Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Implementazione di un typechecker statico e ottimizzazioni per un linguaggio di programmazione

Relatore
Silvia Crafa

Alessio Ferrarini (1223860)

Anno Accademico 2021/2022

Sommario

Questa relazione si pone l'obiettivo di illustrare l'attività di stage svolta dal laureando Alessio Ferrarini presso l'azienda Zucchetti Spa.

Il progetto consiste nello sviluppo di un typchecker statico, l'ottimizzazione e l'aggiunta di nuove feature per il linguaggio di programmazione Code Painter Language (CPL) sviluppato dall'azienda stessa.

Struttura del documento

Il documento è suddiviso in 6 capitoli:

1. Introduzione: Contiene una breve introduzione all'azienda Zucchetti Spa, la motivazione del progetto, obiettivi e vincoli progettuali.
2. Il linguaggio CPL: Fornisce una breve introduzione al linguaggio di programmazione in questione.
3. Sviluppo del typechecker statico: Delinea la progettazione e l'implementazione del typechecker statico per il linguaggio CPL.
4. Ottimizzazioni: Introduce le ottimizzazioni possibili grazie alle informazioni raccolte staticamente dal typechecker.
5. Nuove features: Descrive una serie di funzionalità introdotte all'interno del linguaggio per permettere di sfruttare a pieno il typechecker statico e la loro implementazione.
6. Valutazione retrospettiva: Racchiude tutte le considerazioni finali sul lavoro svolto.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 L'Azienda

Figura 1.1: Logo Zucchetti s.p.a.

Il progetto di stage si è svolto presso l'azienda Zucchetti S.p.A. con sede a Padova e appartenente al gruppo Zucchetti, leader indiscusso del settore dell'Information Technology (IT) con molteplici sedi distribuite non solo sul territorio nazionale ma anche resto del mondo.

La sede di Padova conta una decina di dipendenti specializzati in ambiti diversi, ma che si occupano principalmente della manutenzione di software che permette lo sviluppo automatizzato di applicativi.

1.2 Scopo del progetto

Il linguaggio di programmazione CPL è utilizzato ancora oggi all'interno della maggior parte dei prodotti di punta sviluppati da Zucchetti s.p.a., ma allo stato attuale alcune scelte progettuali ne hanno reso l'utilizzo non sempre naturale. In particolare, la sua natura di verificare i tipi durante l'esecuzione del programma e un sistema di tipi abbastanza primitivo hanno spinto gli sviluppatori alle prese con il linguaggio a scrivere codice contenente errori. Nel contesto di un linguaggio con una verifica dei tipi statica sarebbe stato possibile tranquillamente evitare.

Per rimediare a questa evenienza si possono sviluppare programmi esterni per verificare i tipi in programmi scritti in linguaggi privi di verifica statica dei tipi; questa idea non è di certo nuova, infatti sono già presenti sul mercato strumenti di questo tipo come Flow [1] sviluppato da Facebook e ideato per verificare staticamente i tipi nei programmi JavaScript o Psalm [2] per PHP sviluppato da Vimeo.

Inoltre si è deciso di arricchire il sistema di tipi di CPL introducendo nuovi concetti in grado di permettere agli sviluppatori di scrivere codice più sicuro, flessibile e robusto.

1.2.1 Obiettivi

Gli obiettivi del progetto sono i seguenti:

- Sviluppare un prodotto software in grado di verificare i tipi in modo statico all'interno dei programmi scritti in CPL.
- Sfruttare le nuove informazioni raccolte dal type checking statico per velocizzare l'esecuzione dei programmi CPL.
- Introdurre all'interno del linguaggio nuove funzionalità per poter permettere lo sviluppo di programmi in CPL, con l'aiuto del type checker statico, in modo agile e senza dover scendere a compromessi durante lo sviluppo.
- Mantenere sia compilatore che interprete del bytecode retro compatibili, in modo da poter eseguire vecchi programmi CPL anche sulle nuove versioni sviluppate.

1.2.2 Vincoli

1.2.2.1 Vincoli Tecnologici

1.2.2.1.1 Type checker statico Il type checker statico è stato sviluppato nel linguaggio CPL stesso. nonostante per questo tipo di strumenti spesso si preferiscano linguaggi di programmazione come `StandardML`, `Haskell` o altri linguaggi di natura funzionale e mi ritenga abbastanza competente per affrontare lo sviluppo del progetto in `Haskell`, il tutor interno ha spinto sullo sviluppo dello strumento nel linguaggio CPL.

Questo è dovuto al fatto che se il prodotto risultasse di qualità soddisfacente l'azienda sarebbe interessata a utilizzarlo, di conseguenza sarà necessario mantenerlo nel tempo. Inoltre, trovare personale in grado di lavorare su progetti scritti in `Haskell` risulta difficile nel mercato italiano.

Questo vincolo porterà degli ulteriori vantaggi che personalmente non avevo valutato:

- Ottimizzare il tempo necessario all'apprendimento del linguaggio CPL realizzando le strutture dati necessarie all'interno del progetto come esercizio.
- Avere un progetto scritto in CPL di moderate dimensioni di cui conoscono molto bene il codice. Questo ha permesso di eseguire su di esso alcuni test del type checker.

1.2.2.1.2 Compilatore e interprete CPL Per lo sviluppo di nuove funzionalità all'interno del linguaggio, è invece necessario interfacciarsi con il progetto preesistente. Sia l'interprete che il compilatore sono scritti in `C++` all'interno di una soluzione per l'IDE Visual Studio.

Nonostante queste limitazioni, mi è stata concessa la possibilità di lavorare alla compilazione dei vari componenti su sistemi operativi `GNU/Linux` tramite la creazione di un `Makefile`, e inoltre, di utilizzare uno standard `C++` più moderno come `C++20`.

Questo ha portato ad avere accesso alle nuove funzionalità del linguaggio `C++` durante lo sviluppo del progetto e a non essere vincolato all'IDE di Microsoft durante la fase di sviluppo e test.

1.2.2.2 Vincoli Temporali

Il progetto è stato svolto durante lo stage aziendale per un totale di trecentoventi ore distribuite in cinque giornate lavorative da otto ore.

Questo ha richiesto una particolare elasticità durante lo sviluppo del prodotto. In particolare, durante il lavoro svolto su software già esistente come il compilatore e l'interprete non è stato possibile eseguire un refactor completo del progetto prima di implementare le nuove funzionalità.

Capitolo 2

Il linguaggio CPL

Il linguaggio Code Painter Language (CPL) nasce nel 1998 e prende ispirazione dai linguaggi di programmazione Python ed Eiffel. Dal primo eredita soprattutto la sintassi, mentre dal secondo i costrutti della programmazione orientata agli oggetti e il concetto di design by contract (analizzato in 2.1.3).

Allo stato attuale CPL è ancora utilizzato all'interno di Zucchetti in tutti i software sviluppati dall'azienda.

CPL ha le seguenti caratteristiche:

- È un linguaggio di scripting.
- È orientato agli oggetti.
- Tutto è un oggetto.
- Supporta l'ereditarietà multipla.
- Ha una gestione della memoria automatica tramite reference counting.

Gli aspetti riguardanti la programmazione orientata a oggetti saranno approfonditi nel paragrafo 2.1.2.

Il modello di memoria sarà analizzato nel paragrafo 2.1.7.

La sintassi di CPL è esprimibile con una Deterministic Context-free Grammar (DCFG). L'intera definizione si trova nell'appendice A e sarà analizzata nel paragrafo 2.1.6.

2.1 Analisi del linguaggio

Un programma CPL non ha istruzioni d'inizio e di fine. Il programma viene eseguito dalla prima istruzione all'ultima in modo sequenziale. Un'istruzione termina quando è presente il carattere di nuova riga, a meno che non ci si trovi all'interno di una coppia di parentesi tonde.

```
1 ? "Hello world"
```

Listing 2.1: Hello world in CPL

2.1.1 Costrutti base

Le variabili si dichiarano con la parola chiave `var` seguita dal tipo e il nome. Una variabile di default è inizializzata con il valore `nil` (equivalente a `null` in `Java` per esempio), oppure può essere inizializzata direttamente alla dichiarazione.

```
var int i
? i -- <nil>
i := 42
-- i := "Hello World" -> Type error (string non e' assegnabile ad int)
? i -- 42
var str j := "Test"
? j -- Test
```

Listing 2.2: Hello world in CPL

Funzioni e procedure (funzioni senza valore di ritorno) si dichiarano con le rispettive parole chiave `func` e `proc` seguite dal tipo di ritorno nel caso delle funzioni, nome e la lista di argomenti. Come in Eiffel, non esiste una istruzione specifica per il return, ma si assegna alla variabile `result`. Inoltre, è possibile il passaggio di argomenti per riferimento tramite l'operatore `@`, il riordinamento degli argomenti tramite nome e valori di default.

```
func int add(int a, int b)
  result := a + b
end

? add(10, 11) -- 21

proc printDiv(int a := 0, int b := 1)
  ? a / b
end

printDiv(a := 10) -- 10.0000000000

proc mutate(int@ a)
  a := 10
end

var int m := 0
mutate(@m)
? a -- 10
```

Listing 2.3: Esempi di chiamata a funzione in CPL

Sono supportate le più classiche strutture di control flow come `if`, `for`, `while`, `switch case`, `for each` e `try catch`.

```
if 1 + 1 <> 3
  ? "3"
else
  ? "Non 3" -- Non 3
end

var int i
for each i in [1, 2, 3]
```

```
    ? i -- 1, 2, 3
  end

  try
    raise(1)
    ? "done"
  catch(int x)
    ? x -- 1
  finally
    ? "finally" -- "finally"
  end
```

Listing 2.4: Control flow in CPL

2.1.2 Programmazione orientata agli oggetti

Come in Eiffel, il linguaggio CPL utilizza paradigma il Object Oriented Programming (OOP) basato sul concetto di classi (come fanno ad esempio Java e C++) piuttosto che basarsi sul concetto di prototipo (come fanno ad esempio JavasScript e IO).

Una definizione di classe in CPL avviene all'interno di un blocco delimitato dalle keyword `class` e `end`. Una classe al suo interno può dichiarare campi e metodi.

L'uguaglianza tra tipi è di tipo nominale e non strutturale, cioè l'uguaglianza tra tipi è basata sul concetto di nome: due tipi sono lo stesso tipo se solo se hanno lo stesso nome. Il concetto di uguaglianza strutturale invece si basa totalmente sulla struttura (metodi e campi) di essi [3].

```
class Person
  var str Name

  proc Init(str n)
    Name := n
  end

  proc SayHi()
    ? "Hi! My name is ", Name
  end
end

var Person me := Person("Alessio")
me.SayHi() -- Hi! My name is Alessio
```

Listing 2.5: Definizione di classe in CPL

CPL mette a disposizione il meccanismo dell'ereditarietà multipla. Inoltre, è possibile definire metodi astratti.

```
class A
  proc a()
    ? "a"
  end
end

class B
```

```
8     proc b() abstract
9   end
10
11  class C(A, B)
12    proc b()
13      ? "b"
14    end
15  end
16
17  var C c := C()
18  var A a := c
19  var B b := c
```

Listing 2.6: Ereditarietà in CPL

Alla linea 11 dichiariamo che la classe `C` eredita dalle classi `A` e `B`, alla linea 17 vediamo che un' istanza della classe `C` è assegnabile a una variabile di tipo `C` mentre alle righe 18 e 19 vediamo che è anche assegnabile a variabili di tipo `A` e `B`.

