenza tra le due implementazioni si riduce. Usando il dataset di dimensioni maggiori il tempo di esecuzione è infatti dominato dal tempo di esecuzione del kernel. L'unico guadagno significato rimane sempre nel test `Euler.exe` che richiede, come già detto, frequentemente l'accesso ad informazione legate agli array.

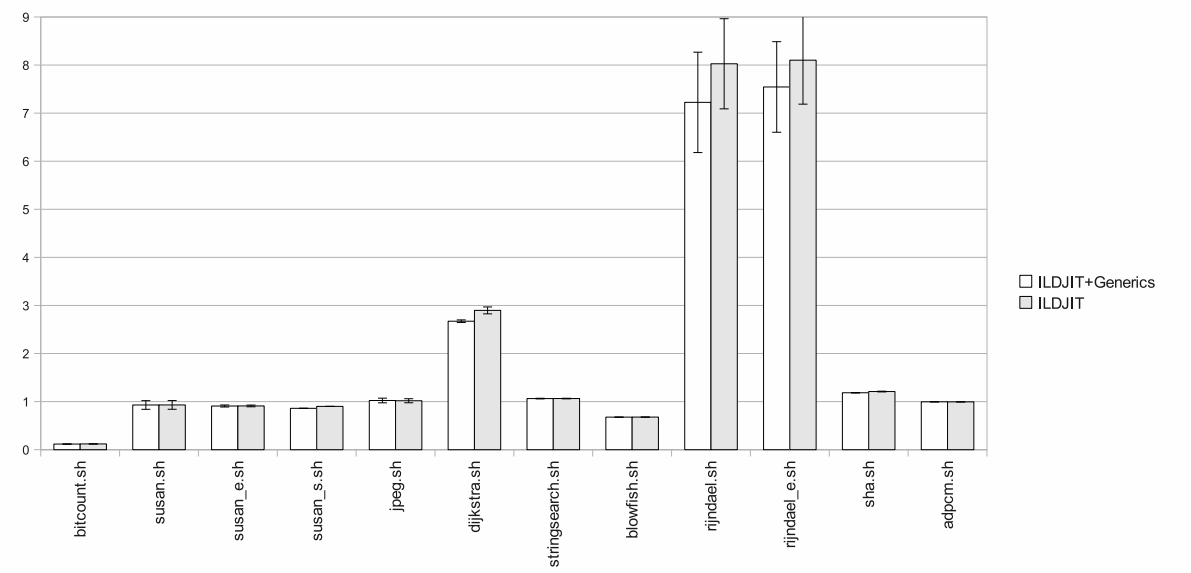
5.1.4 Collective Benchmark

I Collective Benchmark (CBench) sono una collezione di programmi sequenziali assemblati dalla comunità opensource per fornire una suite di benchmark costituita da programmi realistici. A differenza delle suite precedenti (test progettati in C#) i CBench sono scritti in linguaggio C. Questa suite permette di valutare il comportamento della macchina virtuale durante la gestione di un codice CIL profondamente diverso da quello prodotto a partire da C#. Il codice contiene solamente tipi valore, tutti i metodi generati sono statici, è permesso l'utilizzo di puntatori non gestiti dalla macchina e di invocazione indiretta ai metodi. I risultati di questa suite sono mostrati in figura 5.6. I test mostrano lo stesso comportamento visto nei test JFG; i test che sentono maggiormente il passaggio alla nuova libreria sono i test `dijkstra.sh`, `rijndael.sh` e `rijndael_e.sh` il cui funzionamento prevede frequentemente accesso ai vettori.

5.2 Conclusioni

Tutti i test mostrano una bassa varianza a garanzia della qualità dei risultati ottenuti. I risultati mostrano che in generale la piattaforma ILDJIT è ad oggi ancora inferiore alla piattaforma Mono, ciò nonostante, l'introduzione delle nuova libreria per i meta-dati e i nuovi meccanismi per le gestione delle invocazioni virtuali hanno permesso di ottenere dei miglioramenti rispetto al passato. Analizzando i dati ottenuti dal modulo di profiling di ILDJIT, vedi Tabella 5.2, si evidenzia che il costo della traduzioni da CIL a IR, e di

Figura 5.6: Risultati dei Benchmark CBench



conseguenza il tempo speso tra la richiesta del metodo e la sua disponibilità come codice macchina (in tabella mostrati come trampolini), sono sensibilmente diminuiti grazie alla nuova implementazione. La minore latenza si traduce, quando il compilatore lavora in modalità Just-In-Time, in un minore tempo nella reazione dei programmi specie quando si traducono programmi o di piccole dimensioni o caratterizzati dalla presenza di un numero notevole di metodi semplici.

Tabella 5.2: Nella Tabella sono riportati gli speedup ottenuti nella nuova versione del compilatore rispetto alla versione senza supporto ai Generics

Inizializzazione del compilatore	**Traduzione da CIL a IR**	**Inizializzazione della Memoria Statica**	**Trampolini**
1.07	1.67	1.29	1.32

6 Conclusioni

In questa tesi è stata presentata l'estensione del compilatore ILDJIT del Politecnico di Milano, per la programmazione generica, ossia i Generics dello standard ECMA-335 anche noto come .NET.

