Relazione di progetto di "Paradigmi di Progettazione e Sviluppo"

Metaprogrammazione in Scala

Scafi come caso di studio

Svolto da Gianluca Aguzzi

Indice

Glossario

Funzione aggregata Una funzione è aggregata se è una definizione a funzione normale, ma scritta all'interno di un trait che estende (o che ha come self-type) il trait *Constructs*. Al suo interno possono essere usati costrutti aggregati derivati o principali.

Funzioni aggregate principali Funzioni definite a priori e delle quali si conosce il tipo aggregato di ritorno e dei parametri in ingresso.

Programma aggregato Programma dove c'è la definizione di un metodo `main` in un trait che estende il trait `ProgramSchema`. All'interno di un programma aggregato sono leciti gli usi di costrutti aggregati e, in particolar modo, ci si aspetta che esso abbia una struttura ben precisa, che differisce da ciò che scala accetta come codice corretto.

Tipo aggregato Tipo definito da aggregate programming che non corrisponde ai tipi standard definiti in Scala. I tipi descritti sono quattro, L, T e F e *ArrowType*.

1 Introduzione

La costruzione di sistemi software complessi comporta diverse problematiche, associate al divario di astrazione presente tra il dominio del problema e il linguaggio/paradigma di programmazione scelto per lo sviluppo di esso. I Domain Specific Language (*DSL*) sono nati proprio a fronte dell'esigenza di ridurre il gap presente tra queste due realtà. Un *DSL* può essere di due tipi: *External*, costruito insieme ad un interprete o ad un compilatore apposito; *Embedded* costruito come *libreria* di un linguaggio ospite, sfruttando la sua sintassi. Il linguaggio di programmazione Scala, in particolare, permette la creazione di *Embedded DSL* in modo agevole, grazie sia a delle *feature* nel linguaggio (come gli impliciti) sia alla stessa sintassi del linguaggio molto "rilassata".

La metaprogrammazione, in questo panorama, si colloca in supporto alla creazione di *Embedded DSL*, rendendo praticabile l'aggiunta di controlli, altrimenti di difficile gestione.

Questo progetto ha lo scopo di analizzare le principali tecniche di metaprogrammazione nel linguaggio di programmazione Scala, e come caso di studio, lo sviluppo di un plugin per il compilatore del linguaggio scelto, atto ad aggiungere dei controlli lessicali e di tipo al framework *Scafi*. Il lavoro sviluppato si trova in [2]

2 Meta Programmazione

La metaprogrammazione può essere vista come una tecnica che permette la costruzione di un programma che manipola/genera altri programmi. Tale programma, prende il nome di *metaprogramma*, scritto in un certo linguaggio (più precisamente, *metalinguaggio*) target.

Un chiaro esempio di metaprogramma è il compilatore; infatti esso è un programma che prende in input altri programmi (che in alcuni casi possono essere scritti nello stesso linguaggio del compilatore) per poi manipolarli e produrre un file oggetto, eseguibile.

I linguaggi di programmazione, in alcuni casi, sono anche metalinguaggi, cioè il codice scritto è capace di ispezionare la struttura di se stesso. Tali linguaggi godono della proprietà di *introspettività* (chiamata reflection in alcuni linguaggi di programmazione ad oggetti e funzionali), ossia come se il codice potesse guardare se stesso riflesso ad uno specchio.

L'introspettività, però, non è la caratteristica fondamentale per poter sfruttare la metaprogrammazione. Linguaggi come C++ non hanno alcun meccanismo di reflection, ma sfruttano un'altra tecnica chiamata *Template metaprogramming* (spiegata brevemente in seguito). Nel codice 1 si vede un esempio di template metaprogramming in C++.

```
template <unsigned int n>
struct fibonacci {
 enum {
    v = fibonacci<n - 1>::v + fibonacci<n - 2>::v
 };
};

template <>
struct fibonacci<0> {
 enum { v = 0 };
};

template <>
struct fibonacci<1> {
 enum { v = 1 };
};
```

Listing 1: Template metaprogramming in C++, computazione di Fibonacci a compile time

La metaprogrammazione ha infinite applicazioni, alcune tra le più importanti sono:

- *type checking ad hoc a tempo di compilazione*: in alcuni casi, i linguaggi di programmazione non danno abbastanza espressività per aggiungere dei controlli molto specifici. Alcune tecniche di metaprogramming permettono di aggiungere controlli sulla struttura del programma, in modo da evitare situazioni anomale;
- *manipolazione del codice sorgente, al fine di evitare errori e "boiler-code"*: questa possibilità permette di risolvere problematiche comuni nella stesura di programmi, tra tutte la serializzazione;
- *aggiunta di nuovi costrutti*: attraverso la metaprogrammazione, è possibile, in alcuni casi, aggiungere nuovi costrutti mancanti nel linguaggio

host scelto. Un esempio è quello sviluppato su scala per i costrutti async/await presenti in javascript [11], degno di nota è anche quello sviluppato da *EA* su Java [4].

2.1 Tipologie di metaprogrammazione

Ci sono due macro categorie di metaprogrammazione:

1. a run-time: un programma può accedere alla struttura del codice solamente quando viene eseguito. La Reflection di Java è un chiaro esempio di questa categoria;

2. a compile-time: un programma accede e manipola la struttura di se stesso a tempo di compilazione, rendendo possibile controlli statici riguardo la *safety* del programma stesso. Java permette l'uso di questa tipologia di metaprogrammazione con le Annotations;

In ogni categoria, poi, cadono diverse tecniche:

- *introspezione*: tecnica usata prettamente in fase di esecuzione che permette di manipolare caratteristiche statiche degli oggetti. Per farlo, il compilatore aggiunge delle informazioni alle diverse strutture sintattiche durante la compilazione;

- *macro sintattiche*: espansione di codice che va ad alterare l'abstract syntax tree (AST) di un programma;

- *template*: tecnica nella quale, il compilatore utilizza dei template per poter generare del codice, che verrà poi innestato nel codice sorgente;

- *compile time execution*: permette di eseguire codice durante la compilazione, con l'obiettivo di ottimizzare i tempi di esecuzione a run time, oppure effettuare controlli statici sulla struttura del codice.

Le tecniche non sono tra di loro esclusive, infatti esistono linguaggi di programmazione che permettono di usare tutte le precedenti categorie, altre invece, solo alcune (ad esempio, C++ permette l'uso di template ma non dell'introspezione).

Le tecniche a run-time, in generale, soffrono di alcuni difetti: possibile diminuzione di performance; problemi di accesso a strutture dati incapsulate

(si pensi ai campi private nei linguaggi ad oggetti); problemi nell'intercettazione di errori a compile time; perdita di qualità del codice, reso difficilmente comprensibile e manutenibile.

In generale, ove è possibile, si dovrebbero preferire tecniche di metaprogrammazione a compile time, in quanto non introducono cali di performance a run-time e intercettano gli errori a tempo di compilazione.

2.2 Scala

Scala, fino alla versione 2.10, si è basato sulla reflection di Java come tecnica di metaprogrammazione. Questo ovviamente aveva una serie di limiti (Java non è a conoscenza dell'esistenza dei trait, degli object..), per questo sono state introdotte un certo numero di librerie/ metodologie per poter manipolare la struttura di un programma scritto in Scala. È da notare come, tutto ciò che verrà detto in seguito, si basa su elementi ancora sperimentali, quindi potrebbero subire delle modifiche nel tempo. Delle linee guida per quello che sarà il futuro della metaprogrammazione in Scala sono riprese negli articoli [7] e [18].