Il subtyping segue le regole comuni alla maggior parte dei linguaggi orientati agli oggetti.

Si indica con il simbolo <: la relazione di subtyping e $A <: B$ si legge "A è sottotipo di B".

$$C <: C$$

Un tipo C è sempre sottotipo di se stesso, riflessività.

$$\frac{C <: D \qquad D <: E}{C <: E}$$

Il subtyping è transitivo.

$$\frac{\text{CT(C)} = \text{class C(D,E,...)} \ ...}{C <: D, C <: E, ...}$$

Un tipo C è sotto tipo dei tipi da cui eredita.

2.1.3 Design by contract

ome accennato nell'introduzione del capitolo 2 al linguaggio, CPL eredita da Eiffel il concetto di design by contract, cioè la possibilità di poter definire pre e post condizioni per metodi e procedure così da garantire un ulteriore livello di correttezza durante lo sviluppo di un prodotto software.

Sono messe a disposizione tre keyword:

- `pre`: Per indicare le pre-condizioni alla chiamata di una procedura.
- `post`: Per indicare le post-condizioni da validare alla fine della chiamata di una procedura.
- `old`: Per ottenere il valore di una variabile prima dell'esecuzione della procedura all'interno di una post-condizione.

```
class Clock
  var int Ticks := 0
  var bool Started := false

  proc Start()
    pre not Started

    Started := true

    post Started
  end

  proc Tick()
    pre Started

    Ticks := Ticks + 1

    post old(Ticks) + 1 = Ticks
  end
end

var Clock c := Clock()
-- c.Tick(): Errore -> pre-condizione violata
c.Start()
-- c.Start(): Errore -> pre-condizione violata
c.Tick()
```

Listing 2.7: Esempio di DbC in CPL

2.1.4 Type checking

Non viene eseguito nessun controllo per verificare la validità dei tipi prima dell'esecuzione, ma vengono verificati solo durante l'esecuzione del programma. Questo implica che percorsi di programma, che non vengono attraversati durante l'esecuzione, possono contenere errori di tipo e nominare variabili non esistenti.

```
if false
  var int i := "A string"
  ? k + m
  i.NotARealMethod()
end
```

Listing 2.8: Esempio di errori di tipo non catturati da CPL

È evidente che sia un grande problema per linguaggio, in quanto, per validare totalmente il programma, è necessario eseguirlo e percorre tutte le combinazioni possibili dei vari rami di esecuzioni, che non è sempre possibile.

Inoltre, non vengono effettuati controlli riguardati le classi che la maggior parte dei linguaggi OOP mettono a disposizione, come ad esempio:

- È possibile istanziare classi astratte.
- Non vengono eseguiti controlli sulla firma dei metodi quando viene eseguito un override.

- Non vengono eseguiti controlli sui metodi chiamati per verificare che appartengano al tipo della classe in oggetto.

```
1  class A
2    proc a(int i)
3    end
4  end
5
6  class B(A)
7    proc a(str i)
8    end
9
10   proc c()
11   end
12
13   proc b() abstract
14 end
15
16 var A a := B()
17 -- a.a(42) Errore: richiesto str ma ricevuto int
18 a.a("string")
19 a.c()
```

Listing 2.9: Esempio di incongruenze a livello di classe non catturate da CPL

A riga 19 possiamo vedere che nonostante il nome `a` introdotto a riga `16` sia legato al tipo della classe `A` non viene eseguito nessun controllo statico prima dell'esecuzione per verificare che vengano chiamati solo metodi della classe `A`, nel caso il valore assegnato alla variabile possegga quel metodo, appunto come avviene a riga 19, non verrà generato nessun errore. Inoltre, come si evince da riga `13` la classe `B` ha un metodo astratto quindi non dovrebbe essere istanziabile come classe ma non venendo eseguito nessun controllo come possiamo vedere a riga `16`, nel caso venga chiamato un metodo che di fatto è astratto verrà generato un errore a runtime.

2.1.5 Scoping

In CPL il concetto di visibilità delle variabili e dei metodi è presente solo a 3 livelli:

- Globale
- Di classe
- Locale alla procedura

Questo rende il linguaggio molto poco ergonomico, ad esempio non è possibile dichiarare variabili all'interno di costrutti iterativi senza incorrere in un errore runtime se avviene più di una iterazione.

```
1 var int i
2 for each i in [1, 2, 3]
3   var int succ := i + 1 -- Alla seconda iterazione verra' dato errore in
4                         -- quanto la variabile i risulta gia' essere
5                         -- stata dichiarata
6   ? succ
7 end
```

Listing 2.10: Dichiarazione di varaibile all'interno di un ciclo

Infatti, risulta comune trovare all'interno dei vari programmi che mi sono stati messi a disposizione dall'azienda che è pratica comune quella di dichiarare tutte le variabili all'inizio della funzione, creando uno stile di programmazione che ricorda quello tipico dei programmi scritti in `C` prima dello standard `C99`.

Comunque è possibile dichiarare variabili con lo stesso nome a patto che esse si trovino in ambienti diversi.

```
1 var int i := 11
2
3 proc a()
4   var int i := 12
5   ? i
6 end
7
8 a() -- 12
9 ? i -- 11
```

Listing 2.11: Shadowing di variabili

2.1.6 Grammatica

Come descritto all'inizio del capitolo 2 la grammatica che descrive CPL è DCFG questo ci da la possibilità di utilizzare un parser di tipo `LR1`.

L'utilizzo di un parser di tipo `LR1` permette di effettuare il parsing senza effettuare operazioni di back tracking e di generare man mano le istruzioni che poi andranno ed essere eseguite.

Questo è stato deciso durante lo sviluppo per permettere la creazione di un Read–Eval–Print Loop (REPL) in grado di valutare parti del programma e caricare moduli in modo dinamico, così da permettere l'esecuzione interattiva durante la fase di sviluppo del software.

Inoltre, questo porta un ulteriore vantaggio, è possibile creare un programma in grado di trasformare una descrizione della grammatica con una sintassi simile a quella Extended Backus–Naur Form (EBNF) in una classe `C++` in grado di effettuare il parsing `LR1` di essa (dato che EBNF è in grado di descrivere grammatiche più potenti di DCFG il software si occupa anche di verificare che rispetti i vincoli imposti dal parsing di tipo `LR1`).

La descrizione completa della grammatica in EBNF può essere trovata nell'appendice A.

2.1.7 Modello di memoria

In CPL la gestione della memoria è automatica tramite reference counting, a ogni istanza di ogni classe viene associato un contatore il quale tiene traccia del numero di riferimenti ad essa.

Quando il contatore raggiunge lo 0, non sono più presenti riferimenti alla istanza quindi risulta possibile eliminarla in modo sicuro.

Questo tipo di gestione di memoria automatica porta ad una serie di svantaggi e vantaggi rispetto ad il più classico garbage collector.

Il principale vantaggio è il fatto che le prestazioni sono prevedibili in quanto la memoria viene liberata appena possibile e soprattutto non sono presenti interruzioni durante l'esecuzione dei vari thread.

Ma porta anche una serie di svantaggi:

- Come soluzione risulta essere meno performante soprattutto in contesti concorrenti, risulta necessario proteggere il contatore.
- È necessario l'utilizzo di weak reference quando ci sono riferimenti circolari tra oggetti.

Tutti gli oggetti in CPL vengono allocati sull'heap, di conseguenza, sullo stack sono presenti solo riferimenti ad essi, seguendo lo stesso approccio preso da `Java` se non consideriamo i tipi primitivi.

2.2 Esecuzione

Un programma CPL non viene eseguito così com'è, prima di tutto viene compilato in un formato di `bytecode` chiamato `pcode`, il quale poi viene eseguito dalla Virtual Machine (VM) di CPL.

Figura 2.1: Schema del modello di esecuzione

La VM di CPL è una stack machine, essa opera su uno stack in cui vengono posizionati tutti i valori temporanei su cui poi andrà ad eseguire le varie operazioni di trasformazione.

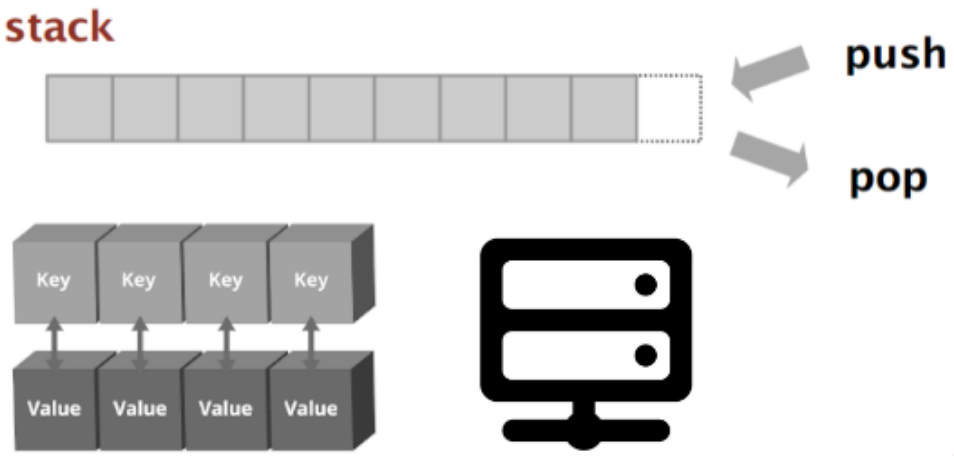


Figura 2.2: Modello di memoria

Lo stesso modello di esecuzione è utilizzato anche dalle VM di altri linguaggi di programmazione come ad esempio dalla Java Virtual Machine (JVM) usata dai linguaggi `Java`, `Kotlin` e `Scala`.

Il bytecode della virtual machine è formato da un totale di 117 operazioni di cui alcune di esse accettano un argomento il quale viene codificato come stringa `UTF-8` dopo l'opcode.

I vari opcode si possono suddividere il più categorie distinte:

- Operatori come ad esempio `ADD` e `SUB`.
- Operazioni sulla memoria come `LOAD` e `STORE`.
- Operazioni sulle classi come `NEW` e `SUPER`.
- Control flow come `JUMP` e `LABEL`.
- Valori letterali come `INTCONST` e `STRCONST`.
- Tipi come `BEGINTYPE` e `TYPE`.
- Dichiarazioni come `DECLCLASS` e `STARTPROC`.
- Speciali come `IMPORT`.
- Di debug come `BREAKPOINT` e `LINE`.

Una lista delle istruzioni è disponibile nell'appendice B.

Una particolarità di questo bytecode è che contiene informazioni sui tipi assegnati alle variabili, questo ovviamente a causa dei controlli che vengono eseguiti a runtime spiegati in 2.1.4, questo verrà sfruttato durante lo sviluppo del typechecker statico.

Capitolo 3

Sviluppo del typechecker statico

Come spiegato nel capitolo paragrafo 2.1.4 il controllo di tipi effettuato a runtime risulta essere problematico, per questo si è deciso di sviluppare un programma esterno per verificare la correttezza dell'utilizzo dei tipi nei programmi CPL.

