Università degli Studi di Torino

Dipartimento di Informatica

Corso di Laurea in Informatica

Tesi di Laurea in Informatica

Magda, un linguaggio con protocollo di inizializzazione modulare: un'implementazione Haskell

Candidato:

Michael Orrù

Relatrice:

Viviana Bono

Sessione di Luglio 2021

DICHIARAZIONE DI ORIGINALITÀ

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non aver plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Sommario

1 Introduzione **1**

1.1 Il linguaggio mixin-based Magda 1

1.2 Debolezze dell'object-oriented programming 2

2 Esempi di codice Magda **4**

2.1 Esempio 1 4

2.2 Esempio 2 5

2.3 Esempio 3 6

2.4 Point e ColorPoint 7

3 Semantica operazionale di Magda **8**

3.1 Notazione 9

3.2 Semantica dell'espressione new 10

3.3 Semantica di un ini-modulo 12

3.3.1 Termine dell'esecuzione 12

3.3.2 Modulo scartato 12

3.3.3 Modulo eseguito 13

4 Moduli di inizializzazione **14**

4.1 Modello della memoria esteso 14

4.2 Attivazione di un ini-modulo 15

4.3 Struttura degli ini-moduli 16

4.4 Analisi sintattica 17

4.4.1 Sintassi per creazione di un nuovo oggetto 18

4.4.2 Dichiarazione di un ini-modulo 18

4.4.3 L'istruzione super[...] 20

4.5 Type checking 21

4.5.1 Attivazione dei moduli required 21

4.5.2 Correttezza dell'espressione new 23

4.5.3 Correttezza di un ini-modulo 25

4.5.4 Correttezza dell'istruzione super 28

4.6 Valutazione 29

4.6.1 Creazione di un nuovo oggetto 29
4.6.2 Inizializzazione di un nuovo oggetto 31

5 Sviluppi futuri **36**

Bibliografia **39**

1 Introduzione

L'elaborato qui presentato si basa sulla tesi di dottorato di Jarosław Kuśmierek, quale contiene la definizione formale del linguaggio Magda [Kus10], sulla tesi di Laurea Triennale di Mattia Fumo, che contiene l'interprete Magda-Haskell, base di partenza di questo lavoro [Fum19] e sulle estensioni di Gioele Tallone apportate all'interprete, grazie alle quali è possibile svolgere un controllo di tipo sulla corretta formulazione di un programma Magda [Tal20]. L'interprete di Mattia Fumo è stato esteso in modo che possa gestire il protocollo di inizializzazione di Magda, vedere Capitolo 4, basato su costrutti, diversi dai più classici costruttori che si è abituati ad usare programmando con protocollo orientato agli oggetti, chiamati moduli di inizializzazione. Viene inoltre ripreso ed esteso il lavoro di Gioele Tallone. L'interprete, a seguito delle estensioni descritte nel Paragrafo 4.5, ora, è in grado di verificare, attraverso una serie di controlli, se le espressioni, le istruzioni ed i moduli di inizializzazione coinvolti nel processo di creazione di un oggetto sono ben formulati. In particolare, dopo averne definito la struttura, sono state implementate le operazioni di parsing, di type checking e di valutazione sugli ini-moduli. Il processo di parsing avviene all'interno del modulo Parser.hs, esso ha il compito di tradurre il programma scritto in Magda in un equivalente programma scritto in codice Haskell. Successivamente l'operazione di type checking, realizzata dal modulo TypeCkeck.hs, svolge un controllo a tempo statico sul programma Haskell per verificare che il programma di partenza sia ben formato. In conclusione il programma Haskell verrà valutato, il modulo Eval.hs eseguirà il codice del programma.
Sono stati, altresì, espansi i concetti di configurazione , contesto di valutazione e contesto di type checking di un programma Magda, per tenere in considerazione gli ini-moduli.

1.1 Il linguaggio mixin-based Magda

Magda è un linguaggio di programmazione orientato ai mixin. Generalmente, un mixin viene definito come un contenitore di metodi o anche come una classe incompleta, che può essere completata da una classe tramite un'operazione di

applicazione del mixin alla stessa. Le due operazioni definite sui mixin sono :

- Applicazione: un mixin viene applicato ad una classe per ottenere una sottoclasse.
- Composizione: più mixin possono essere composti tra di loro per creare concetti più complessi.

In alcuni linguaggi orientati ai mixin è presente anche l'ereditarietà, che però è ortogonale all'applicazione e alla composizione. In magda i mixin hanno anche il ruolo delle classi e sono organizzati in gerarchie. Le operazioni sui mixin sono quindi la composizione, l'ereditarietà e l'istanziazione di oggetti. Inoltre l'implementazione di Magda permette di importare librerie di mixin. In questo paragrafo ci concentreremo sulla struttura di un programma Magda, in particolare, sulla definizione di mixin e inizializzazione dei nuovi oggetti.
Possiamo suddividere un programma Magda in due sezioni:

- Definizione di mixin.
- Programma principale.

Ogni definizione di mixin della forma:

mixin *MixinName* **of** *MixinExpression*
field:MixinExpression; ...
MethodDeclaration; ...
IniModuleDeclaration; ...
end;

si compone di tre liste di dichiarazioni, opzionalmente vuote, che sono campi, metodi e moduli di inizializzazione. In Magda ogni oggetto viene creato da una sequenza non vuota di mixin. I legami di parentela tra mixin vengono indicati nell'espressione *new[...]*. È importante ricordare che ogni mixin, per default, discende dalla radice dell'albero dei mixin, ovvero da *Object*.

1.2 Debolezze dell'object-oriented programming

Uno degli obiettivi finali della programmazione object-oriented dovrebbe essere il riuso del codice. Questo permetterebbe un risparmio in termini di scrittura e chia-

rezza in termini di lettura. Tuttavia, attualmente, questo obiettivo non è ancora completamente raggiunto. Le principali limitazioni che hanno ispirato la creazione di un linguaggio di programmazione mixin-based sono le seguenti:

Protocollo di inizializzazione non modulare.

Nella maggior parte dei linguaggi OO, l'inizializzazione degli oggetti è delegata ai cosiddetti costruttori. Un costruttore è una struttura monolitica, le sue responsabilità vengono eseguite in un unico blocco di codice. La responsabilità di eseguire chiamate a super(...), all'interno di un costruttore, permette di muoversi attraverso la struttura delle classi, lo svantaggio è la necessità di aggiungere tutti i parametri necessari all'inizializzazione delle classi superiori, nella firma dei costruttori delle classi inferiori. Tutto ciò porta ad un lavoro maggiore nel momento in cui diventa necessario modificare una classe. La moltiplicazione dei costruttori, a seguito di composizione di classi con molteplici opzioni di inizializzazione, è il problema principale che i moduli di inizializzazione vogliono risolvere.

Il seguente esempio aiuta a capire meglio l'ultimo concetto:
Un punto ha la possibilità di essere inizializzato con coordinate polari o con coordinate cartesiane, questo impone di fornire, alla classe Punto, due diversi costruttori. Ora immaginiamo una classe PuntoColorato, quest'ultima eredita la classe Punto e permette all'utente di indicarne il colore in due modi diversi, RGB o esadecimale. Per dare la possibilità all'utente di creare un punto con ogni tipo di combinazione, è necessario fornire, alla classe PuntoColorato, quattro costruttori.
Il problema non si presenta se scriviamo il nostro programma in Magda. Se rappresentiamo attraverso i mixin i concetti di Punto e PuntoColorato e la loro parentela attraverso l'operazione di composizione, per come sono definiti i moduli di inizializzazione, ne veranno forniti solo due ad ogni mixin.
Nel capitolo successivo viene riportato l'esempio in codice Magda.