Allo stato attuale, Scala permette di usare tecniche di metaprogrammazione sia a compile-time che a run-time. Entrambe si basano sulle stesse Application Programming Interface (*API*) e su delle astrazioni comuni. L'astrazione dati in comune tra tutti questi meccanismi è l'AST (fig. 1). La radice di tutti i nodi presenti nell'AST è il `Tree`, esso rappresenta un nodo dell'albero, decorato con un `Type` (descrive il tipo riconosciuto dal compilatore in quel punto del nodo, essenziale per poter fare typechecking) e descritto da un `Symbol` (definisce il simbolo al quale il nodo corrente fa riferimento, ad esempio in una chiamata a funzione, il simbolo associato ad essa sarà la definizione alla funzione chiamante). Ogni `Tree` ha un insieme di figli, che a loro volta sono dei `Tree`. In genere, nella costruzione di un certo nodo dell'albero, vi è un certo pattern di costruzione. Tutte le entità che finiscono con API sono interfacce e per ognuna di esse vi si può trovare un'implementazione. Ad esempio in `DefDefApi` vengono descritte quelle che sono le caratteristiche salienti di una definizione a metodo, per poi essere implementate in `DefDef`. Seguono le descrizioni di alcuni dei nodi principali dell'AST di Scala:

- `DefDef`: nodo che descrive una definizione a metodo, le proprietà che contiene sono: i parametri, eventuali tipi generici, il corpo (opzionale, se è un metodo astratto), modificatori di visibilità e il nome;

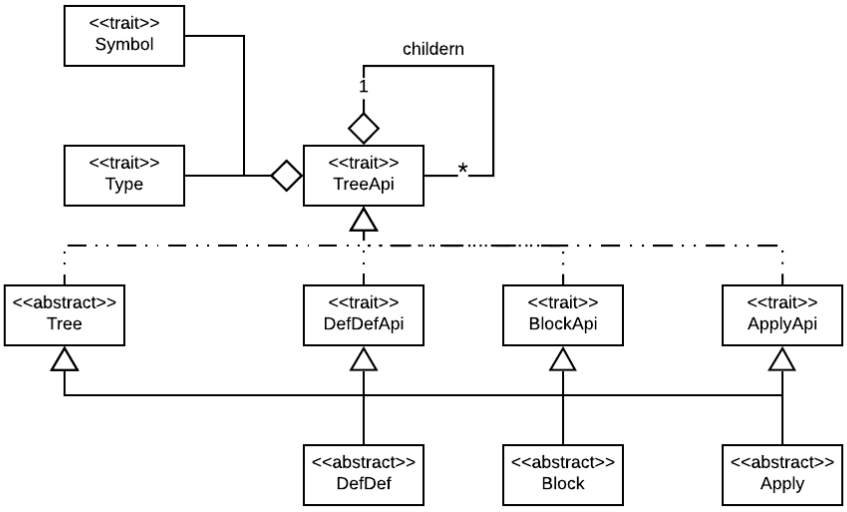

Figura 1: Descrizione semplificata della rappresentazione dell'AST in Scala

- `Apply`: nodo utilizzato per rappresentare una chiamata a metodo/funzione, contiene l'albero associato al chiamato e una lista di argomenti (strutturati come AST);
- `Block`: descrive un sequenza di istruzioni;
- `ClassDef`: nodo che descrive una definizione di una classe, contiene i modificatori di tipi, la lista dei parametri per il costruttore e il corpo della definizione della classe;
- `ModuleDef`: simile a `ClassDef` ma usato per descrivere definizioni di *modulo* che sono associate alla creazione di `object` di Scala.

L'analisi della struttura deriva da un'attenta analisi della Scala doc [9] e alla documentazione della struttura del compilatore [8].

In [1] vi sono alcuni esempi sviluppati utili alla comprensione (base) delle varie tecniche spiegate.

Reflection È la parte di scala che abilita la metaprogrammazione definendo quelli che sono i concetti principali per accedere alla struttura del programma da ispezionare. In primo luogo, per poter utilizzare la reflection

è necessario impostare un ambiente idoneo, che dipende da quando si effettuano i controlli, se a run-time o a compile time. Tale ambiente, nell'*API* di Scala prende il nome di `Universe` e gli ambienti principali sono due: `scala.reflect.runtime.universe` e `scala.reflect.macros.universe`. All'interno di questi ambienti sono presenti delle entità, che prendono il nome di `Mirrors`, che danno accesso alle proprie proprietà. Non tutti i `Mirrors` sono uguali e dipendono sia dall'entità che si sta analizzando sia dal tipo di ambiente scelto. Nei `Mirrors` creati in un ambiente a run-time, ad esempio, è possibile: creare istanze di una certa classe, accedere ai campi di una certa classe, richiedere il tipo generico di un certo oggetto... Con il codice seguente, si vede come è possibile accedere a informazioni di type a run-time sfruttando i `TypeTag`.

```
import scala.reflect.runtime.universe._
def sameType[T:TypeTag, O:TypeTag](f : T, s : O) = {
    typeOf[T] =:= typeOf[O]
}
sameType(10,20) //True
sameType(10, 20.2) //False
```

Le considerazioni scritte in precedenza sono tratte principalmente dalla documentazione ufficiale di scala [5].

Macros Basate sul reflection framework, permettono di abilitare le metaprogrammazione a tempo di compilazione. Allo stato attutale, in Scala sono presenti due tipologie di macros: *Whitebox* (hanno pieno accesso all'AST e possono alterare completamente la sua struttura) *Blackbox Macro* (sono limitate per evitare possibili problemi durante l'espansione). Scrivere una macro associata ad un metodo (chiamate *DefMacro* in Scala) è molto facile, per prima cosa si deve importare uno dei due contesti:

```
//blackbox macro
import scala.reflect.macros.blackbox.Context
//whitebox macro
import scala.reflect.macros.whitebox.Contex
```

Poi si descrive la signature del metodo che verrà espanso con una macro:

```
object MacroDef {
 def log(msg:String):Unit = macro MacroImpl.logImpl
```

```
}
```

Infine, accedendo all'AST del blocco che stiamo espandendo, è possibile implementare la struttura della macro:

```
object MacroImpl {
 def logImpl(c: Context)( msg : c.Expr[String])
                        : c.Expr[Unit] = {
  import c.universe._
  val time = q"new java.util.Date().toString"
  val logline = c.Expr[String](q""""$time+""+$msg""")
  c.Expr[Unit](q"println($logline)")
 }
}
```

Le macro devono dare in output un'espressione (`Expr`, cioè l'albero sintattico taggato con un certo `Type`). Per poter costruire tale espressione è possibile appoggiarsi o a una tecnica chiamata quasiquote (notazione pulita che permette di manipolare l'abstract syntax tree di scala, spiegata brevemente in seguito), oppure attraverso le API esposte da `Universe`. In generale, ove possibile, la notazione via quasiquote è sempre preferibile in quanto rende il codice più leggibile ed evita di esporre dettagli di basso livello associati alla rappresentazione dell'albero sintattico in Scala.