Per lo sviluppo del typechecker ho deciso di operare direttamente sul `pcode`, in quanto:

- Risulta essere molto più semplice da manipolare del linguaggio di superficie
- Mantiene tutte le informazioni di tipo

Il typechecking di un programma CPL si può suddividere in fasi cinque principali:

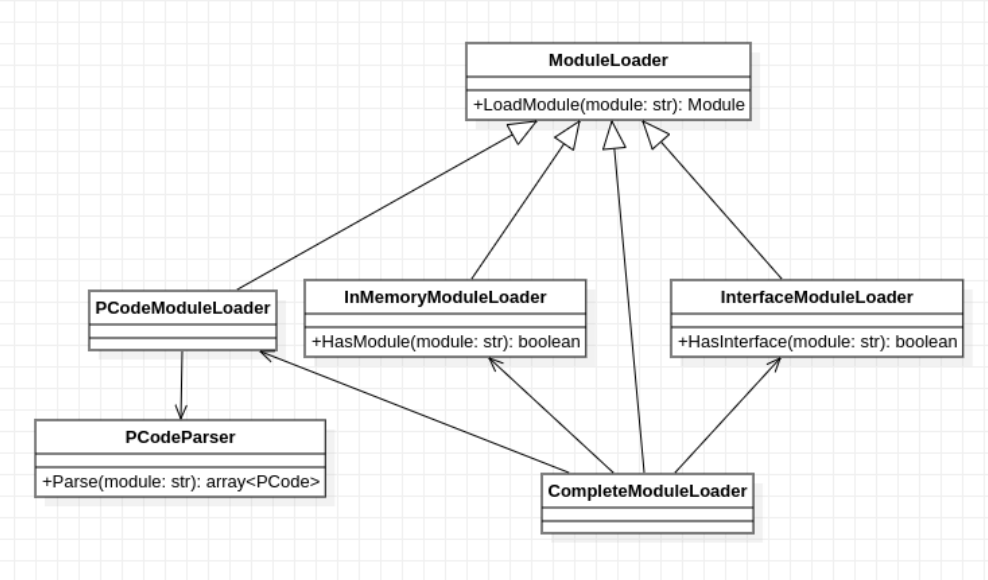
1. Caricamento e parsing del `pcode`.
2. Caricamento dei moduli importati.
3. Estrazione delle definizioni di classe e funzione dal `pcode`.
4. Controllo delle classi per verificarne la correttezza (Ad esempio non ci siano estensioni circolari, validità dell'override dei metodi).
5. Controllo statico dei tipi nelle espressioni.

3.1 Caricamento di un modulo

Per alleggerire l'esecuzione del typechecking e permettere il riferimento circolare tra moduli differenti ho deciso di definire tre differenti per il caricamento di un modulo:

- Da file CPL.
- Da file di definizioni, nel caso in cui si sia già eseguito il controllo di tipi del modulo in precedenza.
- Da memoria in caso il modulo sia già stato caricato durante il typecheck.

3.1.1 Architettura ad alto livello



Per rendere uniforme il caricamento dei moduli è stata definita una classe con il metodo astratto `LoadModule`, questa è estesa dalle classi:

- `PCodeModuleLoader`: Caricamento del modulo da file CPL.
- `InMemoryModuleLoader`: Caricamento di un modulo caricato in precedenza.
- `InterfaceModuleLoader`: Caricamento di un modulo tramite file interfaccia.
- `CompleteModuleLoader`: Tramite dependency injection astrae il caricamento di un modulo.

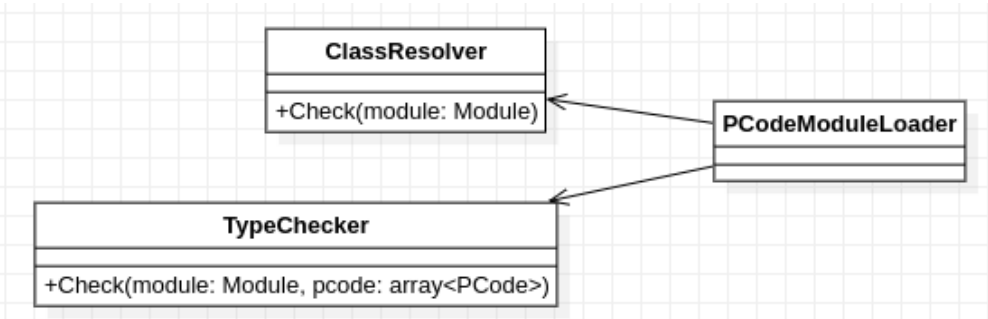
3.2 Type checking di un modulo

Il type checking di un modulo, invece, può essere suddiviso in due parti:

1. Controllo delle classi per verificarne la correttezza.
2. Controllo statico dei tipi nelle espressioni.

Tutto questo avviene solamente quando il modulo viene caricato da file CPL.

3.2.1 Architettura ad alto livello



Tutte queste operazioni sono suddivise in 2 classi differenti:

- `ClassResolver`: Controllo delle classi per verificarne la correttezza.
  - Controllo assenza di estensioni cicliche.
  - Controllo metodi astratti.
  - Controllo degli override dei metodi.
  - Controllo del problema del diamante.
- `TypeChecker`: Controllo dei tipi delle espressioni.

3.2.2 ClassResolver

3.2.2.1 Estensioni cicliche

Una definizione di subtyping circolare come: $A <: C \ \wedge \ C <: A$ se assumiamo $C \neq A$ risulta essere paradossale, ad esempio non possiamo definire concretamente l'ordine di ricerca di un metodo.

Inoltre, a causa regola di transitività definita nel paragrafo 2.1.2, dobbiamo verificare che non avvenga nemmeno in modo indiretto.

In sostanza verificare quindi che se il tipo A è sottotipo di C, allora C non sia sottotipo di A.

$$A <: C \rightarrow C \not<: A$$

Di conseguenza il codice per implementare questo controllo segue direttamente dalla definizione di transitività e dalla definizione precedente:

```
1  -- src/types/types.cpl
2  class Clazz
3  ...
4  func bool IsSub(Clazz other)
5    if Name() = other.Name() or other.IsAny()
6      result := true
7    else
8      result := false
9      var Clazz super
10     for each super in superClasses()
11       if IsSub(super)
12         result := true
13       end
14     end
15   end
16 end
17 ...
18 end
19 -- src/classresolver/classresolver.cpl
20 class ClassResolver
21 ...
22   proc CheckClassSuper(Clazz clazz)
23     var Clazz super
24     for each super in clazz.SuperClasses()
25       if super.IsSub(clazz)
26         Error(clazz.Name() + " cannot inherit from " + super.Name() + "
               since it would result in a loop", true)
27       end
28     end
29   end
```

```
30  ...
31  end
```

Listing 3.1: Controllo estensioni circolari

Riga 4 implementa la regola della riflessività mentre riga 14 implementa la regola della transività. Di conseguenza riga 25 verifica che non ci sia eredità circolare anche nel caso questa sia transitiva.

3.2.2.2 Controllo metodi astratti

Un altro controllo da eseguire è quello d'impedire l'istanziazione di classi con metodi astratti che non sono stati ridefiniti.

Definiamo quindi l'insieme dei metodi astratti della classe C chiamato $abstract(C)$ come l'insieme dei metodi astratti definiti in C unito all'insieme dei metodi astratti definiti dalle sue super classi che non sono stati definiti all'interno di C.

$$\frac{\text{class C(D, E, ...) ... } \overline{func\ T\ x(...)\ abstract} \text{ ... end} \qquad \text{d} = \text{abstract(D)} \qquad \text{e} = \text{abstract(E)}}{\text{abstract(C)} = \overline{x} \cup ((d \cup e \cup ...) \setminus defs(C))}$$

E appunto, come detto in precedenza, definiamo una classe come astratta se l'insieme dei suoi metodi astratti non è vuoto.

$$\frac{abstract(C) \neq \emptyset}{\text{isAbstract(C)}}$$

Anche in questo questo le regole d'inferenza si traducono facilmente in codice:

```
1   -- src/classresolver/classresolver.cpl
2   class ClassResolver
3   ...
4     proc UpdateAbstract(Clazz clazz)
5       if clazz.AbstractMethods().Len <> 0
6         module.DeleteFunction(clazz.Name())
7       end
8     end
9   ...
10  end
11
12  -- src/types/types.cpl
13  class Clazz
14  ...
15    func array[Function] AbstractMethods()
16      result := []
17      var Function method
18      for each method in methods
19        if method.IsAbstract()
20          result.Append(method)
21        end
22      end
23
24      var Clazz super
25      for each super in SuperClasses()
```

```
26        for each method in super.AbstractMethods()
27            if not HasMethod(method.Name())
28              result.Append(method)
29            end
30        end
31      end
32    end
33  ...
34  end
```

Listing 3.2: Controllo metodi astratti

A riga `5` cancelliamo dal modulo il costruttore di una classe nel caso essa sia astratta in questo caso come descritto dalla seconda regola d'inferenza in 3.2.2.2, invece dalla riga `15` viene implementata la prima regola d'inferenza.

3.2.2.3 Controllo override

Prendiamo come esempio il seguente programma:

```
1  class A; end
2  class B(A); end
3  class C(B); end
4
5  class Aleph
6    func B f(B i); end
7  end
8
9  class Aleph2(Aleph)
10   func B f(B i); end -- 1
11   func C f(C i); end -- 3
12   func C f(A i); end -- 5
13 end
```

Listing 3.3: Override in CPL

La classe `Aleph2` eredita dalla classe `Aleph`, quale delle 5 definizioni è un override valido del metodo f descritto dalla classe `Aleph`?

Per quanto riguarda l'override 1 non ci sono dubbi, ma per 3 e 5 la questione cambiano, qual è la regola corretta per definire il subtyping nel caso dell' override? Dipende da come definiamo il subtyping per $\rightarrow$:

$$3\ \frac{A' <: A \qquad B' <: B}{A' \rightarrow B' <: A \rightarrow B} \qquad\qquad 5\ \frac{A <: A' \qquad B' <: B}{A' \rightarrow B' <: A \rightarrow B}$$

La regola d'inferenza 3 stabilisce che l'argomento è covariante mentre 5 stabilisce che è contravariante e di conseguenza la prima che $\rightarrow$ è un profuntore mentre 5 che è un bifuntore. (Infatti in teoria delle categorie $\rightarrow$ è un profuntore).

L'approccio della maggior parte dei linguaggi di programmazione è quello della regola 5, in quanto esso permette di rispettare il principio di sostituzione di Liskov e di conseguenza poter avere un typesystem sound.

Invece l'approccio 3 è stato seguito dal linguaggio di programmazione Eiffel e di conseguenza da richiesta del tutor interno CPL, questo viola totalmente il principio di sostituzione di Liskov rendendo il typesystem unsound.

```
class A; end
class B(A); end
class C(B)
  proc c(); end
end
class Aleph1
  func B f (B i); end
end

class Aleph2(Aleph)
  func C f (C c);
    c.c()
  end
end

var Aleph1 a := Aleph2()
a.f(B())
```

Listing 3.4: CAT-Call in CPL

Questo nel linguaggio tecnico di Eiffel viene chiamato Changing Availability or Type (CAT) Call e risulta in un errore runtime, questa incongruenza con i risultati della teoria dei linguaggi di programmazione viene difesa dal designer principale di Eiffel in quanto è necessaria per il design by contract per come ideato da Bertrand Meyer [4].