Possibili collisioni tra nomi di metodi.

I principali linguaggi ad oggetti offrono la possibilità di comporre classi, ad esempio attraverso l'ereditarietà o l'implementazione di interfacce. Metodi omonimi, con firme uguali o diverse, potrebbero avere significati differenti in classi diverse. I problemi, derivanti dall'omonimia, variano dalla non compilazione alle interruzioni inaspettate durante l'esecuzione. Magda utilizza *identificatori igenici* per metodi e campi, che vanno ad evitare possibili collisioni tra nomi a fronte di composizioni tra mixin.

Meccanismi di composizione limitati.

L'ereditarietà è il meccanismo che più tra tutti permette il riuso del codice e la composizione tra classi. Attualmente tutti i linguaggi OO implementano l'ereditarietà singola, ma solo un sottogruppo permette di programmare con l'ereditarietà multipla, questo perché ritenuta troppo complessa e rischiosa. Simulare l'ereditarietà multipla è possibile, ma questo comporta una duplicazione del codice o dei costruttori.

2 Esempi di codice Magda

Magda utilizza un protocollo di inizializzazione degli oggetti modulare. Questo approccio consente di dividere le responsabilità del processo di inizializzazione in piccoli e componibili costrutti chiamati moduli di inizializzazione. Gli ini-moduli sostituiscono i più classici costruttori dei linguaggi ad oggetti.
Presentiamo alcuni esempi che vogliono aiutare il lettore a capire meglio l'utilizzo pratico di questi costrutti. Alcuni di questi esempi provocano volutamente errori o eccezioni a tempo di interpretazione. Altri esempi, che illustrano l'utilizzo dei moduli, possono essere trovati qui [rep].

2.1 Esempio 1

Il seguente esempio vuole mostrare che i moduli di inizializzazione non servono solo per assegnare valori ai campi di un oggetto. È più corretto vederli come contentori di istruzioni che devono essere sempre, o opzionalmente, eseguite a seguito della

crezione di un nuovo oggetto. Un utilizzo alternativo è quello di generatore di parametri attuali da passare in seguito ad altri moduli.

```
mixin A of Object =
    field: String;

    required A (param: String) initializes ()
    begin
        this.A.field := param;
        super[];
        "required A executed".String.print();
    end

    optional A (other: A) initializes (A.param)
    begin
        super[A.param := other.A.field];
        "optional A executed".String.print();
    end
end

new A [A.other := new A [A.param := "hello"]];
```

Mostriamo inoltre, che le espressioni *new[...]* sono annidabili. Un oggetto può essere inizializzato a partire dalla creazione di un altro oggetto.

2.2 Esempio 2

Il seguente esempio è breve e banale, ma nasconde un errore che non lo farà mai terminare.

```
mixin A of Object =
    field: String;

    required A (param:String) initializes ()
    begin
        this.A.field := param;
        super[];
        new A [A.param := "hello"];
    end
end
```

```
11
12 new A [A.param := "hello"];
```

Bisogna prestare attenzione a espressioni *new[...]* presenti all'interno del corpo di un certo modulo *m*. Se l'istruzione di creazione va ad attivare *m*, si viene a creare un loop infinito. Il problema non si presenta nel caso in cui l'istruzione non vada ad attivare *m*. Dal punto di vista progettuale, si sconsiglia l'uso dell'espressione *new[...]* a top-level all'interno del corpo di un modulo. È sempre meglio inserirla in un'istruzione condizionale.

2.3 Esempio 3

Il seguente esempio vuole andare a mostrare una proprietà particolarmente utile degli ini-moduli.

```
1  mixin Test of Object =
2
3      required Test() initializes ()
4      begin
5          "Instruction 2".String.print();
6          super[];
7          "Instruction 3".String.print();
8      end
9
10     optional Test() initializes ()
11     begin
12         "Instruction 1".String.print();
13         super[];
14         "Instruction 4".String.print();
15     end
16
17 end
18
19 (new Test[]);
```

Per definizione del linguaggio Magda e quindi anche per costruzione dell'interprete, ogni ini-modulo, che si incontra durante il processo di creazione di un oggetto e che ha l'insieme vuoto come insieme di parametri di input, viene sempre eseguito, che esso sia *required* oppure *optional*. Per quanto l'esempio lo faccia sembrare un

fatto inutile è una proprietà potentissima. Questa caratteristica può essere usata tutte le volte che si vuole eseguire un'azione su un oggetto che è attualmente in creazione. Per ogni azione si evita la scrittura di un metodo che la esegue e la chiamata dello stesso.

2.4 Point e ColorPoint

```
mixin Point of Object =
  raggio : Integer; angolo : Integer;
  x : Integer; y : Integer;

  optional Point(rg:Integer; a:Integer) initializes()
  begin
    this.Point.raggio := rg;
    this.Point.angolo := a;
    super[];
    "Il punto e' stato inizializzato con coordinate polari."
      .String.print();
  end

  optional Point(coordX:Integer; coordY:Integer) initializes()
  begin
    this.Point.x := coordX;
    this.Point.y := coordY;
    super[];
    "Il punto e' stato inizializzato con coordinate cartesiane."
      .String.print();
  end

end

mixin ColorPoint of Point =
  exColor : String;
  red : String; green : String; blue : String;

  optional ColorPoint(exc:String) initializes()
  begin
    this.ColorPoint.exColor := exc;
```

```
30      super[];
31      "Il punto e' stato inizializzato con colore esadecimale."
          .String.print();
32    end
33
34    optional ColorPoint(r:String; g:String; b:String) initializes()
35    begin
36      this.ColorPoint.red := r;
37      this.ColorPoint.green := g;
38      this.ColorPoint.blue := b;
39      super[];
40      "Il punto e' stato inizializzato con colore RGB."
          .String.print();
41    end
42
43  end
44
45  new Point[]; //punto non inizializzato
46  new Point, ColorPoint[]; //punto colorato non inizializzato
47  new Point, ColorPoint[ColorPoint.exc:="#b9793d", Point.coordX:=2,
48      Point.coordY:=10];
49      //coordinate cartesiane e colore esadecimale
50  new Point, ColorPoint[ColorPoint.r:="127", ColorPoint.g:="255",
51      ColorPoint.b:="212", Point.coordX:=2, Point.coordY:=10];
52      //coordinate cartesiane e colore RGB
53  new Point, ColorPoint[ColorPoint.exc:="#b9793d", Point.rg:=20,
54      Point.a:=180]; //coordinate polari e colore esadecimale
55  new Point, ColorPoint[ColorPoint.r:="127", ColorPoint.g:="255",
56      ColorPoint.b:="212", Point.rg:=20, Point.a:=180];
57      //coordinate polari e colore RGB
```

3 Semantica operazionale di Magda

In questa sezione andremo a dettagliare come le espressioni, le istuzioni e i moduli di inizializzazione coinvolti nel processo di creazione di un oggetto vengono valutati, per farlo useremo la semantica operazionale big-step introdotta da Kahn in [Kah87]. La semantica operazionale big-step, conosciuta anche come natural semantics, è comunemente utilizzata per descrivere la semantica dei programmi

attraverso dei giudizi.
I giudizi devono essere letti nel seguente modo: "a partire da un determinato contesto, il programma produce un valore finale/ uno stato/ ...".
Le definizioni riportate in seguito descrivono solamente situazioni di successo, nessuna regola è modellata per descrivere un errore. I possibili errori riscontrabili a livello di esecuzione sono: la reference a puntatore nullo, la chiamata di metodo su un oggetto che non lo supporta e il reference ad un campo inesistente.
Gli ultimi due errori vengono controllati dal nostro interprete a livello di type checking. Per quanto riguarda il primo errore, nella semantica, non esiste nessuna derivazione, per un qualche giudizio, che dica che tale programma termini in qualche stato, a tempo di esecuzione si riceverà un errore.