Poiché il supporto per i generici ha un vasto impatto su molte parti del compilatore, questo lavoro ha permesso di migliorare diversi sottosistemi della macchina virtuale quali la libreria per la riflessività e il traduttore da CIL a IR.

La fase di valutazione ha inoltre mostrato che per poter competere con le principali macchine virtuali, è necessario non soltanto scegliere algoritmi idonei, ma anche adottare una programmazione di basso livello e un'organizzazione particolarmente efficace.

Il compilatore esteso con i generici è stato rilasciato ed è ora utilizzato da diversi gruppi di ricerca qui, a Harvard, e altrove. Le sue prestazione sono migliorate rispetto al precedente versione, anche per programmi che non usano i Generics.

Lo sviluppo dei Generics in ILDJIT non è da considerarsi finito; parlando degli sviluppi futuri vi sono tre punti principali.

Primo, si intende confrontare le tecniche di generazione del codice dei metodi, la specializzazione da noi adottata e la condivisione parziale presenti in altri compilatori.

Secondo, la gestione dei Generics implementata è pensata esplicitamente per il modello di compilazione Just-In-Time, ma ILDJIT è un compilatore multi modalità; occorre quindi pensare ad implementazioni specifiche per le diverse tecniche di compilazione supportate.

Terzo, la nuova gestione dei meta-dati ha aperto la strada allo sviluppo di una nuova libreria per la riflessività che permette al programma, non solo di esplorare i meta-dati, ma anche di definirne di nuovi. Questa caratteristica è alla base della tecnologia *Dynamic Language Runtime* (DLR) [6] che permette al .NET di supportare linguaggi dinamici come il Python.

Infine, questo lavoro ha sostanzialmente completato la copertura dello standard ECMA-335 da parte di ILDJIT.

Bibliografia

[1] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2007.

[2] S. Fink D. Grove B. Alpern, A. Cocchi and D. Lieber. Efficient implementation of java interfaces: Invokeinterface considered harmless. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2001.

[3] S. Campanoni, G. Agosta, and S. Crespi Reghizzi. A parallel dynamic compiler for cil bytecode. *SIGPLAN Not.*, 43(4), 2008.

[4] Cbenchmark. `http://ctuning.org/wiki/index.php/CTools:CBench`.

[5] Compiler group, politecnico di milano. `http://compilergroup.elet.polimi.it/`.

[6] Dynamic language runtime. `http://dlr.codeplex.com/`.

[7] Dot Net Experts. `http://www.dotnetexperts.com`.

[8] dotGNU. `http://www.gnu.org/projects/dotgnu`.

[9] E., R. Wa, and J. Gough. Technical overview of the common language runtime, 2000.

[10] An introduction to generic programming. `http://www.generic-programming.org/about/intro/`.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., third edition, 2005.

[12] ILDJIT. `http://ildjit.sourceforge.net`.

[13] ECMA International. *Standard ECMA-334 - C# Language Specification*. 2006.

[14] ECMA International. Standard ECMA-335 - Common Language Infrastructure (CLI), June 2006. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[15] Java Grande Benchmarks. `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/sequential.html`.

[16] A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. *SIGPLAN Not.*, 36(5), 2001.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, second edition, 1999.

[18] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[19] Mono. `http://www.mono-project.com`.

[20] Mono runtime generic sharing documentation. `http://www.mono-project.com/Mono:Runtime:Documentation:GenericSharing`.

[21] Scimark. `http://math.nist.gov/scimark2/`.

[22] C.A. Shaffer. *A practical introduction to data structures and algorithm analysis.* Prentice-Hall, Inc., 1997.

[23] B. Stroustrup. *The design and evolution of C++.* ACM Press/Addison-Wesley Longman Publishing Co., Inc., 1995.

[24] Ubuntu. `http://www.ubuntu.org`.

Ringraziamenti

Voglio ringraziare i miei genitori per avermi dato la possibilità di proseguire negli studi sino a questo punto, per avermi sempre sostenuto nelle mie scelte.

Un ringraziamento speciale a Simone e al il prof. Stefano Crespi per avermi dato la possibilità di lavorare su questo progetto avermi aiutato durante lo sviluppo.

Cosa dire ai ragazzi del Micro. Siete semplicemente i migliori. Devo ringraziare tutti è in particolare Andrea, Paolo, Filippo e Chez. A tutti i miei migliori auguri per una veloce e felice conclusione del percorso di studi.

A tutti i miei amici, beh qui ci vorrebbe troppo per ricordarli tutti, per adesso li ringrazio di tutto. Lascio ai festeggiamenti dopo laurea i ringraziamenti particolari.

Per tutti quelli che non sono stati elencati non si arrabbino di sicuro questo lavoro e anche merito vostro.

Prima di salutarci vorrei ringraziare Claudia, a cui dedico questo lavoro. La tua forza d'animo è stata fondamentale per arrivare in fondo.

Grazie a Tutti.