E anche in questo caso è facile tradurre la regola in codice:

```
-- src/classresolver/classresolver.cpl
class ClassResolver.
...
  var bool ARGUMENT_COVARIANCE := true
...
  proc CheckOverride(Function method, Function superMethod)
    var array[Variable] args := method.Arguments()
    var array[Variable] superArgs := superMethod.Arguments()

    if (args.Len <> superArgs.parameters.Len)
        Error("Cannot override method " + method.Name() + " with a different
            number of arguments", true)
    else
        var int i
        for i := 0 to aArgs.Len - 1;
            var Clazz a := args[i].Type
            var Clazz b := superArgs[i].Type

            var bool respectVariance
            if ARGUMENT_COVARIANCE
                respectVariance := a.IsSub(b)
            else
                respectVariance := b.IsSub(a)
            end
```

```
24
25              if not respectVariance
26                Error("Argument " + a.Name() + " of method " + method.Name() +
                      " does not respect the variance rules", true)
27              end
28          end
29          if not method.ReturnType().IsSub(superMethod.ReturnType())
30            Error("The return type of " + method.Name() + " does not respect
                    the override rules", true)
31          end
32      end
33  ...
34  end
```

Listing 3.5: Controllo metodi astratti

Comunque data la controversialità dell'argomento ho deciso di permettere la modifica e il passaggio alla regola 3 semplicemente cambiando il valore della variabile `ARGUMENT_COVARIANCE` a riga 4.

3.2.2.4 Controllo problema del diamante

Altro aspetto da controllare staticamente è quello del problema del diamante, prendiamo come esempio il seguente programma CPL:

```
1  class A
2    proc a() abstract
3  end
4
5  class B(A)
6    proc a()
7      ? "B::a"
8    end
9  end
10
11 class C(A)
12   proc a()
13     ? "C::a"
14   end
15 end
16
17 class D(C, B)
18 end
19
20 D().a()
```

Listing 3.6: Problema del diamante

Quale metodo `a` dovrebbe essere chiamato?

Linguaggi come `C++` o `Eiffel` eseguendo controlli prima dell'esecuzione del programma, verificando non sia presente questo tipo di ambiguità.

Invece, linguaggi come `Python` decidono di stabilire un ordine arbitrario di priorità, però come soluzione risulta essere particolarmente soggetta ad errori.

Quindi si è deciso di procedere con la prima soluzione, quella di generare un errore e di forzare l'utente a ridefinire il metodo soggetto all'ambiguità.

Questo è stato implementato verificando che i nomi in comune tra le dirette super-classi di ogni classe nel caso condividano nomi essi abbiano ricevuto una ridefinizione nella classe in oggetto.

```
-- src/classresolver/classresolver.cpl
class ClassResolver
...
  proc DiamondCheck(Clazz clazz)
    var array[str] thisNames := clazz.ThisNames()
    var array[str] superNames := []

    var Clazz super
    for each super in clazz.SuperClasses()
      var array[str] currentNames := super.ThisNames()

      var array[str] intersect := currentNames * superNames
      if intersect.Len > 0 and thisNames.IndexOf(intersect[0]) = nil
          Error("Property " + intersect[0] + " inherited in class " +
              clazz.Name + " is subject to the diamond problem and should be
              redefined", false)
      else
        superNames := superNames + currentNames
      end
    end
  end
...
end
```

Listing 3.7: Controllo del problema del diamante

3.2.3 Typecheck

Come descritto in 2.2 il bytecode generato dal compilatore è fatto per essere interpretato da una stack machine.

Per poter verificare la correttezza delle operazioni descritte dal `p-code` quindi risulta necessario simulare l'esecuzione, di conseguenza avremo:

- Uno stack che invece di contenere i valori generati dalla stackmachine contiene i tipi che verranno generati.
- Gli ambienti (globali, di classe e locali alle funzioni) che invece di collegare i nomi ai loro valori li collega al loro tipo.
- Program counter che salvi la posizione corrente all'interno del `p-code`.

Inoltre dato che il linguaggio di superficie, CPL, non è a conoscenza del modello di esecuzione sottostante, alla fine di ogni costrutto condizionale, ciclo o try catch possiamo assumere che lo stack sia vuoto, di conseguenza, il typechecking può essere effettuato attraversando in ordine tutte le istruzioni del `p-code` senza seguire i vari salti condizionati.

```
-- src/typechecker/typechecker.cpl
...
```

```
class TypeChecker
    var TypeCheckerScope scope := TypeCheckerScope()
    var ClassStack stack := ClassStack()
    var int programCounter := 0
...
end
```

Listing 3.8: Stato del typechecker

A questo punto le operazioni necessarie per verificare la correttezza del programma sono quelle di visitare ogni definizione (che sia di variabile, funzione o classe) nel modulo corrente, ottenerne la posizione all'interno del `p-code` in cui ne avviene la dichiarazione e simulare l'esecuzione dell'interprete fino alla fine di essa.

```
-- src/typechecker/typechecker.cpl
class TypeChecker
...
    proc Expression(str stop, bool keep := false)
        while Current().Name <> stop and Error = nil
          switch Current().Name
        ...
          case "Intconst"
            stack.Push(IntClass)
        ...
          case "This"
            var Clazz current := scope.CurrentClass()
            if current <> nil
              stack.Push(current)
            else
              Error("Cannot refer to 'this' outside of a class definition")
            end
        ...
          case "StoreParam"
            var Clazz subject := stack.Pop()
            storedParameters.Insert(Current().Argument, subject)
        ...
          Advance()
        end
    end
...
end
```

Listing 3.9: Stato del typechecker

3.3 Test

3.3.1 Test automatici

Software di questo tipo sono facilmente soggetti a regressione, infatti, non è possibile verificare manualmente tutti i modi in cui alcune funzionalità all'interno del linguaggio interagiscono tra di loro.

Per questo è risultato necessario sviluppare una collezione di test da eseguire in automatico, questi sono composti da una serie di programmi CPL e un file di testo contenente l'output atteso dall'esecuzione del typechecker su di essi.

Allo stato attuale sono presenti `90` test diversi, tutti ideati per verificare la correttezza di una singola funzionalità o come essa interagisce con altre.

Tutti questi test sono eseguiti in automatico da uno script python dando la possibilità all'utente anche di filtrare i test in base alla tipologia e alla feature che viene testata.

3.3.2 Test su programmi

Oltre ai test automatici è importante verificare anche come il typechecker si comporta in contesti reali.

Il typechecker oltre a riuscire a verificare se stesso, è in grado di essere eseguito sui vari progetti software scritti da Zucchetti ed è riuscito a identificare molteplici errori in software presente in produzione, ad esempio:

- Utilizzo di variabili non definite nello scope corrente
- Utilizzo di metodi non presenti in determinate classi
- Istanziazione di classi astratte
- Override non validi

Capitolo 4

Ottimizzazioni

Tutte le informazioni catturate dal typechecker statico possono permettere la creazione di varie ottimizzazioni.

4.1 Disabilitazione dei controlli di tipo runtime

Nonostante i controlli di tipo eseguiti a runtime del linguaggio siano pochi sono anche inefficienti.

Siccome il typechecker è in grado di effettuare tutti questi controlli ahead-of-time (AOT) è possibile permettere alla macchina virtuale di saltare tutte le istruzioni riguardanti i tipi.

Dal punto di vista implementativo risulta essere un cambiamento molto semplice, infatti, risulterà necessario solo come detto prima evitare l'interpretazione di tutte le istruzioni ed evitare tutti i controlli di tipo come ad esempio il controllo durante l'assegnazione a variabile.

```
1  // cplinter.cpp
2  next_idx = GetExecutablePCode(exec_mod, rPC+1);
3  switch (next_idx) {
4  ...
5    case BEGINTYPE:
6      if (!CLVM().check_type) {
7        SkipUntil(ENDTYPE);
8      }
9    break;
10 ...
11 }
12
13 // cplvar.cpp
14 void CPL_Var::Set(CPL_Object* obj) {
15     if (!CLVM().check_type) {
16       value = obj;
17       return;
18     }
19     ...
```

```
20      if (!obj->IsA(type)) {
21          const auto type_name = type->Dump();
22          const auto obj_name = obj->GetClass()->Dump();
23          const auto msg = format(_T("Type error:\nvar = %s\n") \
24                                  _T("expected type = %s\nwrong type = %s"),
25                                  CPL::Get(name), type_name.ToT(),
                                        obj_name.ToT());
26          CPL_ThrowTypeException(msg);
27      }
28      ...
29  }
```

Listing 4.1: Disabilitazione dei controlli

4.1.1 Performance

Nonostante questo tipo di cambiamento possa risultare minimo effettivamente ha portato a una riduzione del tempo di esecuzione di vari programmi CPL di circa il 20%.

Arrivando addirittura ad un miglioramento del 30% in benchmark specifici.

4.2 Ottimizzazione del meccanismo di chiamata a funzione

Come spiegato nel paragrafo 2.1.1 CPL permette di eseguire chiamate a funzione:

- Nominando i nomi degli argomenti e cambiandone l'ordine
- Utilizzando valori di default

Queste funzionalità sono presenti anche in altri linguaggi di programmazione come ad esempio Scala ma il loro utilizzo non viene penalizzato durante l'esecuzione del programma come invece avviene il `CPL`.

Il compilatore di `Scala` conoscendo staticamente i nomi e le firme delle funzioni durante la compilazione riordina gli argomenti e inserisce i valori nei punti di chiamata.

Invece, per la natura dinamica di `CPL` nè il compilatore nè l'interprete conoscono la firma della funzione, se non al momento stesso della chiamata. Di conseguenza questo tipo di operazione viene eseguita durante l'esecuzione del programma, ogni volta che viene eseguita una chiamata di funzione viene creata una `Hashmap` contenente tutti i parametri che poi verranno riordinati successivamente.

Adesso il typechecker statico ha informazioni sulla firma della funzione che viene chiamata, questo permetterebbe, apportando alcune modifiche al typechecker, di compire la stessa operazione che fa il compilatore di `Scala`, rimuovendo totalmente la penalità data dal riordinare gli argomenti al momento dell'esecuzione del programma.

4.2.1 Analisi

Data la complessità della funzionalità dal punto di vista implementativo si è deciso prima di verificare che il compromesso tra complessità implementativa dell'ottimizzazione e performance guadagnate fosse favorevole.

4.2.1.1 Costruzione di un benchmark

Per verificare in modo empirico la quantità di tempo di esecuzione passata a preparare la chiamata di una funzione è risultato necessario sviluppare un programma CPL in grado di generarne un numero importante, così da poter utilizzare un profiler ed identificare con precisione la causa del rallentamento.

```
1 func int fib(int n)
2   if n <= 1
3     result := n
4   else
5     result := fib(n - 1) + fib(n - 2)
6   end
7 end
8
9 ? fib(30)
```

Listing 4.2: Calcolo del fattoriale

L'implementazione del calcolo del fattoriale in questo modo andrà a compiere 832040 chiamate della funzione `fib`.

4.2.1.2 Analisi delle chiamate

Ora che abbiamo un benchmark possiamo usare un `gperf` un (profiler per programmi scritti in `C++` e compilati con il compilatore `gcc`) possiamo vedere quali sono le funzioni in cui l'interprete passa più tempo e di conseguenza individuare il collo di bottiglia.