La tecnica di quasiqouting è molto potente, permette di manipolare l'AST di Scala sfruttando semplici stringhe strutturate. È possibile sia generare l'AST a partire da una stringa, che sfruttare il pattern matching sull'AST già formato. Ad esempio, scrivere:

```
q"10+20"
```

porta alla generazione di un albero così strutturato:

```
Apply(
    Select(
        Literal(Constant(10)), TermName("$plus")
    ),
    List(Literal(Constant(20)))
)
```

Il pattern matching con questa tecnica è intuitivo: per poter fare match rispetto a un albero dove si sommano due numeri, basterà scrivere:

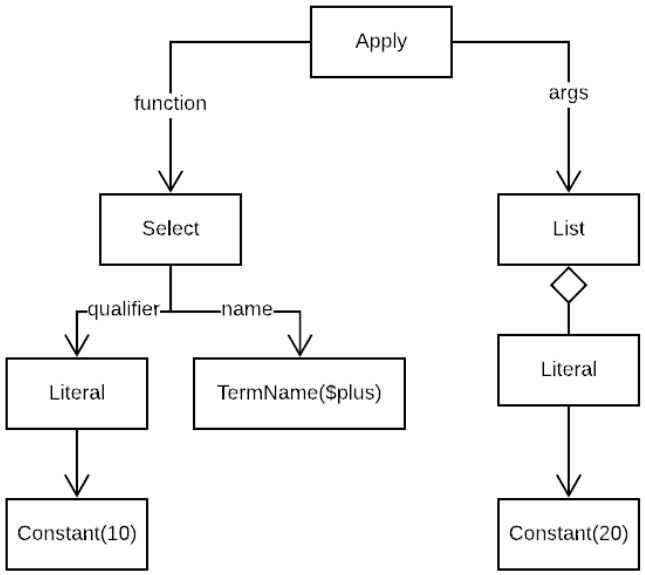

Figura 2: Rappresentazione grafica dell'AST associato all'espressione `10 + 20`

```
tree match {
    case q"$a+$b" =>
    case q"$mods val $pat = $expr" =>
}
```

In questo caso la prima clausola effettuerà match rispetto all'espressione di prima `(10 + 20)` e si potrà accedere agli alberi associati agli argomenti a e b. Un esempio più complesso è il secondo, dal quale sarà possibile, da una definizione a valore, prendere i modificatori, il nome e l'espressione di inizializzazione. Questi esempi non vogliono essere esaustivi, ma hanno lo scopo di dare un assaggio della potenzialità introdotta con le quasiquote. Nella documentazione [15] vengono spiegate in modo dettagliato e approfondito.

Nelle varie versioni di Scala sono state introdotte nuove modalità d'uso delle macro, non solo associate a definizioni a metodo. Una delle più interessanti è quella associata alle Annotations statiche (presenti cioè solo a tempo di compilazione), con esse è possibile annotare codice per poi espanderlo o controllarlo a tempo di compilazione.

Una nota importante per quanto riguarda le macro è che devono essere compilate a parte rispetto alla compilazione del progetto nel quale vengono utilizzate, questo implica una particolare impostazione del progetto sbt. In genere, si cerca di costruire due sottoprogetti, uno dipendente dall'altro. Ad esempio, nel file build.sbt del repository [1], mostrato in 2, viene evidenziata la necessità di creare due sottoprogetti sbt: uno dove verranno sviluppate le varie espansioni a macro (in questo caso chiamato macros) e un altro dove verranno effettivamente utilizzate (chiamato reflect-test). In questo caso è stato scelto di rendere il progetto principale dipendente dal progetto macros, così da avere, ad ogni compilazione, la versione aggiornata dell'implementazione delle espansioni a macro.

```
1  val commonSettings = Seq(
2      ..
3      scalacOptions ++= Seq(
4          "-Ymacro-annotations",
5          "-language:experimental.macros"
6      ),
7      scalaVersion := "2.13.1",
8      libraryDependencies += "org.scala-lang" % "scala-
          reflect" % "2.13.1"
9  )
10 val macroSetting = commonSettings
11 lazy val macros: Project = project.in(file("macros"))
12   .settings(macroSetting)
13   .settings(
14       name := "macros"
15   )
16 lazy val reflectTest: Project = project.in(file("main"))
17   .settings(commonSettings)
18   .dependsOn(macros)
19   .settings(
20       name := "reflect-test"
21   )
22
23 lazy val root = project.in(file("."))
24   .aggregate(macros, reflectTest)
25   .settings(commonSettings)
```

Listing 2: File build.set di un progetto che fa uso delle macro

Per poter sfruttare l'espansione delle macro è necessario abilitarle a livello di compilatore scala e questo è possibile attraverso le istruzioni scritte a linea 3. L'unica dipendenza che occorre per poter sviluppare macro è quella della reflection library.

Plugin Il compilatore di Scala è molto flessibile e permette la costruzione di plugin appositi atti a controllare/trasformare codice in fase di compilazione. Un plugin per il compilatore Scala non è altro che codice Scala che tratta e manipola *Tree*. Per prima cosa, è necessario capire quelle che sono le fasi principali del compilatore preso in esame. Purtroppo questo è alquanto difficile in quanto, tra una versione di Scala e l'altra, possono cambiare in modo notevole dei dettagli interni al compilatore. In Scala 2.11 quelle principali sono:

1. *parser*: in questa fase il compilatore crea un AST non tipato. Vengono identificati eventuali errori di sintassi
2. *namer*: ha il compito di risolvere i nomi e aggiunge i simboli trovati nell'AST
3. *packageobjects*: aggiunge i package object nell'AST
4. *typer*: si associa un `Type` ai vari nodi dell'AST ossia si fa inferenza dei tipi, si ricercano gli impliciti nell'albero, si effettua l'espansione delle marco, avvengono le conversioni implicite, si effettuano i controlli riguardanti il missmatch di tipo, avviene la risoluzione dell'overloading dei metodi, ...
5. *pickler*: serializzazione della symbol table, utile al momento della generazione del bytecode
6. *uncurry*: viene rimosso il currying nelle definizioni a funzione.

Per vedere la lista completa della fasi, è possibile scrivere la seguente istruzione da linea di comando:

```
scalac -Xshow-phases
```

Le fasi sono sequenziali, cioè per poter avanzare nel processo di compilazione, si deve aspettare che la fase precedente abbia completato l'intero processo su tutti i file da compilare.

È importante conoscere le varie fasi per poter collocare un plugin all'interno del processo di compilazione. Ogni fase ovviamente, aggiunge / trasforma l'AST. Ad esempio, in output dalla fase di parsing, non avremmo ancora il tipo in alcun nodo, così come i simboli.

Nuove fasi di compilazione non possono collocarsi tra qualsiasi delle fasi standard. Ad esempio, namer, packageobjects e typer sono concettualmente una sola fase che non può essere spezzata.

Con i Plugin è possibile: fare qualsiasi variazione all'AST; fare dei controlli molto più fini in quanto si ha accesso completo a tutti i file che stanno per essere compilati; introdurre nuova semantica a costrutti già presenti. In [16] vengono riassunti quelli che sono stati i vari impieghi dei plugin per il compilatore Scala, tra i quali:

- modificare il backend di compilazione, per produrre codice oggetto diverso (scalajs [12], scala native [13]);
- alterare la sintassi Scala aggiungendo nuova semantica (kind-projector [17] introduce nuovi tipi astratti, come le lambda);
- analizzare la struttura del codice (acyclic [6] evita file ciclici nel codice sorgente);
- aggiungere nuovi costrutti ristrutturando il codice (scala continuations [10]);

La costruzione di un plugin deve seguire una struttura ben precisa, dettata da Scala. Per prima cosa si deve creare una classe che estende da Plugin:

```
class APlugin(override val g: Global) extends Plugin {
  override val name: String = "<name>"
  override val description: String = "<description>"
  override val components = List(<components>)
}
```

Un concetto importante è il valore Global, esso è il contesto di compilazione corrente, e contiene tutte le definizioni dei nodi dell'AST e delle funzionalità per manipolare un albero.