3.1 Notazione

Come da convenzione useremo due nomi differenti per le assunzioni che occorrono nelle regole che seguono.
Chiameremo *premesse* o *conclusioni* tutti quei giudizi della forma :

$$... \models ... \Rightarrow ...$$

e chiameremo tutte le altre assunzioni *condizioni ausiliarie*.
I simboli che comunemente precedono la conseguenza logica ($\models$) descrivono i seguenti concetti:

- *adr*: denota un elemento dell'insieme *Addresses*. *Addresses* è l'insieme di tutti i possibili valori delle espressioni. Il valore di un'espressione può essere un indirizzo, un oggetto o *null*.

- *ctx*: denota un elemento dell'insieme *Context*. *ctx* rappresenta il contesto statico di esecuzione riferito ad un metodo o ad un ini-modulo attualmente in esecuzione.

- *env*: denota un elemento dell'insieme *Environment*. *Environment* è l'insieme in cui ogni elemento rappresenta lo stato di tutti gli identificatori locali visibili in un determinato istante d'esecuzione. *env* ha due forme:

  - Una funzione parziale che associa nomi di identificatori locali con indirizzi di memoria.
  - Una coppia della forma ($\top$, *adr*) dove *adr* è un elemento dell'insieme *Addresses*. Viene usata quando l'esecuzione di un metodo si è conclusa a seguito di un'istruzione **return**. L'indirizzo *adr* è il risultato della valutazione appena terminata. Questa forma non è mai usata per gli ini-moduli.

- *st*: denota un elemento dell'insieme *State*. *st* rappresenta una funzione parziale, che associa indirizzi al loro contenuto. Gli indirizzi contengono comunemente oggetti. *st* verrà più comunemente chiamato heap.

- *mod*: denota un elemento dell'insieme *ModDecls* e rappresenta un ipotetico modulo di inizializzazione dichiarato nel programma attualmente in valutazione.

All'interno di ogni regola, lo stato e l'ambiente, che precedono il simbolo di conseguenza logica $\models$, vengono determinati a seguito di trasformazioni avvenute in premesse precedenti contenute nella stessa regola.
Attraverso il simbolo $\Rightarrow$ rappresentiamo l'esecuzione di un generico elemento del nostro linguaggio. In particolare l'esecuzione di un ini-modulo è definita dal simbolo $\Rightarrow^{ini}$, $\Rightarrow^{I}$ rappresenta l'esecuzione di un'istruzione e $\Rightarrow^{ex}$ rappresenta l'esecuzione di un'espressione.

3.2 Semantica dell'espressione new

$$\frac{\begin{array}{c} env, ctx, st_0 \models exp_1 \Rightarrow^{ex} (st_1, adr_1) \quad ... \quad env, ctx, st_{k-1} \models exp_k \Rightarrow^{ex} (st_k, adr_k) \\ objVal = EmptyObject(mixins) \qquad adr' = FirstEmpty(st_k) \\ adr', st_k\{adr' \mapsto objVal\} \models (IniModules(mixins), \overline{ParID \mapsto adr}) \Rightarrow^{ini} st'' \end{array}}{env, ctx, st_0 \models \textbf{new}\ Mixins[ParID_1 := exp_1, ..., ParID_k := exp_k] \Rightarrow^{ex} (st'', adr')}$$

La valutazione dell'espressione *new[.../* è dettata dalla regola sopra definita. L'insieme di premesse, raggrupate nel primo giudizio, asserisce che i parametri di

inizializzazione vengono valutati e salvati in un mappa che associa, ad ogni identificativo $parID_k$, il rispettivo valore salvato all'indirizzo adr_k, ricavato dalla valutazione dell'espressione exp_k. In accordo con il secondo giudizio vengono svolte le seguenti operazioni. Alla variabile *objVal* viene assegnato il valore di ritorno della funzione *EmptyObject*. La funzione prende come parametro una sequenza di nomi di mixin e ritorna un oggetto non inizializzato del tipo richiesto dalla sequenza di input. L'oggetto ritornato è caraterizzato da:

- una sequenza di mixins M che ne descrive il tipo;
- una mappa che associa il valore *null* ad ogni suo campo;

Il primo indirizzo disponibile per salvare l'oggetto, viene cercato e riservato. La funzione *FirstEmpty* ritorna il più piccolo indirizzo vuoto nello stato in cui il programma si trova nell'esatto momento in cui la funzione è chiamata. Un indirizzo si definisce vuoto nello stato S se nessun oggetto è associato a quell'indirizzo nello stato pari ad S. Infine, i moduli di inizializzazione vengono valutati nello stato in cui l'indirizzo *adr'* contiene *objVal*. La terza e ultima premessa indica che, l'indirizzo *adr'* conterrà il nuovo oggetto completamente inizializzato.
La lista, dei moduli di inizializzazione da valutare, consiste nell'inverso della sequenza risultante dall'applicazione della funzione *IniModules* alla sequenza di nomi rappresentata da *Mixins*. La funzione prende, come argomento, una sequenza di nomi di mixin e restituisce una sequenza con un elemento per ogni ini-modulo contenuto nella lista presa in input. Ogni elemento della sequenza risultante è una tupla formata da:

1. La sequenza dei parametri di input del modulo.
2. La sequenza dei parametri di output e dei corrispettivi tipi.
3. Il corpo dell'ini-modulo.
4. Il nome del mixin in cui il modulo è dichiarato.

Gli elementi della sequenza restituita dalla funzione occorrono nello stesso ordine dei nomi in *mixins* e vengono estratti da ogni mixin rispettando l'ordine testuale delle dichiarazioni all'interno dello stesso. Questo significa che tutti i moduli

contenuti nel primo mixin della sequenza saranno rappresentati dai primi elementi della sequenza restituita. Quando un mixin ha dichiarati al suo interno due moduli di inizializzazione, il primo modulo dichiarato precede la rappresentazione del secondo. Le definizioni delle funzioni *EmptyObject* e *FirstEmpty* possono essere trovate qui [Kus10].

3.3 Semantica di un ini-modulo

Le regole di inizializzazione di un oggetto stabiliscono come un ini-modulo deve essere eseguito e come i parametri di inizializzazione vengono prodotti e consumati dal processo. Il processo inizia in uno stato della memoria pari ad *st* e siamo a conoscenza che l'oggetto da inizializzare è contenuto all'indirizzo *adr* sullo heap. Chiameremo *mods* la sequenza di moduli da analizzare e $\overline{pars}$ l'insieme di parametri di inizializzazione che i moduli *mods* devono consumare. Il processo di inizializzazione terminerà in uno stato st' in cui l'oggetto è inizializzato (all'interno di un ini-modulo possono essere svolte operazioni diverse da quella di semplice assegnamento a campo, di conseguenza il processo potrà riflette anche altri effetti.).

$$adr, st \models (mods,\ \overline{pars}) \Rightarrow^{ini} st'$$

L'intero processo è giudato dalla sequenza dei moduli da valutare. Ogni modulo dell'insieme è controllato, la sua esecuzione è influenzata da altri fattori. Esistono tre casi.