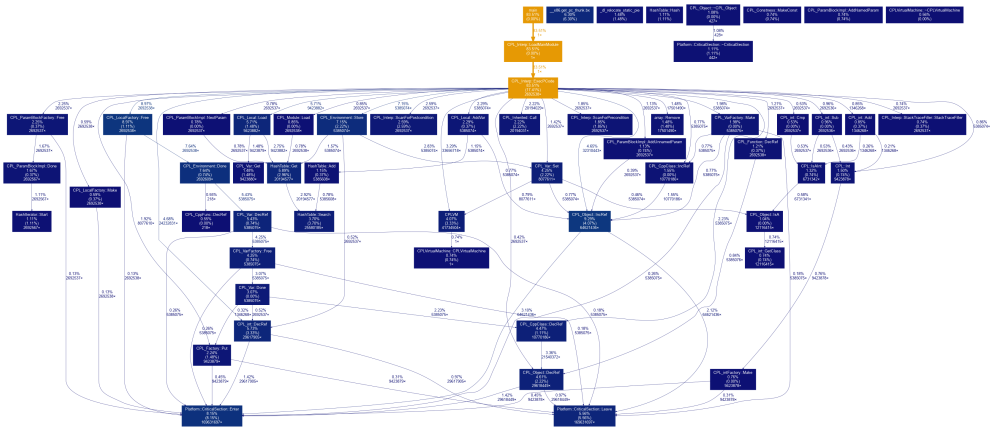
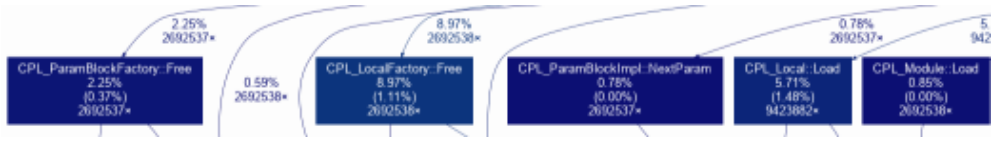


Figura 4.1: Grafo delle chiamate dell'interprete

In particolare possiamo vedere l'impatto della `Hashmap` dei parametri all'interno della classe `CPL_ParamBlockFactory` e `CPL_ParamBlockImpl`.



Però esso corrisponde solo a circa il 3% del tempo di esecuzione mentre l'8% del tempo è passato solo nella distruzione dell'ambiente locale a causa del reference counting.

Dopo una discussione con il tutor interno, per sostituire il reference counting si dovrebbe passare a un garbage collector ma questo richiederebbe una grande quantità di lavoro non compatibile con i tempi di stage mantenendo il resto degli obiettivi. Inoltre, l'implementazione potrebbe rompere la retrocompatibilità in caso di codice dipendente dalla distruzione di oggetti.

Infine il tutori aziendale ha richiesto di proseguire con l'implementazione dell'ottimizzazione, verificando se si potesse ottenere solo una riduzione del 3% del tempo di esecuzione.

4.2.2 L'implementazione

Come discusso nel paragrafo 4.2.1.2 non è detto che l'ottimizzazione sia effettivamente conveniente in realazione complessità implementativa, di conseguenza si è deciso di implementarla in modo non completo come prova.

È stato necessario quindi:

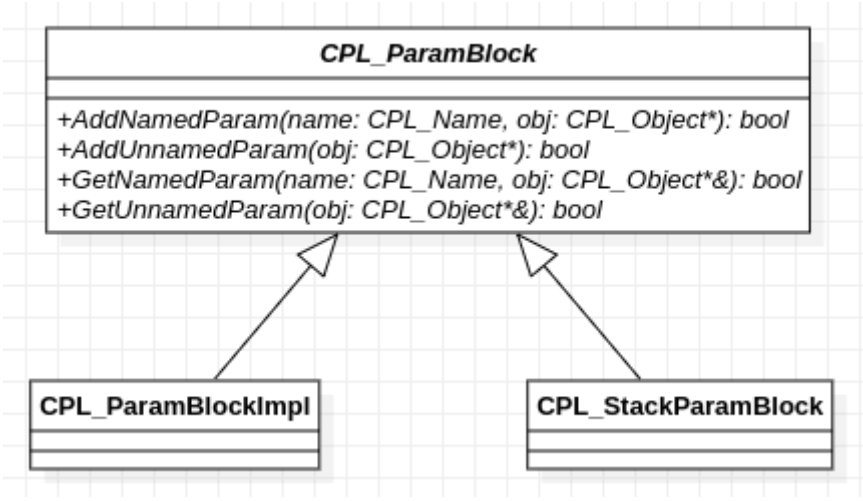
- Implementare una nuova istruzione per l'interprete per segnalare una chiamata a funzione ordinata dal typechecker.
- Creare un programma in grado di disassemblare e riassemblare il bytecode per inserire all'interno di programmi la nuova istruzione (così da non dover implementare la feature all'interno del typechecker statico solo per effettuare un test).

4.2.2.1 Assembler e disassembler

Ho deciso di sviluppare il programma in `C++` così da poter riutilizzare codice dell'interprete e del compilatore.

4.2.2.2 Cambiamenti all'interno dell'interprete

Data la complessità dell'interprete ho deciso di sfruttare il più possibile il codice già presente estraendo l'interfaccia di `CPL_ParamBlockImpl` in una classe astratta e creando una nuova sottoclasse.



Come spiegato in 4.2.1.2 per rendere la chiamata di funzione più leggera manteniamo solo un riferimento allo stack.

L'utilizzo di una classe astratta invece permette di non modificare il codice relativo alla chiamata di funzione cioè come esso aggiunga all'interno dell'ambiente locale i parametri.

```
1  // directcall.h
2  class CPL_StackParamBlock : public CPL_ParamBlock {
3  public:
4    using StackRef = CPL_Stack<CPL_Object *>&;
5  ...
6  private:
7    StackRef stack;
8    int given = 0;
9    int param_count = 0;
10 };
11
12 // directcall.cpp
13 bool CPL_StackParamBlock::AddUnnamedParam(CPL_Object *) {
14   param_count++;
15   return true;
16 }
17
18 bool CPL_StackParamBlock::NextParam(CPL_Name, CPL_Object *&obj) {
19   if (given > param_count) {
20     return false;
21   }
22
23   obj = stack.Top(param_count - given);
24   ++given;
25   return true;
26 }
```

Listing 4.3: Implementazione di CPL_StackParamBlock

4.2.3 Risultati

Eseguendo di nuovo una profilazione con `gperf` effettivamente di `CPL_StackParamBlock` non è presente nessuna traccia.

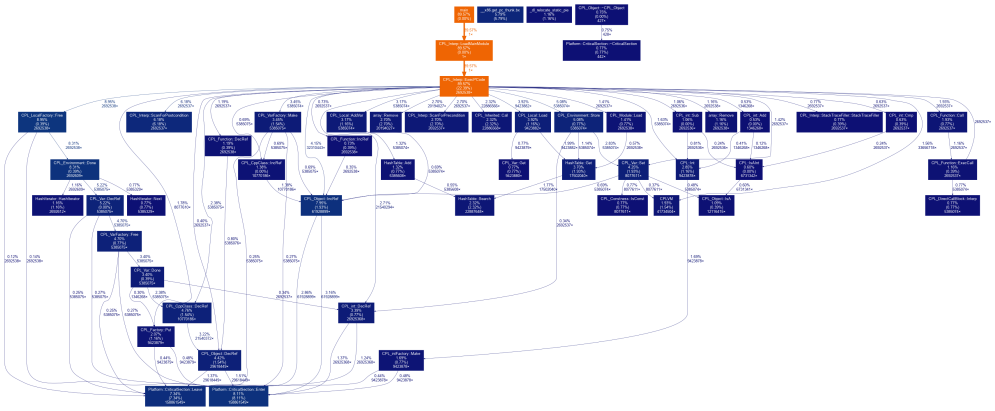


Figura 4.2: Grafo delle chiamate dell'interprete dopo l'ottimizzazione

In particolare è possibile notare che la percentuale mancante è stata distribuita tra le funzioni che si occupano di operare sul reference counting dell'ambiente, proprio a sottolineare come sia effettivamente quello il collo di bottiglia.

Successivamente utilizzando il software `hyperfine` si è fatto un benchmark tra l'implementazione già presente (`Benchmark 1`) e quella nuova (`Benchmark 2`).

```
Benchmark 1: ./cploldcall test/fib.cpl
  Time (mean ± σ):     12.968 s ±  1.008 s    [User: 12.845 s, System: 0.002 s]
  Range (min … max):   11.738 s … 14.846 s    10 runs

Benchmark 2: ./cpl test/fib.cpl
  Time (mean ± σ):     11.383 s ±  0.614 s    [User: 11.339 s, System: 0.002 s]
  Range (min … max):   10.566 s … 12.835 s    10 runs
```

Figura 4.3: Benchmark ottimizzazione chiamate a funzione

Effettivamente ottenendo circa il 2% di tempo di esecuzione in meno come prevedibile dalla profilazione tramite `gperf` nel paragrafo 4.2.1.2.

Data la complessità aggiunta all'interprete e il lavoro da spendere sul typechecker statico il tutor interno ha convenuto sul non proseguire con il completamente della funzionalità.

Capitolo 5

Nuove funzionalità

La combinazione di mancanza di controlli statici e type system molto primitivo ha portato gli sviluppatori che utilizzano CPL a non seguire sempre le bune norme di programmazione, in particolare, ad abusare del tpo `any` nelle dichiarazioni di variabili.

Questo è stato notato durante l'esecuzione del typechecker statico su vari progetti scritti in CPL dall'azienda.

L'utilizzo del tipo `any` vanifica in molte situazioni l'utilizzo del typechecker statico infatti non possiamo sapere:

- Quali metodi su di esso possono essere chiamati, in particolare non possiamo nemmeno sapere che tipo ritorna questa funzione.
- Se è assegnabile a una determinata variabile o meno.

Creando sezioni di programma in cui nonostante il controllo statico possono comunque verificarsi un errore di tipo a runtime.

Inoltre un'altra lamentela delle persone che utilizzano CPL giornalmente riguarda le regole di scoping del linguaggio, esplicitate nel paragrafo 2.1.5, che oltre a rendere il linguaggio più scomodo da usare, vedasi appunto la dichiarazione di variabili all'interno di un ciclo, possono nascondere anche ulteriori errori runtime che con opportune modifiche al compilatore e al typechecker sarebbero identificabili staticamente.

Di conseguenza si è deciso di operare sul compilatore e sull'interprete per aggiungere alcune nuove funzionalità, in grado rendere il linguaggio più semplice da usare, sempre mantenendo la retro compatibilità.

5.1 Scoping

Come descritto nel paragrafo 2.1.5, lo scoping esiste solo a livello di: modulo, classe e funzione, questo implica che il seguente programma CPL:

```
1 if cond
2   var int i := 11
3 end
4
```

```
5 ? i
```

Listing 5.1: Dichiarazione di variabile condizionale

Risulterà nella stampa di "11" nel caso il valore di `cond` sia `true` mentre nel caso sia `false` risulterà in un errore runtime in quanto la variabile non risulterà dichiarata.

Questo tipo di errore può avvenire ad esempio anche in `Python` allo stesso modo, mentre in linguaggi in cui è presente un controllo statico, come ad esempio `C++`, il programma verrebbe rifiutato durante la fase di compilazione.

Allo stato attuale il typechecker statico non può effettuare questo tipo di controllo, in quanto il `p-code` essendo una serie di istruzioni per una stack machine, non ha informazioni sulle strutture condizionali presenti all'interno del programma, esse sono compilate in una serie di salti condizionali.

Di conseguenza è risultato necessario introdurre una nuova istruzione all'interno del `p-code`, per indicare quando una variabile esce dallo scope.

5.1.1 Modifiche al compilatore

Il compilatore deve mantenere informazioni sulle variabili dichiarate all'interno dello scope corrente. Ci si può trovare all'interno di un numero arbitrario di scope. Dagli scope si esce ed entra con una logica Last In, First Out (LIFO).