Un plugin poi si compone di una sequenza di PluginComponent, nei quali verrà descritta la logica della nuove fase di compilazione. La struttura base per descrivere un componente è la seguente:

```
class AComponent(val g: Global) extends PluginComponent {
  override val phaseName: String = "<name>"
  override val runsAfter: List[String] = List("<phase>")
  override val runsBefore: List[String] = List("<phase>")

  override def newPhase(prev: Phase): Phase = {
    new StdPhase(prev) {
      override def apply(unit: g.CompilationUnit): Unit = {
       ...
      }
    }
  }
}
```

Ogni componente avrà bisogno del contesto di compilazione per poter accedere ai tipi dell'AST corretti. In più, in fase di definizione, è necessario definire il nome del componente e dove si colloca nella catena di compilazione (attraverso i valori runsAfter e runsBefore). Attraverso il metodo newPhase si va a definire la logica di una nuova fase di compilazione che verrà chiamata in base a cosa si è specificato nei campi precedenti. La classe StdPhase semplifica la creazione di una fase e permette di definire la logica di compilazione per un file alla volta.

Apply restituisce Unit, e ciò rende complessa la modifica dell'albero. Un modo per creare un componente che alteri la struttura dell'albero è il seguente:

```
class AComponent(val g : Global) extends PluginComponent
  with Transform
  with TreeDSL {
  import g._ //utile per l'uso dei tipi definiti in Global

  override protected def newTransformer(u: CompilationUnit)
  : Transformer = {
    new Transformer {
      override def transform(tree: Tree): Tree = {
        if(logic(tree)) {
          ... altera AST ...
        } else {
          super.transform(tree)
        }
```

```
      }
    }
  }
}
```

La struttura rimane simile a quella di un componente standard, tranne per il fatto che qui non si definisce il metodo `newPhase`, ma un nuovo concetto che è il `Transformer`. Tale oggetto ha l'obbiettivo di alterare l'albero nel caso in cui la condizione sia soddisfatta, altrimenti si demanda la trasformazione al padre che non farà altro che richiamare transform su tutti i figli dell'albero che si sta valutando.

Il progetto in cui si sviluppa il plugin deve essere separato rispetto al progetto in cui viene utilizzato. In più, affinché il plugin sviluppato possa essere usato, deve essere impacchettato all'interno di un jar con un file che ne descriva la struttura. In questo caso, scala richiede un file di tipo XML dove viene descritto il nome del plugin e la classe del plugin. Ad esempio, nel progetto [2], il file XML è strutturato come descritto in 3. Tale file deve essere collocato nella sottocartella di progetto resourcers.

```
<plugin>
    <name>scafiplugin</name>
    <classname>..package..ScafiDSLPlugin</classname>
</plugin>
```

Listing 3: File di descrizione del plugin richiesto da scala

Osservando il file build.sbt del progetto [2], mostrato in 4, si nota come esso risulti più semplice rispetto a quello definito per le macro.

```
...
libraryDependencies ++= Seq(
  "org.scala-lang" % "scala-compiler" % scalaVersion.value,
)
...
```

Listing 4: File build.set di un progetto per costruire un plugin del compilatore scala

Come si può notare dal Listing 4, l'unica cosa importante è la dipendenza rispetto al compilatore di scala in cui vengono definite le entità principali per poter definire un plugin. Per creare il jar dell'intero progetto, appoggiandosi al servizio di build di sbt, bastarà scrivere `sbt package`. Il jar creato si

troverà in ./target/versione di scala/. Per verificare che tutto il processo sia andato a termine, si verifichi se all'interno del jar è presente anche il file .xml.

Infine, per poter utilizzare il plugin, si deve modificare opportunamente il file sbt del progetto nel quale si vuole utilizzare. Ci sono diverse possibilità per farlo:

- sfruttare il file jar generato : una volta generato il jar, si dovranno aggiungere le istruzioni presenti al listing 5 nella configurazione del progetto sbt:

```
autoCompilerPlugins := true,
scalacOptions += "-Xplugin:<path>/<nome>.jar"
```

Listing 5: Istruzioni sbt per aggiungere il plugin via jar

- dipendere da un progetto pubblicato in un repository online (ad esempio maven central): in questo caso, se è possibile pubblicare l'artefatto software, allora si dovrà aggiungere una dipendenza come descritto nel listing 6.

```
addCompilerPlugin("<organization>" % "<name>" % "<
    version>")
```

Listing 6: Istruzioni sbt per aggiungere il plugin via repository

ScalaMeta Questo progetto ha sviluppato per primo delle tecniche di metaprogrammazione a compile-time in Scala che poi sono state introdotte nel linguaggio ufficiale. La macro, per esempio, sono state definite prima attraverso ScalaMeta. L'obiettivo principale di questo framework è quello di generare un AST da codice Scala, scritto sotto forma di stringa o caricato da file. Questo è molto utile per fare analisi del codice prodotto come autorefactoring, tools per produrre la documentazione, ecc. . In questo caso, la rappresentazione dell'AST è molto più semplice pur rimanendo *losses* (permette di tradurre l'AST meta in quello di scala e viceversa). Anche in questo caso è possibile sfruttare quasiquoting, ma, ovviamente, la struttura dell'albero prodotto sarà diversa:

```
q"10 + 20"
```

Scalameta produce il seguente albero.

```
Term.ApplyInfix(
    Lit.Int(10),
    Term.Name("+"),
    Nil,
    List(Lit.Int(20))
)
```

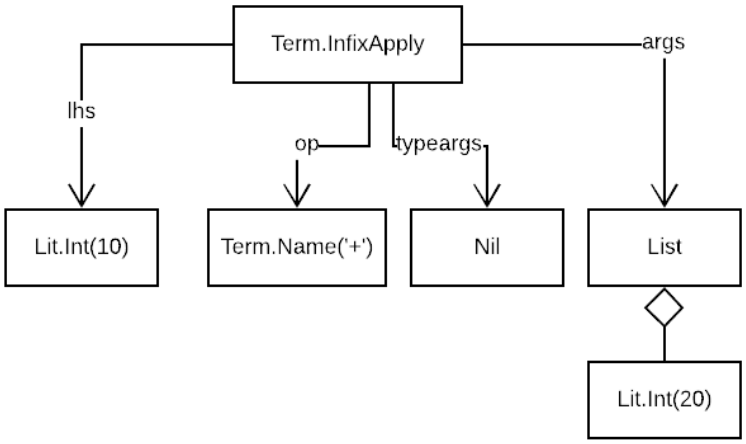

Figura 3: Rappresentazione grafica dell'AST associato all'espressione `10 + 20` nella libreria scalameta

Per ulteriori approfondimenti, rimando alla pagina ufficiale [14]. Per provare questo framework è possibile scrivere degli script su scalaFiddle al seguente link : `https://scalameta.org/docs/trees/scalafiddle.html`.

3 Progetto

Scafi è un framework scritto in Scala per la costruzione di applicazioni aggregate. Al suo interno, tra i vari componenti, viene definito un DSL che permette la definizione e l'uso dei costrutti di aggregate - programming. Nonostante Scala permetta di costruire DSL molto articolati, a volte la capacità espressiva esposta dal linguaggio non basta e vi è la necessita di aggiungere

ulteriori controlli. In questo caso, in Scafi manca una definizione dei tipi aggregati.

In aggregate programming, si potrebbero definire tre macro tipi:

- L : tipo di dato che rappresenta un dato locale al dispositivo
- F : tipo di dato che rappresenta un *campo* di valori raccolti da più dispositivi;
- T : tipo che potrebbe essere sia locale che campo.