3.3.1 Termine dell'esecuzione

$$adr, st \models ((), ()) \Rightarrow^{ini} st$$

Quando la sequenza di ini-moduli da analizzare è vuota, il processo termina. La regola richiede che anche l'insieme dei parametri di inizializzazione sia vuoto, e che quindi ogni parametro sia stato consumato.

3.3.2 Modulo scartato

$$\frac{mod|_1 \cap dom(\overline{pars}) = \emptyset \neq mod|_1 \qquad adr, st \models (\overrightarrow{mods}, \overline{pars}) \Rightarrow^{ini} st'}{adr, st \models (\overrightarrow{mods} \cdot mod,\ \overline{pars}) \Rightarrow^{ini} st'}$$

Nel caso in cui il modulo *mod* da valutare abbia un qualche parametro di input $mod|_1$ tale per cui non esiste una corrispondenza nella lista di parametri di inizializzazione, il modulo non viene valutato e il processo prosegue andando ad esaminare il prossimo modulo della sequenza.

3.3.3 Modulo eseguito

$$\frac{\begin{array}{c} mod|_1 = (Mix.ip_1, ..., Mix.ip_k) \\ \overline{pars'} = \overline{pars} \setminus^* \{Mix.ip_1 \mapsto adr_1^I; ...; Mix.ip_k \mapsto adr_k^I \\ \overrightarrow{local : T}\ \textbf{begin}\ I_1;\ \textbf{super[}opID := exp\textbf{]}; I_2\ \textbf{end;} \quad = mod|_3 \\ env = \{ip_1 \mapsto adr_1^I; ...; ip_n \mapsto adr_k^I; local_1 \mapsto null; ...; local_m \mapsto null; \textbf{this} \mapsto adr\} \\ env, mod, st \models I_1 \Rightarrow^I (env', st_0) \\ env', mod, st_0 \models exp_1 \Rightarrow^{ex} (st_1, adr_1^O) \quad ... \quad env', mod, st_{l-1} \models exp_l \Rightarrow^{ex} (st_l, adr_l^O) \\ \overline{pars''} = \overline{pars'}\ \{opID_1 \mapsto adr_1^O; ...; opID_l \mapsto adr_l^O\} \\ adr, st_l \models (\overrightarrow{mods},\ \overline{pars''}) \Rightarrow^{ini} st'' \\ env', mod, st'' \models I_2 \Rightarrow^I (env'', st''') \end{array}}{adr, st \models (\overrightarrow{mods} \cdot mod,\ \overline{pars}) \Rightarrow^{ini} st'''}$$

L'ultima regola descrive il caso in cui tutti i parametri di input ($ipar_1$, ..., $ipar_k$) del modulo *mod*, in esame, sono contenuti nella lista di parametri di inizializzazione $\overline{pars}$.

La prima condizione ausiliaria nomina *mod* il primo modulo di inizializzazione della sequenza dei moduli da valutare. In accordo con la seconda condizione ausiliaria, i parametri formali vengono mappati sui parametri attuali contenuti all'interno di $\overline{pars}$. Alla lista dei parametri di inizializzazione vengono sottratti i parametri che sono appena stati computati. In accordo con il terzo giudizio, gli elementi $\overrightarrow{local : T}$, I_1, $\overrightarrow{opID}$, *exp*, I_2 rappresentano il corpo del modulo in esecuzione. Il quarto giudizio asserisce come viene costruito il nuovo ambiente. Le variabili locali vengono inizializzate a *null* e il valore di **this** viene aggiornato con l'oggetto da inizializzare. In seguito, al quinto giudizio, le istruzioni del corpo del modulo, che precedono l'istruzione *super[...]*, vengono eseguite. Al sesto giudizio le espressioni che caratterizzano l'istruzione *super[...]* vengono valutate, successivamente, un nuovo insieme di parametri di inizializzazione viene costruito. A $\overline{pars'}$ viene sommato l'insieme ottenuto dalla precedente valutazione. In accordo con l'ottavo giudizio, il prossimo modulo verrà scelto a partire dalla sequenza $\overline{pars''}$. Al

termine del processo, le istruzioni successive alla *super[...]* vengono eseguite e la valutazione termina.

4 Moduli di inizializzazione

In questo capitolo andremo a mostrare le estensioni apportate all'interprete. Il codice completo può essere consultato qui [rep].

4.1 Modello della memoria esteso

Il principale tipo, che definisce il modello della memoria, all'interno dell'interprete, è *Config*. Nell'implementazione di Mattia Fumo [Fum19] veniva definito come una tupla di quattro elementi. La configurazione è stata estesa nel seguente modo:

$$Config : Heap \times ParamValues \times Environment \times Context \times Declaration$$

```
data Config = Config { configHeap   :: Heap
                     , configPmVls  :: ParamValues
                     , configEnv    :: Environment
                     , configCtx    :: Context
                     , configDecls  :: [Mixin]}
  deriving Eq
```

- $ParamValues = ParamIDs \rightharpoonup Addresses$: è una mappa dove ogni elemento rappresenta l'insieme dei valori di un parametro di inizializzazione passato durante il processo di creazione di un oggetto e processato durante l'inizializzazione dell'oggetto stesso. Formalmente ogni elemento di questo insieme è una funzione parziale che assegna degli indirizzi agli identificatori dei parametri.

```
type ParamValues = Map.Map String Value
```

- $Context : (Address \times String \times String) \cup (Address \times IniModule) \cup \top$ rappresenta il contesto statico di esecuzione, tenendo traccia del corrente metodo o ini-modulo attivato.
  L'attivazione di un metodo viene rappresentata attraverso:

- l'indirizzo dell'oggetto su cui il metodo è stato chiamato;
- il nome del mixin in cui si trova il metodo ed il nome del metodo;

L'attivazione di un ini-modulo viene rappresentata attraverso:

- l'indirizzo dell'oggetto che si sta inizializzando;
- il corrente modulo in esecuzione;

Il valore speciale $\top$ viene usato quando non è ancora stato chiamato un metodo o un ini-modulo.

```
data Context =
    Method {ctxThis :: Value, ctxMet :: (String, String)}
  | Module {ctxThis :: Value, ctxMod :: IniModule}
  | Top
deriving (Show,Eq)
```

4.2 Attivazione di un ini-modulo

Un modulo di inizializzazione si definisce attivato, nel processo di creazione di un oggetto, se il suo insieme di parametri di input è sottoinsieme dell'insieme dinamico dei parametri di inizializzazione, insieme rappresentato dalla mappa *ParamValues*. La lista di assegnamenti a parametri indicata all'interno delle parentesi quadre dell'espressione *new[...]* viene aggiunta alla mappa, il processo di creazione di un oggetto ha quindi inizio. Il processo continua andando alla ricerca del primo ini-modulo attivabile con i parametri correnti. La ricerca parte sempre dall'ultimo modulo dell'ultimo mixin indicato nella sequenza dell'espressione *new[...]*. Trovato un modulo attivabile dai parametri della mappa *ParamValues*, i parametri di input del modulo vengono sottratti alla mappa. Raggiunta l'istruzione *super[...]* del modulo attivato, la lista di assegnamenti a parametri di output viene aggiunta alla mappa. Il processo prosegue cercando il prossimo modulo attivabile e terminerà nel momento in cui l'interprete controllerà il primo modulo del primo mixin indicato nella sequenza di inizializzazione. Al termine del processo di inizializzazione, se il programma Magda è stato ben formato, ci si aspetta che la mappa *ParamValues* sia vuota.