Di conseguenza risulta logico modellare il sistema di scoping come una serie di ambienti contenuti all'interno di uno `Stack`:

```
1  // scoping.h
2  class Scope {
3  public:
4    Scope()
5
6    void declare(std::string name);
7
8    void enter_scope();
9    void exit_scope(Tokens);
10
11 private:
12   std::stack<Frame> frames;
13 };
14
15 // scoping.cpp
16 void Scope::declare(std::string name) { frames.top().declare(name); }
17
18 void Scope::enter_scope() { frames.push(Frame()); }
19
20 void Scope::exit_scope(Tokens tokens) {
21   auto frame = frames.top();
22   for (const auto& variable_name : frame) {
23     tokens.pcode_s(DELVAR, variable_name.c_str());
24   }
25   frames.pop();
26 }
```

Listing 5.2: Definizione dello scope all'interno del compilatore

E un frame non è altro che una collezione di variabili:

```
// scoping.h
class Frame {
private:
    std::unordered_set<std::string> declared_vars;

public:
  using iterator = decltype(declared_vars)::iterator;
  Frame();

  iterator begin();
  iterator end();

  void declare(std::string name);
};
```

Listing 5.3: Definizione di un Frame all'interno del compilatore

5.1.2 Modifiche nel typechecker

Per quando riguarda il typechecker invece non deve altro che eliminare dall'ambiente la variabile nominata dall'istruzione `DELVAR`.

5.1.3 Modifiche all'interno dell'interprete

Invece all'interno dell'interprete non è possibile eliminare la variabile all'interno dell'ambiente corrente ogni volta che si incontra l'istruzione `DELVAR`, questo romperebbe la retro compatibilità con programmi scritti in precedenza.

Per questo si è deciso invece di mantenere all'interno dell'ambiente una lista di nomi eliminati, così da poter mostrare un messaggio di warning quando una variabile uscita dallo scope viene nominata.

Inoltre, questo ci da la possibilità di permettere la dichiarazione di variabili all'interno dei cicli, in quanto, ogni volta che termina un'interazione la variabile verrà segnata come eliminata dallo scope per poi essere dichiarata nuovamente all'iterazione successiva.

```
// cplenv.h
class CPL_Local: public CPL_Environment {
...
private:
  std::unordered_set<CPL_Name> deleted = {};
};

// cplenv.cpp
CPL_Object* CPL_Local::Load(CPL_Name name) {
  if (deleted.find(name) != deleted.end()) {
    CPL_Warn("variable " << CPL::Get(name) << " was used when out of scope");
  }
  ...
}

bool CPL_Local::AddVar(CPL_Name name, CPL_Var* var, bool raise_error) {
  if(deleted.erase(name)) {
```

```
18      RemoveItem(name);
19    }
20    ...
21  }
```

Listing 5.4: Introduzione dei nomi eliminati all'interno dell'ambiente

5.2 Generics

Come spiegato all'inizio del capitolo 5 la mancanza di polimorfismo parametrico, cioè la possibilità di implementare classi e funzioni in grado di operare su valori senza dipendere dal loro tipo (Parametric polymorphism, [...], allows a single piece of code to be typed "generically," using variables in place of actual types, and then instantiated with particular types as needed. Parametric definitions are uniform: all of their instances behave the same. [5]) ha portato all'abuso del tipo `any`, in particolare, nella definizione di contenitori generici.

Sono molteplici anche all'interno della stessa libreria standard del linguaggio le classi e funzioni che utilizzano `any`, quando con la presenza di un meccanismo di permetterebbe di avere codice in grado di essere verificato staticamente, un esempio di queste e la classe builtin `array`:

```
class array(Structure)
    var int Len
    array[any] operator *(any)
    array[any] operator *(array[any])
    array[any] operator +(any)
    array[any] operator +(array[any])
    array[any] operator -(any)
    array[any] operator -(array[any])
    func array[any] Append(any item)
    func int IndexOf(any item, bool exact)
    func array[any] Insert(int index, any item)
    func array[any] Remove(int index)
    func array[any] RemoveItems()
    func array[any] Revert()
end
```

Figura 5.1: Documentazione riguardante la classe array

Esistono molteplici modi per implementare questa funzionalità all'interno di un linguaggio di programmazione, nel caso di quella basata sul concetto di oggetto spiccano: l'implementazione tramite type erasure come ad esempio fa il linguaggio `Java` oppure tramite template come fa il linguaggio `C++`.

5.2.1 Generics (Java)

L'implementazione di generics alla `Java`, cioè tramite type erasure genera una istanza singola istanza della classe o della funzione generica a prescindere dal numero di suoi

utilizzi.

Prendiamo ad esempio la seguente definizione `Java`:

```
public class Box<T> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Listing 5.5: Classe generica in Java

Durante l'esecuzione del programma contenente non esisterà nessuna classe `Box<Integer>` o `Box<Float>` ma esisterà un'unica definizione simile a questa:

```
public class Box {
    private Object;

    public void set(Object) { this.t = t; }
    public Object get() { return t; }
}
```

Listing 5.6: Classe generica durante il runtime in Java

Questo significa che l'esistenza della classe generica è qualcosa che esiste solamente durante il typechecking del programma [6].

Questo tipo di approccio però non è applicabile a CPL, in quanto, dobbiamo tenere a mente che il typechecker statico è uno strumento esterno al linguaggio di programmazione. Sostituire tutti le istanze di un tipo generico con `any` ci farebbe il controllo runtime di tipi che avviene ogni volta che avviene un assegnazione ad una variabile come è stato descritto nel paragrafo 2.1.4.

5.2.2 Template (C++)

L'approccio di `C++` invece è totalmente opposto, prendiamo la seguente definizione di classe templetizzata in `C++`:

```
template <typename T>
class Box {
private:
    T t;

public:
    void set(T t) { this.t = t; }
    T get() { return t; }
};
```

Listing 5.7: Classe generica in C++

Durante la compilazione del programma vengono generate classi specifiche per ogni istanza del template, se abbiamo riferimenti alle classi `Box<int>` e `Box<float>` allora verranno generate definizioni similari alle seguenti [7]:

```
class BoxInt {
private:
    int t;

public:
    void set(int t) { this.t = t; }
    int get() { return t; }
};

class BoxFloat {
private:
    float t;

public:
    void set(float t) { this.t = t; }
    float get() { return t; }
};
```

Listing 5.8: Istanze del template in C++

Questo si adatta di più al nostro caso, infatti, generare definizioni di classi e di metodi non parametrici sostituendo gli argomenti dei template con i tipi effettivi ci permette di mantenere i controlli che l'approccio tramite generics alla `Java` non ci permetteva di avere.

5.2.3 Implementazione

Come appunto descritto in 5.2.2 si è deciso di procedere con un approccio ti tipo generativo, per l'implementazione sono necessari quindi i seguenti passaggi:

- Introdurre una nuova istruzione per stabilire l'introduzione di una nuova variabile di tipo all'interno dell'ambiente.
- Stabilire la sintassi e modificare il parser di conseguenza
- Catturare la definizione della classe come se fosse una qualsiasi classe normale.
- Quando la classe generica viene saturata sostituire le variabili di tipo.

5.2.3.1 Sintassi

Per mantenere la grammatica del linguaggio sempre DCFG e coerente con i costrutti sintattici già presenti, come ad esempio array e i dizionari, si è deciso di introdurre e le variabili di tipo tra parentesi quadre, prima della definizione e di istanziare le classi generiche nello stesso modo in cui vengono istanziati dizionari e array.

```
class [T, K] Pair
  var T fst
  var K snd

  proc Init(T f, K s)
    fst := f
    snd := s
  end

  func T ProjectFst()
    result := fst
```

```
12    end
13
14    func K ProjectSnd()
15      result := snd
16    end
17  end
18
19  var Pair[int, str] p := Pair[int, str](1, "one")
20  ? p.ProjectFst() -- 1
21  ? p.ProjectSnd() -- one
```

Listing 5.9: Classi generiche in CPL.

5.2.3.2 Compilazione

Quando viene incontrata la definizione della lista delle variabili di tipo tra parentesi quadre il compilatore emette una serie di istruzioni `GENERIC` accompagnate dal nome della variabile di tipo, esse verranno aggiunte ai tipi nell'ambiente durante tutta la definizione della classe o della funzione.

E solamente al momento dell'istanziazione della classe generica viene effettivamente effettuata la sostituzione.

La sostituzione ovviamente non può essere diretta, in quanto, bisogna verificare di non sostituire variabili (di tipo) che sono state introdotte date ulteriori definizioni generiche interne alla classe con lo stesso nome di quella che stiamo sostituendo, infatti, si deve mantenere igiene [8] all'interno della sostituzione.

```
1  CPL_Object *replace(CPL_Object *ty, CPL_GenericType::GenericConstraints
       with, bool isInTypeDef = false) {
2    if (dynamic_cast<CPL_TypeParameter *>(ty)) {
3      auto dyn_param = static_cast<CPL_TypeParameter *>(ty);
4      auto pos = std::find_if(with.begin(), with.end(), [dyn_param](auto p) {
5        return p.first == dyn_param->Name();
6      });
7      if (pos != with.end()) {
8        if (pos->second != CPL::nil && !isInTypeDef) { // If we aren't inside
             a type we should increment the reference counter of the object
9          pos->second->IncRef();
10       }
11       return pos->second;
12     } else {
13       return ty;
14   }
15     }
16
17   if (dynamic_cast<CPL_GenericType *>(ty)) {
18     auto generic_ty = static_cast<CPL_GenericType *>(ty);
19     auto new_type_def = std::set_difference(with, generic_ty->boundNames(),
         NAME_COMPARATOR);
20     return replaceGeneric(generic_ty, new_type_def, true);
21   }
22
23   return ty;
```

```
24 }
```

Listing 5.10: Funzione di sostituzione

Appunto a riga 19 eliminiamo i nomi catturati dalla classe generica interna così da evitare la cattura di nomi definiti da essa.

5.2.3.3 Controllo statico dei tipi

Data la costruzione di classi specifiche per ogni istanza della classe generica, il processo di verifica statica dei tipi, risulta del tutto identico a quello che avviene su una classe non generica, infatti, risulterà sufficiente verificare la correttezza dei tipi all'interno delle varie classi generate.

5.3 Tipi somma

Un altro contesto in cui l'utilizzo del tipo `any` è molto diffuso è quando si definire una funzione in grado di operare su più tipi, senza che essi abbiano tra loro una relazione di subtyping [9], anche in questo caso sono molteplici i casi in cui la libreria standard fa uso in modo informale di questo tipo di costrutto.

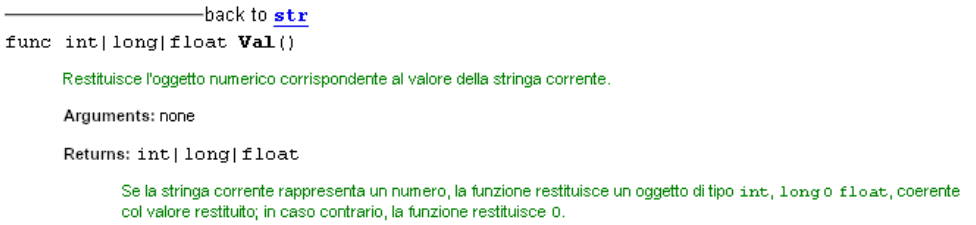


Figura 5.2: Documentazione riguardante la funzione Val

5.3.1 Typing

Definiamo le relazioni di subtyping per tipo per la somma:

$$A <: A + B$$

$$B <: A + B$$

Ovviamente questa mantiene la regola si transitività descritta in 2.1.2.

Questo, però, rende necessaria l'introduzione di un nuovo costrutto per fare introspezione sulla somma per verificare effettivamente il tipo dell'oggetto durante l'esecuzione del programma, così da poter operare in modo sicuro su di essa.

L'introduzione di un costrutto in grado di operare in modo differente in base al tipo, introduce il concetto di polimorfismo ad-hoc [10] all'interno del linguaggio, in particolare, si è deciso di prendere ispirazione da il costrutto `_Generic` introdotto in `C11` [11].

5.3.2 Implementazione

Dal punto di vista implementativo si è deciso di procedere con una unione priva di tag, così da permettere l'utilizzo delle regole di subtyping tipiche della programmazione orientata agli oggetti.

Per quanto riguarda la possibilità di fare introspezione sulle variabili si è deciso di riutilizzare l'istruzione di switch case per fare effettuare controlli statici sul tipo.

5.3.2.1 Sintassi

Un tipo somma si dichiara come una serie di tipi separati da `||` che si legge come `or`.