Prendendo però la definizione `nbr`, uno dei costrutti core del linguaggio Scafi, vediamo che manca questo concetto:

```
def nbr[A](expr: => A): A
```

Questo perché, i tipi aggregati sono *impliciti* in Scafi, cioè si vuole evitare di differenziare quelli che sono tipi locali da quelli di campo per avere tutto il supporto dei tipi standard di Scala. La definizione di `nbr`, in realtà dovrebbe essere:

```
type F[A] //esiste un certo tipo campo
type L[A] //esiste un certo tipo locale
//F[A] != L[A]
def nbr[A](expr: => L[A]): F[A]
```

Uno degli obiettivi di questo progetto quindi, è stato quello di cercare di aggiungere controlli di tipo sopra al typesystem di scala, per permettere di scrivere programmi aggregati leciti. Infatti, un'espressione di questo tipo:

```
nbr(nbr(10))
```

usando il primo costrutto è lecita, mentre usando il secondo è errata in quanto esiste un missmatch di tipo. Nella sezione 3.1 verranno descritti in dettaglio i vari controlli da aggiungere al framework.

3.1 Requisiti

- Aggiunta di controlli specifici di correttezza all'interno di programmi aggregati (come notificare la presenza di costrutti if che dovrebbero essere evitati) (**req-if**);

- controlli di correttezza di uso dei costrutti core di Scafi (*nbr*, *foldhood*, *rep*,..); (**req-check**);
- riuscire a definire tutti i costrutti derivati da quelli principali, per poi controllarne la correttezza d'uso (**req-discover**);
- delineare le lambda utilizzate all'interno di programmi aggregati con la funzione *aggregate* (**req-wrap**);
- per risolvere queste problematiche, si dovranno sfruttare tecniche di metaprogrammazione (**req-meta**).

3.1.1 Analisi

Bisogna porre particolare attenzione a *dove* verranno eseguiti i controlli/trasformazioni che si dovranno aggiungere. Esempio palese è il **req-if** in quanto, notificare la presenza di costrutti if in qualsiasi parte del programma, sarebbe errato e confonderebbe il programmatore. Tutti i controlli quindi, hanno senso solo all'interno di un certo contesto.

Il problema più complesso da risolvere si concentra nel requisito **req-check** in quanto, per tutti i costrutti aggregati derivati, non è nota a priori la signature di tipo aggregato. Se i controlli si limitassero ad un insieme noto di costrutti, ad esempio `nbr` e `foldhood`, si potrebbe pensare di fare controlli ad hoc per ognuno di essi che individuino direttamente errori tipo `nbr(nbr(..))` oppure `foldhood(nbr)`... e così via. Si dovrà quindi trovare un modo generico per controllare se una certa chiamata a funzione è lecita o meno, ad esempio aggiungendo delle ulteriori informazioni di tipo ad ogni costrutto trovato. Qui quindi, si collega il **req-discover**: il modo con il quale ricercare e marcare i costrutti aggregati sarà cruciale.

Importante sarà la valutazione della tecniche di metaprogrammazione da usare, facendo particolare attenzione ai problemi che potrebbero introdurre. Ad esempio, sarebbe meglio evitare l'uso della *reflection* in quanto potrebbe portare a rallentamenti o errori inattesi a runtime. Il desiderio sarebbe quello di utilizzare approcci offline (a compile-time) in modo da verificare a priori la correttezza del programma aggregato. Data la potenza degli approcci offline, ci si dovrà chiedere come risolvere **req-discover**. Si potrebbe pensare di marcare l'AST associato alle varie funzioni aggregate con delle annotations che mantengono delle informazioni di tipo, oppure si potrebbe mantenere una collezione che contiene tutte le definizioni aggregate risolte. Per poter

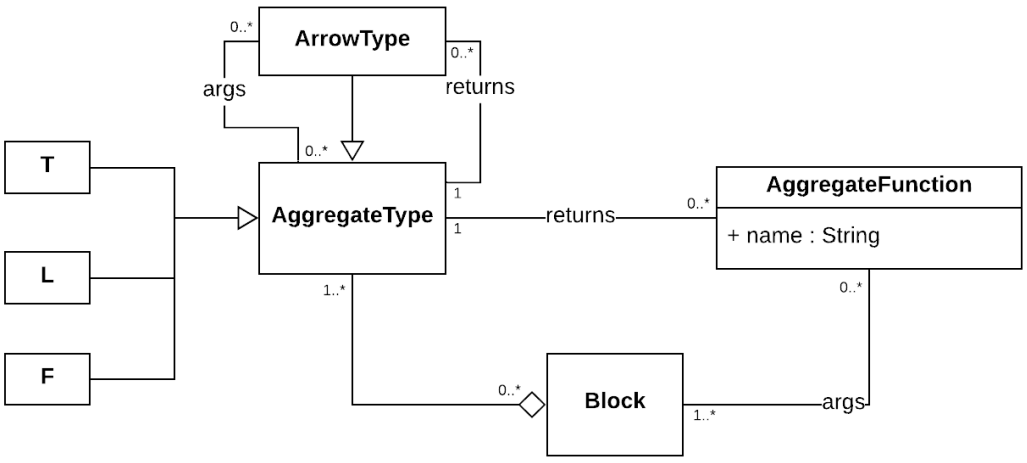

Figura 4: Descrizione della struttura dei tipi

analizzare i programmi aggregati, è giusto cosa si intende con definizione a funzione aggregata (chiamata in questo contesto `AggregateFunction`). Può essere vista come un oggetto contenente il nome della funzione, il tipo di ritorno e i parametri che accetta. Per tipo si intende in realtà un `AggregateType` che può essere `L`, `T`, `F` oppure un tipo funzione (`ArrowType`). In particolare, vista la struttura del codice Scafi, i parametri dovranno essere raggruppati in blocchi, per includere funzioni che utilizzano il currying. Ciò che è stato appena descritto è riassunto della figura 4.

Da questa astrazione delle funzioni, si dovranno riuscire a fare i vari controlli definiti in **req-check**.

3.2 Progettazione

Per poter soddisfare i requisiti imposti dal progetto, si è optato, come strumento di metaprogrammazione, per la costruzione di un plugin del compilatore in quanto:

- permette maggior flessibilità rispetto agli altri meccanismi descritti;
- garantisce l'accesso a *tutti* i sorgenti in fasi di compilazione, utile per risolvere il **req-discover**;

- permette di modificare completamente la struttura del codice (necessario per il **req-wrap**);

L'idea è stata quella di suddividere i vari compiti in componenti specifici del compilatore:

- `TransformComponent`: soddisfa ciò che viene richiesto dal requisito **req-wrap**, cioè analizzando il codice dei programmi aggregati, dovrà verificare se sono presenti lambda e wrapparle con una chiamata a funzione `aggregate`;
- `DiscoverComponent`: ha il compito di andare a scovare tutte le definizioni di funzioni aggregate all'interno della compilazione corrente;
- `TypeCheckComponent`: ha il compito di verificare la correttezza alle chiamate a funzioni aggregate (**req-check**) e verifica la correttezza del codice aggregato (**req-if**).

I componenti devono operare nello stesso contesto di compilazione, ed in più, per poter svolgere i compiti prefissati, necessitano di ulteriori informazioni riguardanti le definizioni della varie funzioni aggregate. Dovrà esistere perciò, un contesto (chiamato `ComponentContext`) condiviso da tutti e tre i componenti, creato dal plugin stesso che manterrà:

- istanza dell'oggetto global: necessaria per la creazione del componente stesso, imposta dalla struttura del compilatore di Scala;
- nome del trait dove vengono definiti i programmi aggregati;
- nome del trait dove vengono definite ulteriori funzioni aggregate;
- insieme di funzioni aggregate da controllare;

Da notare che, l'insieme delle funzioni, potrebbe evolvere del tempo in quanto, DiscoverComponent, potrebbe aggiungerne altre trovate durante la compilazione. La struttura che dovrà avere il plugin viene descritta in fig. 5:

Ogni componente dovrà essere disattivabile attraverso opzioni passate al compilatore.