Per definizione, tutti gli ini-moduli, che si incontrano durante il processo di inizializzazione e che hanno un insieme vuoto come insieme di parametri di input, vengono sempre attivati.

4.3 Struttura degli ini-moduli

La firma di un ini-modulo richiede di indicare se il modulo è:

- **required**: il modulo di inizializzazione così marcato verrà sempre eseguito durante il processo di inizializzazione;

oppure

- **optional**: l'esecuzione di un modulo optional è, come indicato dalla parola chiave stessa, opzionale. Non viene richiesto da nessuna regola che l'insieme di parametri di inizializzazione di un'ipotetica espressione *new[...]* attivi questo modulo.

Ogni ini-modulo contiene la dichiarazione di una lista di *parametri di input*, i quali verranno passati al costutto durante il processo di inizializzazione. Viene inoltre specificata una lista di *parametri di output*, questa lista definisce parametri di input di altri moduli. La parola chiave **initializes** divide i due insiemi appena descritti. Il corpo di ogni ini-modulo è composto da un'istruzione *super[...]*, racchiusa tra due insiemi di istuzioni (opzionalmente vuoti). All'interno del corpo di un ini-modulo non viene data la possibilità di utilizzare l'istruzione *return*, poichè non si prevede che un modulo ritorni un qualche risultato. In simil modo anche l'insieme delle espressioni è limitato, attualmente non è possibile eseguire l'override dei moduli di inizializzazione di un mixin di base, perciò non è consentita la chiamata a *super(...)*.
L'istruzione *super[...]* è responsabile di chiamare ulteriori moduli di inzializzazione e di fornire dei valori di input. Essi vengono specificati come parametri di output nel suddetto modulo. All'interno delle parentesi quadre è presente un assegnamento per ogni parametro di output presente nel modulo in cui l'istruzione *super[...]* è presente.
Viene riportata la sintassi formale dei moduli di inizializzazione appena descritta,

definita nel documento [Kus10]:

(* required / optional *)* *Mixin* (*Iparam1: TypeI1*; ...; *IparamN: TypeIN)
initializes (*Mixin1.Operam1; ...; MixinM.OperamM*)
VarName:**Type; ...;** *VarName***Type;**
begin
Instr **; ...**
super[*Mixin1.Operam1* := *exp* **; ...;** *MixinM.OperamM* := *exp***];**
instr **;...**
end;

Nell'interprete la struttura dei moduli di inizializzazione è stata definita nel seguente modo:

```
data IniModuleScope = IniRequired | IniOptional
  deriving Eq

type ParamAssignment = [((String, String), Expression)]

data IniModule = IniModule { moduleScope     :: IniModuleScope
                           , moduleMixin     :: String
                           , moduleInParams  :: [Identifier]
                           , moduleOutParams :: [(String, String)]
                           , moduleLocals    :: [Identifier]
                           , moduleBodyA     :: Maybe Instruction
                           , moduleSuper     :: ParamAssignment
                           , moduleBodyB     :: Maybe Instruction}
```

4.4 Analisi sintattica

In questo paragrafo analizzeremo la sintassi dell'espressione *new[...]*, necessaria per cominciare il processo di inizializzazione di un oggetto e la sintassi degli ini-moduli. La definizione completa della sintassi può essere trovata qui [Kus10] e l'analisi dei rimanenti componenti del linguaggio è definita qui [Fum19].
La definizione formale della grammatica è espressa nella forma di Bacus-Naur, dove il simbolo | rappresenta un'alternativa, le formule racchiuse in parentesi tonde sono

da intendere come opzionali e le formule racchiuse nelle (...)* sono ripetute zero o più volte.

4.4.1 Sintassi per creazione di un nuovo oggetto

```
OBJECT_CREATION ::=
     'new' MIXIN_EXPRESSION '[' PARAM_ASSIGNMENTS ']'

PARAM_ASSIGNMENT  ::= <ID> '.' <ID> ':=' EXPRESSION
PARAM_ASSIGNMENTS ::= [ PARAM_ASSIGNMENT (',' PARAMASSIGNMENT)* ]
```

L'espressione di creazione di un nuovo oggetto si compone di:

- la parola chiave **new**;
- una sequenza non vuota di mixin che definirà il tipo dell'oggetto che stiamo andando a creare;
- una sequenza di assegnamenti a parametri racchiusi tra parentesi quadre e separati da una virgola. Ogni assegnamento a parametro consiste nel: nome di un mixin, nome di un parametro e un'espressione.

```
objNew = do
      reserved "new"
      types <- typeExpr
      ps <- brackets $ sepBy inimodFieldAssign (symbol ",")
      return $ ExprNew types ps

inimodFieldAssign = do f <- inimodField
                       reservedOp ":="
                       e <- expr
                       return (f, e)
```

4.4.2 Dichiarazione di un ini-modulo

```
INI_MODULE_DECLRATION ::=
    ( 'required' | 'opzional' ) <ID>
    PARAMETER_DECLS 'initializes' OUTPUT_PARAMETERS
    INI_MODULE_BODY

```

```
INI_MODULE_BODY ::= VARIABLE_DECLARATIONS
                    'begin'
                        INSTRUCTIONS
                        MODULE_SUPER_CALL
                        INSTRUCTIONS
                    'end'

VARIABLE_DECLARATIONS ::= ( <ID> ':' MIXIN_EXPRESSION ';' )*
OUTPUT_PARAMETER  ::= <ID> '.' <ID>
OUTPUT_PARAMETERS ::=
     '(' [ OUTPUT_PARAMETER (';' OUTPUT_PARAMETER )* ] ')'
PARAMETER_DECL    ::= <ID> ':' MIXIN_EXPRESSION
PARAMETER_DECLS   ::=
     '(' [ PARAMETER_DECL (';' PARAMETER_DECL )* ] ')'
```

La dichiarazione di un ini-modulo si compone di:

- la parola chiave **required** o la parola chiave **optional**;
- il nome del mixin all'interno del quale l'ini-modulo è stato dichiarato;
- la lista dei parametri di input contenuta all'interno di parentesi tonde, dove ogni elemento è separato da punto e virgola. Ogni parametro di input è dichiarato indicando: il nome del parametro, i due punti e una lista di mixin che rappresentano il tipo del parametro di input;
- la parola chiave **initializes**;
- la lista dei parametri di output contenuta all'interno di parentesi tonde, dove ogni elemento è separato da punto e virgola. Ogni parametro di output è dichiarato indicando: il nome del mixin, il carattere '.' e il nome del parametro di output.
- il corpo dell'ini-modulo, che è composto a sua volta da:
  - una lista di dichiarazioni di variabili locali;
  - la parola chiave **begin**;
  - una lista di istruzioni separate dal simbolo ';', contenete esattamente una istruzione *super[...]*.