```
var int||str||long x := 42
? x -- 42
x := "Hello, World!"
? x -- Hello, World!
```

Listing 5.11: Dichiarazione di una variabile di tipo somma

L'istruzione switch type al suo interno dichiara una lista di casi accompagnati dal tipo, all'interno del blocco case dal punto di vista del typechecking statico la variabile nominata dallo `switchtype` cambia il suo tipo in quello nominato da il `case`.

```
class A
  proc a()
    ? "a"
  end
end

class B
end

var A||B x := A()

switchtype x
case A
  x.a() -- a
case any
  ? "B or nil"
end
```

Listing 5.12: Istruzione di switch type

È possibile indicare un `case` con il tipo `any` nel quale finiranno tutti i tipi non nominati in precedenza e il valore `nil`.

Inoltre, il typechecker statico controlla che i casi all'interno `switchcase` siano totali, cioè che sia presente un caso per ogni tipo nella somma oppure sia presente un caso `any`.

5.3.2.2 Esecuzione runtime

Durante l'implementazione dobbiamo però stare attenti a una serie di possibili errori:

- `SWITCHTYPE` annidati.

- Rispettare la transitività.

```
...
case SWITCHTYPE: {
  auto name = pcode.addr;
  auto variable = look_up_variable(exec_local, exec_inst, exec_mod, name);
  auto value = variable->Get();

  auto found_case = false;
  auto count = 0;
  while (!found_case && !(count == 0 && pcode.idx == ENDSWITCHTYPE)) {
    pcode = exec_mod->PCode(++rPC);

    if (pcode.idx == SWITCHTYPE) {
      ++count;
    } else if (pcode.idx == ENDSWITCHTYPE) {
      --count;
    } else if (count == 0 && pcode.idx == STARTTYPECASE) {
      auto type = look_up_type(
        exec_local,
        exec_inst,
        exec_mod,
        generic_variables,
        cl,
        rPC
      );

      if (value != CPL::nil && value->IsA(type) || type == CPL::nil) {
        found_case = true;
      }

      if (type) {
        type->DecRef();
      }
    }
  }

  if (!found_case) {
    CPL_ThrowTypeException(format(
      _T("Unsatisfied switchtype on variable %s"),
      CPL::Get(name)
    ));
  }
}
...
case ENDTYPECASE: {
  while (pcode.idx != ENDSWITCHTYPE) {
    pcode = exec_mod->PCode(++rPC);
  }
  break;
}
...
```

Listing 5.13: Implementazione switchtype all'interno dell'interprete

Alle righe 12-16 viene effettivamente controllato che non ci si trovi all'interno di uno switchtypeannidato tenendo traccia del numero di SWITCHTYPE incontrati.

Invece a riga 26 se il tipo rispetta la relazione IsA, e in quel caso si procede a interpretare il p-code normalmente, finchè non si raggiunge la fine del blocco.

5.3.2.3 Verifica statica

Come descritto in 5.3.2.1 deve:

- All'interno dei blocchi cambiare il tipo della variabile nominata dallo switchtype con il tipo nominato da il case.
- Verificare la totalità dei casi dello switchtype.

```
1  proc SwitchType()
2  pre Current().Name = "SWITCHTYPE"
3      var Variable subject := LookUpVariable(Current().Argument)
4      var Clazz real_ty := subject.Type
5      var array[Clazz] bounds := real_ty.SubTypes().Clone()
6      Advance()
7
8      while Current().Name <> "ENDSWITCHTYPE"
9          assert Current().Name = "STARTTYPECASE"
10         TypeLiteral()
11         var Clazz ty := stack.Pop()
12         if not ty.IsAny()
13             subject.Type := ty
14             var Clazz it
15             for each it in bounds; if it.Name <> ty.Name
16                 bounds := bounds - [it]
17             end; end
18         else
19             bounds := []
20             subject.Type := real_ty
21         end
22         Expression("ENDTYPECASE")
23     end
24
25     if bounds.Len > 0
26         Error("Unsaturated switch type on variable " + subject.Name, true)
27     else
28         subject.Type := real_ty
29         Advance()
30     end
31 end
```

Listing 5.14: Verifica della totalità

A riga 5 vengono salvati i tipi possibili che la variabile può assumere e alle righe 17 e 20 vengono rimossi tutti i tipi che sono stati incontrati, così da poter verifica se lo switchtype è totale a riga 26.

A riga 13 invece viene cambiato il tipo della variabile per poi essere ripristinato a riga 29 appenda fuori dallo switchtype.

5.4 Tipi non nullabili

Il problema dei riferimenti a oggetti nulli è un problema che affligge il mondo dei linguaggi di programmazione orientati agli oggetti, talmente tanto che lo stesso creatore del concetto, Tony Hoare, lo considera il "Billion-dollar mistake".

5.4.1 Tipi non nullabili in altri linguaggi

La maggior parte dei linguaggi nati di recente ha risolto alla radice il problema eliminando direttamente il concetto di riferimento nullo.

Invece per i linguaggi che purtroppo hanno già introdotto il concetto di riferimento nullo. sbarazzarsene risulta risulta tutt'altro che semplice.

Due grandi esempi importanti sono gli approcci presi dai linguaggi di programmazione `Java` e `C#`.

5.4.1.1 Tipi non nullabili in Java

Il linguaggio di programmazione `Java` ha cercato di arginare il problema introducendo all'interno della sua libreria standard la classe `Optional<T>` per contenere i riferimenti che possono essere nulli. Questo, però, introduce una serie di problematiche:

- Risulta essere qualcosa di prettamente documentale in quanto una variabile di tipo `Optional` può comunque essere nulla.
- Non interagisce bene con la type erasure, infatti come spiegato in 5.2.1 `Optional<Integer>` e `Optional<Double>` sono di fatto lo stesso tipo, questo porta a perdere la possibilità di eseguire un overload di metodo nel caso l'unico parametro a differenziarli sia il tipo all'interno di `Optional`.

5.4.1.2 Tipi non nullabili in C#

Il linguaggio `C#` invece da preso una strada diversa, da `C# 8.0`, previa attivazione di un flag all'interno della configurazione del progetto, ha reso tutti i tipi di default nullabili.

Questo nonostante risolva i problemi dell'implementazione di `Java` ne introduce altri:

- Rende incompatibile il codice scritto con i tipi non nullabili disattivati e richiede una quantità di lavoro non banale aggiornare codice scritto senza questa funzionalià attivata.
- Rende particolarmente scomodo interagire con librerie scritte senza di essi.

5.4.2 Soluzione scelta in CPL

Introdurre una classe che provi a emulare il comportamento della monade `Maybe` come viene fatto in `Java` e in `C#` non sarebbe utilizzabile in quanto manca il supporto alle Higher-order function (HOF).

Di conseguenza, si è deciso d'intraprendere l'approccio opposto cioè quello di lasciare i tipi nullabili e introdurre il tipo non nullabile come concetto extra.

5.4.3 Typing

Dato il tipo T definiamo come $T!$ la sua versione non nullabile.

Allora abbiamo la regola di typing:

$$!T <: T$$

Dobbiamo anche mantenere la proprietà di subtyping:

$$\frac{D <: E}{!D <: !E}$$

Inoltre, vale sempre la regola di transitività definita in 2.1.2.

Anche in questo caso come in 5.3.1 ci servirà un costrutto per convertire in modo sicuro T in $T!$.

Dato che il costrutto `switchtype` introdotto in 5.3.2 effettua il match solo quando il valore all'interno della variabile è diverso da `nil` possiamo riutilizzare la stessa struttura.

5.4.4 Implementazione

Dal punto di vista implementativo il costrutto di tipo non nullabile non aggiunge nessun comportamento runtime, però è necessario comunque aggiungere un una nuova istruzione all'interno del formato `pcode`, per indicare la presenza di `!` in una dichiarazione di tipo.

5.4.4.1 Sintassi

Un tipo non nullabile si dichiara come un tipo normale ma accompagnato dal simbolo `!` alla fine del tipo.

```
1 var int! x := 11
2 -- x := nil       -> Errore!
```

Listing 5.15: Dichiarazione di variabile non nullabile

E come spiegato in 5.4.3 all'interno dello scope un `case` di uno `switchtype` è considerato come non nullabile.