È necessario scegliere in che punto della compilazione si devono collocare i componenti. Sicuramente, il `TypeCheckComponent` dovrà essere l'ultimo

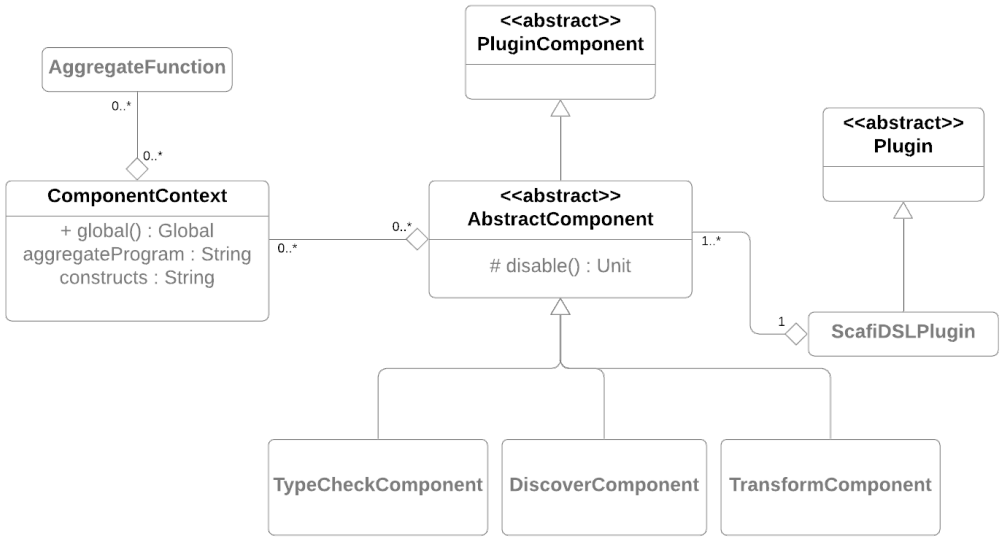

Figura 5: Struttura principale del plugin creato

della catena di questi tre elementi in quanto necessita delle informazioni ricavate da `DiscoverComponent`. Andando a collocarli nelle fasi di Scala, il `TransformComponent` sarebbe preferibile inserirlo direttamente dopo la fasi di parser, così da poter modificare il codice senza preoccuparsi della gestione dei simboli, mentre `TransformComponent` e `DiscoverComponent` per funzionare devono lavorare su un AST tipato e decorato, quindi si collocheranno sicuramente dopo la fase di typer. In modo riassuntivo, la nuova catena di fasi creatasi è riassunta in figura 6.

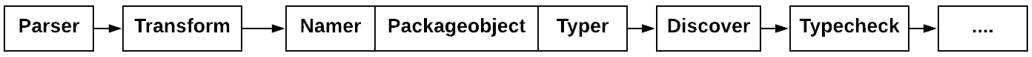

Figura 6: Nuova catena di compilazione creata con lo sviluppo del plugin

3.3 Implementazione

In questa fase, si è cercato di implementare quelle che erano le specifiche richieste dal progetto. In prima battuta, è stato necessario definire come poter navigare l'albero e come ricavare le informazioni richieste. Per poter ricercare,

all'interno di un AST, tutti gli AST che sottostassero ad una certa struttura, è stato implementato un algoritmo in comune, posto in `AbstractComponent` e così definito:

```
protected def search(root : Tree)
    (extr : Tree => Option[Tree]) : List[Tree] = {
    //extr sta per extractor
    extr(root) match {
      case None => root.children.flatMap(search(_)(extr))
      case Some(tree) => List(tree)
    }
  }
}
```

`extr` è una funzione semplice che estrae da un AST un altro AST se esso contiene le informazione richieste, altrimenti il risultato è None. Attraverso una definizione ricorsiva è stato possibile ricercare tutti gli AST che soddisfacessero la richiesta dell'`extr`, un esempio d'uso potrebbe essere:

```
search(tree) {
    case Block(stats, exp) => exp
    case _ => None
}
```

In questo caso, si cercheranno tutti gli AST `Block` (sequenza di istruzioni) presenti nella radice e si estrarrà solamente l'ultima espressione.

Un'altra operazione importante è stata quella di valutare come gestire le chiamate a metodo in presenza di currying. In Scala, una definizione così fatta:

```
def func(v1 : Int)(v2 : String) : String = {
    v1 + v2
}
```

può essere usata nel seguente modo:

```
func(10)("ciao")
```

In realtà, questa chiamata a funzione si trasforma in:

```
//------caller-----
(func.apply(10)).apply("ciao")
```

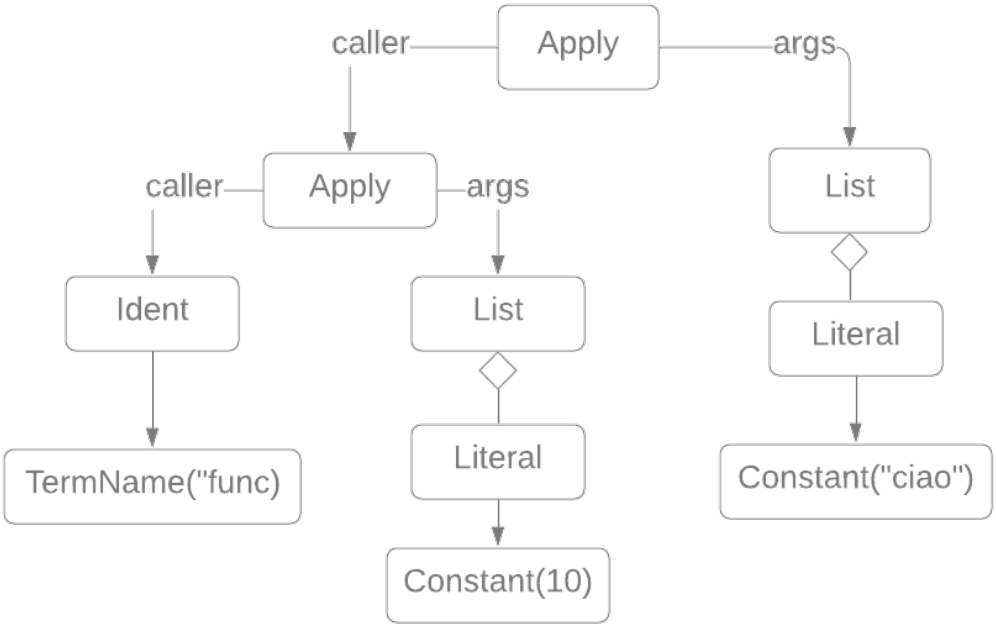

Figura 7: Rappresentazione dell'AST di espressione che fa uso di currying

La rappresentazione dell'AST associata a tale codice è descritta in figura 7. la radice dell'albero è un `Apply`, dove il caller corrisponde a `func.apply(10)` e come args ha `"ciao"`. Per accedere al valore 10, si deve espandere l'albero di sinistra e poi prendere nuovamente args. Per fare ciò, è stata creata una funzione il cui obiettivo è quello di effettuare *uncurrying* per risalire alla i-esima chiamata a funzione:

```
def uncurrying(apply : Apply, uncurryTimes : Int)
    : Option[Apply] = (uncurryTimes,apply) match {
    case (0,_) =>
        Some(apply)
    case (n, Apply(fun : Apply, _)) =>
        uncurrying(fun, n -1)
    case _ =>
        None
}
```

Per quanto riguarda `ComponentContext`, è stata presa una scelta specifica su *come* dovessero essere rappresentati i nomi dei vari costrutti

aggregati.

```
package.nomiDiClassi.nome
```

Questo per evitare omonimie di concetti definiti in package diversi.
Infine, tutto ciò che viene definito come programma aggregato, si trova all'interno di una funzione main con una specifica signature. Per poter verificare se una particolare definizione di metodo sia un main *aggregate* sono stati utilizzati i quasiqoutes e il pattern matching:

```
def isAggMain(defDef : DefDef) = defDef match {
  case q"override def main() : $tpe = $body" =>
    true
  case _ => false
}
```

Nella sezioni seguenti, verranno spiegate le principali scelte implementative attuate nei vari componenti del plugin sviluppato.