– la parola chiave **end**;

```
inimod p = do
  scope <- try scopeRequired <|> scopeOptional
  mixin <- p
  ps <- parens $ sepBy localId (symbol ";")
  reserved "initializes"
  initializes <- parens $ sepBy inimodField (symbol ",")
  vars <- many $ do x <- localId
                    symbol ";"
                    return x
  reserved "begin"
  body <- optionMaybe modInstruction
  super <- superStmt
  body' <- optionMaybe modInstruction
  reserved "end"
  return $
    IniModule scope mixin ps initializes vars body super body'
    where
      scopeRequired = do reserved "required"
                         return IniRequired
      scopeOptional = do reserved "optional"
                         return IniOptional

inimodField = do mix <- identifier
                 symbol "."
                 field <- identifier
                 return (mix, field)
```

4.4.3 L'istruzione super[...]

```
MODULE_SUPER_CALL ::= 'super' '[' PARAM_ASSIGNMENTS ']' ';'

PARAM_ASSIGNMENT  ::= <ID> '.' <ID> ':=' EXPRESSION
PARAM_ASSIGNMENTS ::= [ PARAM_ASSIGNMENT (',' PARAM_ASSIGNMENT)*]
```

L'istruzione *super[...]* deve essere chiamata a top-level e non all'interno di un'altra istruzione, quali possono essere cicli o istuzioni condizionali. In questo modo viene

garantito che, a conclusione dell'esecuzione del corpo del modulo, l'istruzione sarà stata eseguita solo una volta. L'istuzione si compone di:

- la parola chiave **super**;
- una lista di assegnamenti a parametri racchiusa tra parentesi quadre, ogni assegnamento è separato da punto e virgola. La lista contiene un assegnamento per ogni parametro di output dichiarato nella firma del modulo in cui l'istruzione si trova. Ogni assegnamento, a cui corrisponde un certo parametro di output p, è così formato: il nome del mixin, il carattere '.', il nome del parametro di output p, il simbolo di assegnamento ':=' e un'espressione.;

```
1  superStmt = do
2      reserved "super"
3      fs <- brackets $ sepBy inimodFieldAssign (symbol ",")
4      semi
5      return fs
6
7  inimodFieldAssign = do f <- inimodField
8                         reservedOp ":="
9                         e <- expr
10                        return (f, e)
```

4.5 Type checking

Di seguito, riportiamo tutti i controlli che vengono effettuati, a tempo statico, sulla forma delle istruzioni e dei costrutti coinvolti nell'operazione di creazione di un oggetto.

4.5.1 Attivazione dei moduli required

Nel processo di creazione di un oggetto, ogni modulo contrassegnato come **required** e contenuto nei mixin presenti nella sequenza di creazione dell'oggetto, deve essere attivato. La sequenza, che attiverà il modulo, ha origine dall'istruzione *new[...]*, durante il procedimento, ogni modulo muterà la sequenza, sottraendo i propri parametri di input e aggiungendo quelli di output.

```
--definition of function: given a set of mixin names and a set of
    initialization parameter identifier, returns a set of ini-
    modules declared within the given mixins, which are activated
    by the given initialization parameters.
activated :: [String] -> [Identifier] -> TypeCheck [IniModule]
activated names ids = do
  im <- iniModules' names
  activated' (reverse im) ids

--definition of function of support: 'activated' function main
    work
activated' :: [IniModule] -> [Identifier] -> TypeCheck [IniModule]
activated' [] _ = return []
activated' (mod : mods) pars = do
  isContained <- isContained (moduleInParams mod) pars
  case isContained of
    True -> do
      pars' <- dropInParams (moduleInParams mod) pars
      pars'' <- addOutParams (moduleSuper mod) pars'
      tail <- activated' mods pars''
      return $ mod : tail
    False -> activated' mods pars
  where
    isContained :: [Identifier] -> [Identifier] -> TypeCheck Bool
    isContained [] _ = return True
    isContained _ [] = return False
    isContained (modId : modIds) newIds = do
      b1 <- pure $ elem (idName modId)
        (map (\(Identifier name t) -> name) newIds)
      case b1 of
        True -> do
          t <- pure $ Map.lookup (idName modId)
               (Map.fromList $ map (\(Identifier name t)
               -> (name, t)) newIds)
          case t of
            Just t -> do
              b2 <- subtype (idType modId) t
              tail <- isContained modIds newIds
              return $ b1 && b2 && tail
```

```
36              Nothing -> raise "Fatal error in function isContained"
37          False -> return False
38
39      dropInParams :: [Identifier] -> [Identifier]
                     -> TypeCheck[Identifier]
40      dropInParams [] pars = return pars
41      --dropInParams pMod [] = raise "error"
42          -- This error should never be thrown. Useless case
43      dropInParams (p : pMod) pars =
44              dropInParams pMod (deletePars p pars)
45        where
46          deletePars :: Identifier -> [Identifier] -> [Identifier]
47          --deletePars _ [] = [] useless case, p must be in the list
48          deletePars p (x : xs) = if (idName p == idName x) then xs
49              else x : (deletePars p xs)
50
51      addOutParams :: ParamAssignment -> [Identifier]
52          -> TypeCheck [Identifier]
53      addOutParams xs ys = do
54        t <- mymap tcheckExpr (map (\((mix, field), e) -> e) xs)
55        names <- pure $ map (\((mix, field), e) -> field) xs
56        l <- pure $ zipWith (\n t -> Identifier n t) names t
57        return $ l ++ ys
```

4.5.2 Correttezza dell'espressione new

L'espressione di creazione di un oggetto della forma

$$new\ Mix_1; ...; Mix_n[Mix^1.par^1 := exp^1; ...; Mix^n.par^n := exp^n]$$

è corretta se, e solo se, le seguenti condizioni sono verificate:

- La sequenza $Mix_1; ...; Mix_n$ è consistente.
- Per ogni $k = 1...n$, tutti i mixin di base del mixin Mix_k sono inclusi nella sequenza $\{Mix_1; ...; Mix_{k-1}\}$.
- Tutti gli ini-moduli dichiarati all'interno dei mixin, presenti nella sequenza, marchiati come **required**, sono attivati dall'insieme dei parametri di inizializzazione.

- Dato un parametro exp^k dall'insieme dei parametri attuali $exp^1; ...; exp^n$, avente tipo T^k. Il tipo di exp^k deve combaciare con il tipo del corrispondente parametro formale.

- Ogni parametro di inizializzazione $Mix^k.par^k$, usato nell'espressione, è presente nella firma di un qualche modulo di inizializzazione dichiarato all'interno del mixin Mix^k e il tipo di questo parametro è T^k.