```
1 var int x := 11
2 switchtype x
3 case int
4   var int! y := x
5 case ?
6   ? "x era nullo"
7 end
8
9 -- var int! y := x      -> Errore! x potrebbe essere nullo
```

Listing 5.16: Dichiarazione di variabile non nullabile

5.4.4.2 Verifica statica

Dal punto di verifica statica ci basterà aggiungere alla definizione dello `switchtype` data in 5.3.2.3 l'informazione che nel contesto il tipo è non nullabile.

Capitolo 6

Valutazione retrospettiva

6.1 Prodotti sviluppati

Il typechecker statico può essere considerato pronto all'uso in produzione. È riuscito a identificare errori sia all'interno dell'implementazione stessa, che in vari software scritti dall'azienda.

Inoltre, per esso è disponibile un'ampia suite di test in grado di verificare in modo trasversale le varie funzionalità implementate e le varie interazioni che si sviluppano tra di esse.

Le funzionalità introdotte all'interno del linguaggio sono state discusse sia con il designer originale del linguaggio che con le persone che lo utilizzano ogni giorno.

Nonostante le nuove funzionalità introdotte, per questioni di tempo, non siano state provate nel contesto di produzione da parte di sviluppatori estranei al progetto, durante il refactor del typechecker statico ho potuto verificarne l'utilità, infatti, esse mi hanno aiutato a ridurre sensibilmente la quantità di codice scritta.

6.2 Preparazione universitaria

Nonostante la laurea triennale in informatica non fornisca corsi specifici sulla teoria dei linguaggi di programmazione, sicuramente fornisce basi solide per esplorare l'argomento in autonomia.

Inoltre, oltre alla preparazione teorica, fornisce anche gli strumenti per poter affrontare un progetto software di questo calibro, all'interno di un contesto aziendale, come ad esempio effettuare in mood corretto analisi dei requisiti e costruire una buona architettura software in grado di soddisfare i requisiti qualitativi richiesti dal tutor esterno.

6.3 Problematiche tecniche

Il fatto che il linguaggio sia nato nel `1998` ha portato una serie di problematiche come il dover porre particolare attenzione alla retro compatibilità, inoltre, interagire con un

progetto software di quell'età è risultato particolarmente ostico, in quanto, molte delle buone prassi che si danno per scontate nei progetti moderni non erano state seguite.

6.4 Considerazioni personali

Non accade tutti i giorni, in particolare in Italia, di avere la possibilità di lavorare a un linguaggio di programmazione al di fuori dell'ambito strettamente accademico, questa è stata una opportunità preziosa, mi ha dato la possibilità di approfondire gli argomenti che più mi interessano dell'informatica non solamente in un contesto teorico ma applicandolo a una realtà aziendale.

Appendice A

Sintassi del linguaggio EBNF

⟨*program*⟩ ::= { ⟨*statement*⟩ };

⟨*statement*⟩ ::= 'class' ⟨*classdef*⟩
  | 'func' ⟨*funcdef*⟩ ⟨*funcbody*⟩
  | 'proc' ⟨*procdef*⟩ ⟨*funcbody*⟩
  | 'import' ⟨*id*⟩ { '.' ⟨*id*⟩ }
  | 'try' ⟨*stm_try*⟩
  | ⟨*body*⟩;

⟨*type*⟩ := 'any'
  | '?'
  | 'int'
  | 'long'
  | 'str'
  | 'bool'
  | 'float'
  | 'array' '[' ⟨*type*⟩ ']'
  | 'dict' '[' ⟨*type*⟩ ']';

⟨*vardecl*⟩ := ⟨*type*⟩ ⟨*id*⟩ [':=' ⟨*expr*⟩];

⟨*funcdef*⟩ := ⟨*type*⟩ ⟨*id*⟩ ⟨*parlist*⟩;

⟨*procdef*⟩ := ⟨*id*⟩ ⟨*parlist*⟩;

⟨*funcbody*⟩ := { ⟨*body*⟩ } end;

⟨*parlist*⟩ := '(' [⟨*param*⟩ { ',' ⟨*param*⟩ }] ')';

⟨*param*⟩ := ⟨*type*⟩ ['@'] ⟨*id*⟩ [:= ⟨*expr*⟩];

⟨*classdef*⟩ := ⟨*id*⟩ ['(' ⟨*id*⟩ { ',' ⟨*id*⟩ } ')']
    { 'var' ⟨*vardecl*⟩
  | 'func' ⟨*funcdef*⟩ (⟨*funcbody*⟩ | 'abstract')
  | 'proc' ⟨*procdef*⟩ (⟨*funcbody*⟩ | 'abstract') }
    'end';

⟨*body*⟩ := 'var' ⟨*vardecl*⟩
| 'if' ⟨*stm_if*⟩
| 'switch' ⟨*stm_case*⟩
| 'while' ⟨*stm_while*⟩
| 'repeat' ⟨*stm_repeat*⟩
| 'for' ⟨*stm_for*⟩
| 'try' ⟨*stm_try*⟩
| 'raise' '(' ⟨*expr*⟩ ')'
| 'pre' ⟨*expr*⟩
| 'post' ⟨*expr*⟩
| 'asset' ⟨*expr*⟩
| '?' ⟨*expr*⟩
| ⟨*expr*⟩;

⟨*stm_if*⟩ ::= ⟨*expr*⟩ { ⟨*body*⟩ }
{ 'elseif' ⟨*expr*⟩ { ⟨*body*⟩ } } ['else' { ⟨*body*⟩ }]
'end';

⟨*stm_case*⟩ ::= ⟨*expr*⟩
{ 'case' ⟨*expr*⟩ ⟨*body*⟩ } ['else' { ⟨*body*⟩ }]
'end';

⟨*stm_while*⟩ ::= ⟨*expr*⟩ { ⟨*body*⟩ } 'end';

⟨*stm_repeat*⟩ ::= { ⟨*body*⟩ } 'until' ⟨*expr*⟩;

⟨*stm_for*⟩ ::= 'each' ⟨*stm_each*⟩
| ⟨*stm_for_l*⟩

⟨*stm_each*⟩ ::= ⟨*id*⟩ 'in' ⟨*expr*⟩ { ⟨*body*⟩ } 'end';

⟨*stm_for_l*⟩ ::= id ':=' ⟨*expr*⟩ ('to' | 'downto') ⟨*expr*⟩
{ ⟨*body*⟩ } 'end';

⟨*stm_try*⟩ ::= { ⟨*body*⟩ }
{ 'catch' '(' ⟨*type*⟩ ⟨*id*⟩ ')' { ⟨*body*⟩ } } ['finally' { ⟨*body*⟩ }]
'end';

⟨*expr*⟩ ::= ⟨*exp1*⟩ { 'or' ⟨*exp1*⟩ };

⟨*exp1*⟩ ::= ⟨*rexp*⟩ { 'and' ⟨*rexp*⟩ };

⟨*rexp*⟩ ::= ⟨*aexpr*⟩ { ('⟨' / '<=' / '=' / '⟩=' | '>' | '⟨⟩') ⟨*aexpr*⟩ };

⟨*aexpr*⟩ ::= ('-' ⟨*fact*⟩ | '+' ⟨*fact*⟩ | ⟨*fact*⟩) { ('+' | '-') ⟨*fact*⟩ };

⟨*fact*⟩ ::= ⟨*term*⟩ { ('*' | '/' | '%' | 'div') ⟨*term*⟩ };

⟨*term*⟩ ::= ⟨*literal*⟩
| ⟨*fullaccess*⟩ [':=' ⟨*expr*⟩];

⟨*fullaccess*⟩ := ⟨*fullaccess*⟩ '.' ⟨*id*⟩
| ⟨*fullaccess*⟩ '[' ⟨*expr*⟩ { ',' ⟨*expr*⟩ } ']'
| ⟨*fullaccess*⟩ '(' ⟨*expr*⟩ { ',' ⟨*expr*⟩ } ')'
| '@' ⟨*fullaccess*⟩;

Appendice B

Istruzioni pcode disponibili

Opcode	byte
STARTVAR	0
THISOBJ	1
TYPE	2
ENDVAR	3
STOREX	4
INTCONST	5
STORE	6
OK	7
ADD	8
PRINT	9
LOAD	10
STRCONST	11
SUB	12
MUL	13
DIV	14
MOD	15
AND	16
OR	17
NOT	18
POP	19
LOADX	20
DECLCLASS	21
ENDCLASS	22
SLICE	23

Opcode	byte
CALL	24
BOOLCONST	25
SUPER	26
EQL	27
NEQ	28
GRT	29
LST	30
GTE	31
LTE	32
JUMP	33
JUMPZ	34
LINE	35
LABEL	36
UMINUS	37
SOURCE	38
JUMPOR	39
JUMPAND	40
STARTFUNC	41
ENDFUNC	42
STARTPROC	43
IMPORT	44
STARTPMETH	45
STARTFMETH	46
VAR	47
PROP	48
LOCALVAR	49
PARAMETERS	50
PARAM	51
ENDPARAM	52
STOREPARAM	53
NIL	54
COMPILERERROR	55
POPPARAM	56
STARTDEFAULTPARAM	57
ENDDEFAULTPARAM	58
THISMODULE	59
BEGINTYPE	60

Opcode	byte
ENDTYPE	61
ARRAY	62
STORESLICE	63
PARAMARRAY	64
DICT	65
PARAMDICT	66
TRY	67
CATCH	68
FINALLY	69
RAISE	70
RAISEAGAIN	71
ENDTRY	72
CATCHALL	73
CATCHVAR	74
RAISEFINALLY	75
ITERATOR	76
ITERATORNEXT	77
SWITCH	78
CASEINT	79
CASESTR	80
CASENEXT	81
CASEDEFAULT	82
ITERATORKEY	83
BEGINASSERT	84
ENDASSERT	85
BEGINPRE	86
ENDPRE	87
BEGINPOST	88
ENDPOST	89
BEGINOLD	90
ENDOLD	91
PUREVIRTUAL	92
STARTEVENT	93
STARTEMETH	94
SENDER	95
LOADREF	96
PARAMREF	97

Opcode	byte
IMPORTFROM	98
INHERITED	99
INVERSEITERATOR	100
FLOATCONST	101
MULTISLICE	102
STOREMULTISLICE	103
DATECONST	104
DATETIMECONST	105
CONSTOBJ	106
CLOCKTIMECONST	107
LOADXREF	108
PROMPT	109
DEBUGMODE	110
LONGCONST	111
BREAKPOINT	122
LOADBUILTINDIRECT	123
STOREINSTANCEDIRECT	124
LOADINSTANCEDIRECT	125
STOREMODULEDIRECT	126
LOADMODULEDIRECT	127

Appendice C

Istruzioni pcode aggiunte

Opcode	byte
GENERIC	112
DELVAR	113
TYPESUM	114
NONNULL	115
SWITCHTYPE	116
STARTTYPECASE	117
ENDTYPECASE	118
ENDSWITCHTYPE	119

Riferimenti

[1] Facebook Inc., *Flow*. Disponibile su: https://flow.org/

[2] Vimeo Inc., *psalm*. Disponibile su: https://psalm.dev/

[3] B. C. Pierce, in *Types and Programming Languages*, The MIT Press, 2002, pag. 251.

[4] B. Meyer, «Static typing and other mysteries of life», vol. 94, 1995.

[5] B. C. Pierce, in *Types and Programming Languages*, The MIT Press, 2002, pag. 340.

[6] J. Gosling, B. Joy, G. L. Steele, G. Bracha, e A. Buckley, *The Java® language specification*, Java SE 8 edition. Upper Saddle River, NJ: Addison-Wesley, 2014.

[7] ISO, *ISO/IEC 14882:2017: Programming languages - C++*. pub-iso, 2017.

[8] S. Krishnamurthi, in *Programming Languages: Application and Interpretation*, Self-published, 2017, pagg. 98–99.

[9] B. C. Pierce, in *Types and Programming Languages*, The MIT Press, 2002, pagg. 132–136.

[10] W. P. Cardelli Luca, «On understanding types, data abstraction, and polymorphism», *ACM Computing Surveys*, vol. 17, n. 4, pagg. 471–523, 1985, doi: 10.1145/6041.6042.

[11] ISO, *ISO/IEC 9899:201x: Programming languages - C*. pub-iso, 2011.

Acronimi

AOT ahead-of-time. 23

CAT Changing Availability or Type. 18

CPL Code Painter Language. i, ii, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19, 20, 24, 25, 29, 33

DCFG Deterministic Context-free Grammar. 4, 10, 34

EBNF Extended Backus–Naur Form. 10

HOF Higher-order function. 40

IDE Integrated Development Environment. 2

IT Information Technology. 1

JVM Java Virtual Machine. 12

LIFO Last In, First Out. 30

OOP Object Oriented Programming. 6, 8

REPL Read–Eval–Print Loop. 10

VM Virtual Machine. 11, 12

Glossario

Code Painter Prodotto software sviluppato dall'azienda Zucchetti s.p.a. per la gestione e versioning di grandi progetti software. i, 4, 52

Design By Contract Metodologia per lo sviluppo di software che si concentra sullo sviluppo di procedure corrette tramite l'utilizzo di pre condizioni, post condizioni ed invarianti per descrivere l'effetto sullo stato del programma dato dall'esecuzione di una procedura. 4, 7, 18

Garbage collector Gestione automatica della memoria che viene eseguita da una linea di esecuzione differente da quella principale che si occupa durante determinati intervalli di tempo di liberare la memoria che non può essere più referenziata. 10

Weak reference Riferimento che non incrementa o diminuisce il conteggio dei riferimenti dell'oggetto referenziato. 11