3.3.1 TransformComponent

Questo componente si occupa di effettuare un wrapping delle lambda trovate nella definizione di un programma aggregato. Per farlo, prima di tutto, si dovranno estrarre gli AST associati alla definizione di un main, su di essi poi si dovranno trovare tutte le lambda di cui verrà effettuato un wrapping, andando a trasformare l'AST. Per farlo, si è sfruttata la tecnica dei quasiqoute insieme al pattern matching. In particolare, l'albero destrutturato a partire dalla stringa

```
q"(..$args) => { ..$body }"
```

permette di accedere agli argomenti della lambda e al suo body. Per trasformare questa lambda basterà sfruttare di nuovo i quasiqoute, ma questa volta per generare un AST:

```
q"(..$args) => aggregate{ ..$body }"
```

Si dovrà fare particolare attenzione a non rifare wrapping di una lambda che ha già usato aggregate. In definitiva, il core della logica di questo componente è riassunta in queste linee di codice:

```
tree match {
```

```
    case q"(..$args) => aggregate($body)" =>
        super.transform(tree)
    case q"(..$args) => { ..$body }" =>
        q"(..$args) => aggregate{ ..$body }"
    case _ =>
        super.transform(tree)
}
```

3.3.2 DiscoverComponent

In questo componente, per poter marcare tutte le definizioni di funzione, non sono state utilizzate annotations, ma semplicemente ci si è appoggiati su di una struttura dati esterna nella quale si è mantenuto il mapping tra simboli e definizioni delle funzioni aggregate. Tale struttura viene mantenuta e aggiornata nella classe `ComponentContext`.
DiscoverComponent si divide in due fasi. Nella prima fase si ricercano tutte le definizioni di funzioni nei trait/ classi che estendono o che hanno come self type il nome del trait salvato in `constructs`. Tali definizioni non hanno ancora un tipo aggregato associato.
Nella seconda fase, si cerca di dare un tipo a tutte le definizioni di funzione trovate: si va a controllare l'AST di ogni definizione e quando all'interno vi è un'altra chiamata a funzione (nodo `Apply`) si controlla se è una possibile funzione aggreggata. In caso affermativo ci sono due possibilità: la funzione è già stata risolta e quindi ha un tipo aggregato, la funzione non è ancora risolta. A questo punto, si è deciso di intraprendere un approccio depth-first, cioè, quando si trova una funzione non risolta, si tenta di risolverla. Quando si è trovato il tipo della chiamata a funzione, si verifica se sono stati utilizzati gli argomenti della funzione che si sta valutando riuscendo così a dare un tipo a tale argomento. Alla fine della valutazione, si verifica se i vari tipi trovati per un dato argomento sono consistenti (ad esempio, se viene associato sia L che F vuole dire che non sono compatibili, se invece vengono associati L e T, è corretto). Infine, per valutare il tipo di ritorno, si cerca di estrarre dall'ultima istruzione il tipo aggregato. In tutti i casi nei quali non è possibile fare assunzioni sul tipo, viene attribuito il tipo `T`.
Lo schema per risolvere le varie definizioni a funzione è riassunto in seguito:

```
def resolveReturnType(tree : Tree) : AggregateType = {
```

```
...
}
def resolveArg(argDef : Tree) : AggregateType = {
...
}
val args = funDef.vparamss
  .map(params => block(params.map(resolveArg):_*))
  .filter(_.nonEmpty)

val returnType = resolveReturnType(funDef.rhs)
AggregateFunction.fromSymbol(
    funDef.symbol,
    returnType,
    args
)
```

L'algoritmo è riassunto nel diagramma delle attività in figura 8

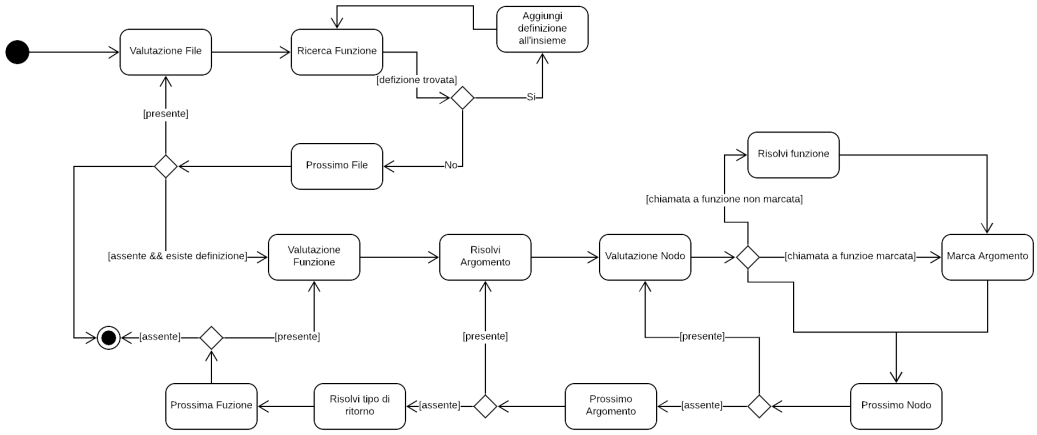

Figura 8: Diagramma delle attività che riassume la logica di ricerca delle funzioni aggregate

3.3.3 TypecheckComponent

Anche questo componente è composto da due fasi, una nella quale si verifica la presenza di if all'interno del programma aggregato, e nell'altra si verifica la correttezza d'uso delle funzioni aggregate, effettuando un

typechecking parallelo a quello di Scala. La prima parte, sfruttando la struttura dell'albero in Scala, è stata implementata nel seguente modo:

```
tree match {
  case If(_,_,_) =>
    warning(tree.pos, ifInfoString)
    tree.children.foreach(ifPresenceCheck)
  case _ =>
    tree.children.foreach(ifPresenceCheck)
}
```

Nella fase di typecheck vera e propria, si andranno a cercare tutte quelle che sono chiamate a funzione, se una di esse è stata marcata durante la fase di discover o faceva parte dell'insieme iniziale, dovrà essere controllata. Per farlo, si valuta ogni argomento della funzione e se ha un tipo inconsistente allora viene lanciato un errore. Il controllo dell'albero di ogni argomento è così fatto: ogni argomento può essere un blocco di istruzioni, per ognuna di esse si verifica che non esista un tipo inconsistente rispetto all'argomento passato. Questo punto in particolare, dovrà essere valutato per capire se è il modo giusto per verificare incorrettezze di tipo. Ad esempio, per verificare che il tipo di di un certo argomento è un campo, il controllo che si effettua è il presente:

```
//check if exists a call that return a field
private def isFieldPresent(tree : Tree) : Boolean =
   {
  tree.children.map(extractAggregateFunction)
    .collect { case Some(aggFun) => aggFun }
    .exists(_.returns == F)
}
```

Dove il metodo extractAggregateFunction, preso un albero, estrae una definizione a funzione aggregata se il simbolo del `Tree` è stato associato ad una funzione aggregata.