Riportiamo i controlli in codice Haskell che verranno eseguiti dal modulo TypeCheck.hs:

```
tcheckExpr (ExprNew t p) = do
  c <- getContext
  decls <- pure $ tcCtxDecls c
  deleteList <- deleteRepetition t
  b1 <- ordered (reverse deleteList) -- check second condition
  abs <- abstractMets deleteList
  impl <- implementsMets deleteList
  b2 <- subset abs impl
  b4 <- subset impl abs
  b3 <- overrideCoerent (reverse deleteList)
  b5 <- pure ((length deleteList) == (length t))
  -- b2 && b3 && b4 && b5 check first condition
  rmod <- rModule t
  types <- mymap tcheckExpr (map (\((mix, field), e) -> e) p)
  names <- pure $ map (\((mix, field), e) -> field) p
  ids <- pure $ zipWith (\n t -> Identifier n t) names types
  act <- activated deleteList ids
  moduleSubSet rmod act -- check third condition
  formalParams <- pure.(Map.fromList) $ map (\(Identifier name t)
              -> (name, t)) (concat $ map (\mod ->  map
              (\(Identifier n t) -> Identifier ((moduleMixin mod)
              ++ "." ++ n) t) (moduleInParams mod))
              (concat $ map mixinIniMods decls))
  tcheckParams p formalParams -- check fourth and fifth condition
  if (b1 && b2 && b3 && b4 && b5)
    then return deleteList
    else raise $ firstError [b1,b2,b3,b4,b5]
["new expression error: order of base mixin not valid",
"new expression error: some abstract method are not implemented",
```

```
"new expression error: few implement method",
"new expression error: not coerent sequence for overrides",
"repeated mixin in sequence"]
  where
    moduleSubSet :: [IniModule] -> [IniModule] -> TypeCheck ()
    moduleSubSet [] _ = return ()
    moduleSubSet _ [] = raise "Some ini-module declared in some
                          mixin as required is not activated, no
                          one module marked as activated"
    moduleSubSet (req : reqs) act = do
      case elem req act of
        True -> moduleSubSet reqs act
        False -> raise $ "Some ini-module declared in some mixin
                  as required is not activated: " ++ show req

    tcheckParams :: ParamAssignment -> Map.Map String TypeExpr
                    -> TypeCheck ()
    tcheckParams [] _ = return ()
    tcheckParams (((mix, field), e) : xs) m = do
      name <- pure $ mix ++ "." ++ field
      case Map.lookup name m of
        Nothing -> raise $ "Initialization parameter " ++ name ++
                    " without correspondence within any ini-modules"
        Just t -> do
          texpr <- tcheckExpr e
          val <- t `subtype` texpr
          case val of
            True -> tcheckParams xs m
            False -> raise $ "Wrong parameter type"
```

4.5.3 Correttezza di un ini-modulo

Le regole sotto elencate, se rispettate, definiscono la correttezza della dichiarazione di un modulo di inizializzazione *mod* all'interno di un mixin Mix_c. Indichiamo con *mods* gli ini-moduli specificati al di sopra di *mod*.

- Ogni parametro di output $(Mix_1.op_1; ...; Mix_m.op_m)$, dichiarato con tipo $T^1; ...; T^m$, del modulo di inizializzazione, trova corrispondenza in un para-

metro di input dichiarato all'interno di un altro modulo. Quest'altro modulo deve essere dichiarato all'interno dello stesso mixin o in un suo mixin di base.

- I nomi dei parametri di input, del modulo di inizializzazione, differiscono da tutti gli altri nomi dei parametri di altri moduli che sono dichiarati all'interno dello stesso mixin.
- Il tipo dei parametri di input $(T^1; ...; T^n)$ ed i tipi delle variabili $(S^1; ...; S^n)$ contengono solo nomi di mixin dichiarati all'interno del programma.
- Nel contesto dell'attuale modulo di inizializzazione, le istruzioni che ne compongono il corpo sono corrette.
- I nomi dei parametri di input del modulo, non contengono e sono diversi dalla parola chiave **this**.
- I nomi delle variabili locali, dichiarate all'interno del corpo del modulo, non contengono e sono diversi dalla parola chiave **this**.
- I nomi dei parametri di input sono diversi dai nomi usati per le variabili locali.

I controlli vengono effettuati nel seguente modo:

```
lookupMixinModule :: String -> TypeCheck [IniModule]
lookupMixinModule name = do
  mix <- lookupMixin name
  mods <- pure $ mixinIniMods mix
  return mods

tcheckModule :: IniModule -> TypeCheck ()
tcheckModule m = do
  c <- getContext
  setContext $ c { tcCtxMethodModule = Right(Right m) }
  mix <- lookupMixin $ moduleMixin m
  outPars <- pure $ map (\(name, field)->field)(moduleOutParams m)
  subtype <- base $ mixinName mix
  inRef <- inputPars subtype (mixinIniMods mix)
  tcheckOutParams outPars (map idName inRef)
```

```
16  tcheckParamsName (moduleInParams m) (concat $ map moduleInParams
      (delete m (mixinIniMods mix))) (moduleMixin m)
17  typesDef <- deleteRepetition $ (concat $ map idType
                $ moduleInParams m) ++ (concat $ map idType
                $ moduleLocals m)
18  typesProg <- pure $ map mixinName (tcCtxDecls c)
19  tcheckParamsType typesDef typesProg
20  case moduleBodyA m of
21    Just i -> tcheckInstr i
22    Nothing -> return ()
23  tcheckSuper $ moduleSuper m
24  case moduleBodyB m of
25    Just i -> tcheckInstr i
26    Nothing -> return ()
27  inParDef <- pure $ map idName (moduleInParams m)
28  localDef <- pure $ map idName (moduleLocals m)
29  differentFromThis $ inParDef ++ localDef
30  differentName inParDef localDef
31  return ()
32    where
33      tcheckOutParams :: [String] -> [String] -> TypeCheck ()
34      tcheckOutParams [] _ = return ()
35      tcheckOutParams (par : pars) list = case elem par list of
36        True -> tcheckOutParams pars list
37        False -> raise $ "Output parameter " ++ par ++ " referes
                    to unexisting parameter"
38
39      tcheckParamsName :: [Identifier] -> [Identifier] -> String
                            -> TypeCheck ()
40      tcheckParamsName [] _ s = return ()
41      tcheckParamsName _ [] s = return ()
42      tcheckParamsName (x : xs) list s = case elem x list of
43        True -> raise $ "Same name for input parameters '"
                  ++ (idName x) ++ "' in differents modules within
                  the mixin " ++ s
44        False -> tcheckParamsName xs list s
45
46      tcheckParamsType :: [String] -> [String] -> TypeCheck ()
47      tcheckParamsType [] _ = return ()
```

```
48      tcheckParamsType (x : xs) list = case elem x list of
49        True -> tcheckParamsType xs list
50        False -> raise $ "Wrong type declared " ++ x
```

4.5.4 Correttezza dell'istruzione super

L'istruzione *super[...]* è corretta, all'interno del contesto di un ini-modulo, se:

- i nomi dei parametri par_1; ...; par_n, usati all'interno delle parentesi quadre, combaciano con i nomi dei parametri di output del modulo in cui l'istruzione di trova;
- i parametri attuali exp_1; ...; exp_n hanno lo stesso tipo dei parametri di output del modulo in cui l'istruzione super si trova;

```
1  tcheckSuper :: ParamAssignment -> TypeCheck ()
2    tcheckSuper [] = return ()
3    tcheckSuper ps = do
4      c <- getContext
5      m <- pure $ tcCtxMethodModule c
6      case m of
7        Left () -> raise "super[...] instruction called outside
                     any ini-module"
8        Right (Left met) -> raise "super[...] instruction called
                             inside a method"
9        Right (Right mod) -> do
10         idOut <- toIdentifier $ moduleOutParams mod
11         matchOutPars ps idOut
12         matchExprType ps (Map.fromList $ map
13           (\(Identifier n t) -> (n, t)) idOut)
14         return ()
15   where
16     matchOutPars :: ParamAssignment-> [Identifier]-> TypeCheck ()
17     matchOutPars [] _ = return ()
18     matchOutPars (((mix, field), e) : xs) list = do
19       list' <- pure $ map idName list
20       case elem (mix ++ "." ++ field) list' of
21         True -> matchOutPars xs list
```

```
22          False -> raise $ "Undefined parameter name " ++ mix ++
                    "." ++ field ++ " used in super[...] instruction"
23
24      matchExprType :: ParamAssignment -> Map.Map String TypeExpr
                      -> TypeCheck ()
25      matchExprType [] _ = return ()
26      matchExprType (((mix, field), e) : xs) m = do
27        t <- tcheckExpr e
28        case Map.lookup (mix ++ "." ++ field) m of
29          Just tp -> do
30            b <- tp `subtype` t
31            case b of
32              True -> matchExprType xs m
33              False ->  raise $ "Undefined expression type '"
                          ++ (concat $ map (\s -> s ++ " ") t) ++
                          "' used in super[...] instruction"
34          Nothing -> error $ "Error"
35            -- This error should never be thrown. The presence of
               the field was checked in previous function
```