3.3.4 Risultato

Impostando il progetto Scafi come spiegato su [2] i controlli aggiuntivi avvengono come atteso. Ad esempio, scrivendo questo codice all'interno di un programma aggregato:

```
def intNbr(i : => Int) : Int = nbr(i)
if(Math.random() > 0.5) {
  nbr(intNbr (10))
} else {
  intNbr(20)
}
```

il compilatore, in output, riporta i seguenti logs:

```
Information:(42, 9) resolved:sims.standard.
   BasicProgram.intNbr(L): F
    def intNbr(i : => Int) : Int = nbr(i)

Warning:(43, 5) if not allowed in aggregate main
    if(Math.random() > 0.5) {

Error:(44, 18) it.unibo.scafi.core.Semantics.
   ConstructsSemantics.nbr(L): F wrong type:
   expected L but found F
      nbr(intNbr (10))
```

3.3.5 Note a termine

L'insieme di funzioni aggregate iniziale non è configurabile via opzioni del plugin. Per poter aggiungere nuove definizioni, si deve andare a modificare il codice presente in `ScafiDSLPlugin`.

3.4 Testing

Per poter verificare il corretto comportamento del plugin sviluppato, è stato creato un ambiente di testing sfruttando direttamente le *API* del compilatore di Scala. È stato possibile creare una piattaforma di compilazione (`ScafiCompilerPlatform`), con la quale, in modo programmatico durante gli unit test, si potesse passare del codice sotto forma di stringa, e compilarlo, al fine di visualizzare eventuali errori emersi durante la compilazione. Una volta costruito questo framework, si è seguito uno sviluppo Test Driven Development (*TDD*). Per ricreare uno schema simile al progetto Scafi ed evitare di compilare ogni volta l'intero progetto,

si è scelto di ricreare la struttura principale contenente i costrutti aggregati *core* per poi usare tale struttura come base della compilazione.
La piattaforma espone due metodi: `compile` che, prendendo in input una stringa che codifica codice in Scala, effettua l'intera compilazione e `transform` che invece effettua la compilazione fino alla fase di transform. Entrambi i metodi restituiscono un oggetto di tipo `CompilationReport` (transform in più restituisce il codice modificato dal compilatore). La struttura generale richiesta ai test è descritta nella classe `PluginTest`. Con questa struttura è stato possibile costruire test che valutassero l'effettiva efficacia del plugin. Di seguito viene mostrato il test che verifica il funzionamento del check sul costrutto `nbr`:

```
val nbrSig = aggFun(
    "it.unibo.scafi.core.Language.Constructs.nbr",
    F,
    args(block(L))
)
"Scafi plugin" should "raise an error if there are
   nested nbr" in {
  val nestedNbr = compiler.compile(
    writeInMain(
    """
      |nbr{nbr{10}}
    """.stripMargin
    )
  )
nestedNbr.hasErrors shouldBe true
```

4 Conclusioni

Le tecniche di metaprogrammazione permettono di gestire una complessità tale che, attraverso le normali tecniche, sarebbe intrattabile. Detto ciò però, il risultato ottenuto al termine di questo lavoro, deve essere visto come un'esplorazione. Allo stato attuale non può essere definito come un lavoro terminato e dovrebbero essere rianalizzate le varie scelte intraprese per soddisfare i requisiti posti. Può comunque essere usato come punto di partenza per la creazione dei controlli richiesti. In particolare, dovrebbe

essere rivalutata la scelta della rappresentazione delle funzioni aggregate. Molti aspetti inizialmente non sono stati trattati, pensando di dover affrontare un problema più semplice. Durante lo sviluppo, sono state riscontrate diverse problematiche, tra le principali:

- overloading: inizialmente, le funzioni venivano identificate attraverso il loro nome completo, non valutando la presenza di possibili nomi ripetuti. Per sopperire alle limitazioni imposte da tale scelta, è stato utilizzato come ulteriore identificativo il simbolo dell'AST dove viene definita la funzione, che rimane univoco per ogni definizione.

- marking delle funzioni: le funzioni vengono marcate sfruttando strutture esterne all'AST. Si dovrebbe ripensare a questa scelta, e verificare se fosse possibile marcare direttamente i nodi dell'albero, sia le definizioni che possibili valori.

Infine, nel repository [3] vengono sottolineato quelli che sono i casi coperti attualmente dal plugin e quelli che ancora non è stato possibile coprire.

4.1 Sviluppi futuri

A fronte di quanto detto prima, futuri lavori potrebbero comprendere:

- aggiungere controlli di typecheck più fini: al momento, per motivi di tempo, non sono state effettuate valutazioni di tipo aggregato su ogni AST. Questo però è essenziale in quanto, in molti casi, con un controllo superficiale non si riesce ad intercettare situazioni non lecite, ad esempio:

```
nbr(
  mux(true)(nbr(10))(10)
)
//oppure
val x = nbr(10)
nbr(x)
```

- rifinire il wrap delle funzioni aggregate: al momento, il wrapping è eseguito nella fase direttamente successiva a quella del parser, ossia dove ancora non sono presenti informazioni di simbolo e di tipo.

Bisognerebbe valutare l'inserimento del costrutto `aggregate` nelle fasi successive al parser e verificare la fattibilità di tale operazione;

- valutare l'uso di annotations per marcare i simboli con i tipi aggregati: così facendo, non vi è la necessità di andare a verificare i file sorgenti a mano, ma ogni simbolo si porta dietro quest'altra informazione;
- permettere la serializzazione/ deserializzazione della signature della funzioni trovate: questo potrebbe essere un'alternativa per fare in modo che si possa mappare un simbolo in una certa funzione;
- valutare l'approccio di discover: in generale sarebbe da chiedersi se, un approccio di risoluzione di tipo fatto in questo modo, sia esauriente.

Riferimenti bibliografici

[1] G. Aguzzi. Pps-19-reflection-example. https://github.com/cric96/PPS-19-reflection-example.

[2] G. Aguzzi. Pps-19-scafi-plugin. https://github.com/cric96/PPS-19-scafi-plugin.

[3] G. Aguzzi. scafi-plugin-application. url=https://github.com/cric96/scafi-plugin-application.

[4] Electronicarts. electronicarts/ea-async. https://github.com/electronicarts/ea-async.

[5] P. H. Heather Miller, Eugene Burmako. Scala documentation, reflection. https://docs.scala-lang.org/overviews/reflection/overview.html.

[6] Lihaoyi. lihaoyi/acyclic. https://github.com/lihaoyi/acyclic.

[7] M. Odersky and N. Stucki. Macros: the plan for scala 3. https://www.scala-lang.org/blog/2018/04/30/in-a-nutshell.html.

[8] Scala. Scala compiler 2.12.4. https://www.scala-lang.org/api/2.12.4/scala-compiler/scala/tools/nsc/index.html.

[9] Scala. Scala reflection library 2.12.4. https://www.scala-lang.org/api/2.12.4/scala-reflect/scala/reflect/index.html.

[10] Scala. scala/scala-continuations. https://github.com/scala/scala-continuations.

[11] Scala. scala/scala-async. https://github.com/scala/scala-async, Feb 2020.

[12] Scala-Js. Scala.js. https://www.scala-js.org/.

[13] Scala-Native. scala-native/scala-native. https://github.com/scala-native/scala-native.

[14] Scalameta. Tree guide. https://scalameta.org/docs/trees/guide.html.

[15] D. Shabalin. Scala documentation, quasiquotes. https://docs.scala-lang.org/overviews/quasiquotes/intro.html.

[16] S. Tisue. Scala documentation, plugin. https://docs.scala-lang.org/overviews/plugins/index.html.

[17] Typelevel. typelevel/kind-projector. https://github.com/typelevel/kind-projector.

[18] Ólafur Páll Geirsson. Roadmap towards non-experimental macros. https://www.scala-lang.org/blog/2017/10/09/scalamacros.html.