4.6 Valutazione

Al valutatore, che a runtime esegue il codice tradotto del programma Magda, sono state aggiunte le regole di valutazione per l'espressione *new[...]* e per gli ini-moduli.

4.6.1 Creazione di un nuovo oggetto

Il processo di valutazione dell'espressione di creazione di un nuovo oggetto della forma

$$\textbf{new}\ \mathit{Mixins}\ [\mathit{ParID}_1 := \mathit{exp}_1, ..., \mathit{ParID}_k := \mathit{exp}_k]$$

si compone di quattro step così definiti:

1. I parametri di inizializzazione attuali $\mathit{exp}_1, ..., \mathit{exp}_k$, vengono valutati negli indirizzi $\mathit{adr}_1, ..., \mathit{adr}_k$. Nell'intero processo viene utilizzato $\mathit{ParamValues} = \mathit{ParamIDs} \rightharpoonup \mathit{Addresses}$ per denotare la mappa che associa i nomi dei parametri ai relativi valori valutati in questo step.

2. Viene creato un oggetto vuoto *obj* attraverso la funzione *EmptyObject*.

3. Un indirizzo *adr'*, non utilizzato nello heap, viene riservato per eseguire lo store dell'oggetto.

4. I moduli di inizializzazione vengono valutati nello stato in cui *adr'* contiene *obj*. Il processo di esecuzione di tutti i moduli termina in uno stato in cui l'oggetto immagazzinato in *adr'* è inizializzato.

```
1  evalExpr (ExprNew types params) = do
2    evalParams params -- first step, params evaluated and
                          stored in ParamValues
3    c <- config
4    h <- pure $ configHeap c
5    obj <- emptyObject types -- second step
6    addr <- pure $ fst $ Map.findMax h
7    h' <- pure $ Map.insert (addr + 1) obj h -- third step
8    mixs <- bindMixin types
9    modseq <- pure $ reverse $ getModsFromMixin mixs
10   put $ c { configHeap = h'}
11   evalMods modseq -- fourth step
12   return $ ObjMixin (addr + 1)
13   where
14
15   getModsFromMixin :: [Mixin] -> [IniModule]
16   getModsFromMixin [] = []
17   getModsFromMixin (x:xs) =(mixinIniMods x)++ getModsFromMixin xs
18
19   evalParams :: ParamAssignment -> Eval ()
20   evalParams [] = do return ()
21   evalParams (((mix, field), e) : xs) = do
22     addr <- evalExpr e
23     c <- config
24     pv <- pure $ configPmVls c
25     pv' <- pure $ Map.insert field addr pv
26     put $ c { configPmVls = pv' }
27     evalParams xs
28     return ()
29
```

```
30  emptyObject :: TypeExpr -> Eval Object
31  emptyObject =
32    let f = \x -> Object x (Map.fromList $ createFields x) in
33      (fmap f).bindMixin
34
35  bindMixin :: TypeExpr -> Eval [Mixin]
36  bindMixin [] = do return []
37  bindMixin (m:ms) =
38    let matchMixName = \n x -> mixinName x == n in
39      do decls <- fmap configDecls config
40          mix <- pure $ filter (matchMixName m) decls
41          mixs <- bindMixin ms
42          return $ head mix : mixs
43
44  createFields :: [Mixin] -> [((String, String), Value)]
45  createFields [] = []
46  createFields (m:ms) =
47    let f = \x -> ((mixinName m, fieldName x), ObjNull) in
48      (map f $ mixinFields m) ++ createFields ms
```

4.6.2 Inizializzazione di un nuovo oggetto

Ogni modulo dichiarato all'interno di un mixin, facente parte della sequenza di inizializzazione, viene controllato. La valutazione di ogni singolo modulo è dettata dalla struttura dello stesso e dall'insieme dei parametri di inizializzazione *ParamValues*. Esistono tre casistiche:

L'inizializzazione termina. Quando la sequenza dei moduli da valutare è vuota, il processo di inizializzazione termina. La regola richiede che anche l'insieme dei parametri di inizializzazione (*ParamValues*) sia vuoto alla fine del processo. Quest'ultima condizione viene verificata in precedenza nello step di type checking.

Il modulo non viene valutato. Questa regola viene usata nel caso in cui il modulo di inizializzazione *mod*, preso in esame, abbia qualche parametro di input tale per cui non ci sia nessun valore corrispondente nella lista di parametri di inizializzazione. In questo caso *mod* non viene eseguito ed il processo continua prendendo in esame il prossimo modulo della sequenza.

Il modulo viene valutato. Il seguente caso mostra il processo da eseguire nel caso in cui la lista dei parametri di input $(ipar_1, ..., ipar_k)$, del modulo *mod* correntemente controllato, trovi corrispondenza, per intero, nella lista di parametri di inizializzazione, rappresentata da *ParamValues*. In questo caso diciamo che *mod* è attivato, il suo corpo verrà valutato e quindi eseguito. Il processo di esecuzione del modulo procede nel seguente modo:

1. La lista dei parametri di input del modulo viene sottratta dalla lista dei parametri di inizializzazione, ci si aspetta che questi parametri vengano usati all'interno del corpo del modulo.

2. Il nuovo ambiente, contenitore delle informazioni utilizzabili dal modulo, viene costruito in tre parti:

   - I nomi dei parametri di input vengono mappati sui corrispondenti valori dei parametri attuali contenuti nella lista dei parametri di inizializzazione.
   - Le variabili locali vengono inizializzate con il valore *null*.
   - La variabile **this** viene mappata nell'attuale oggetto che si sta inizializzando. Quest'ultimo passo può risultare banale, pensando che il riferimento all'oggetto resti immutato per tutto il processo di inizializzazione, ma non è così, perchè si presume che l'oggetto muti dopo ogni esecuzione di un ini-modulo. In haskell non sono presenti i classici puntatori presenti nei linguaggi ad oggetti. L'operazione di mappare un oggetto si traduce nel dare un nominiativo ad un valore ricavato tramite una funzione. Il nominativo sarà poi utilizzabile in altre funzioni. Durante il processo di inizializzazione viene aggiornato solo l'oggetto presente nello heap, è per questo motivo che risulta necessario, ad ogni cambio di contesto, ricavarsi l'oggetto modificato.

3. A questo punto, l'insieme di istruzioni che precede l'istruzione *super[...]* viene eseguito.

4. I valori dei parametri di output $exp_1, ..., exp_n$, definiti all'interno dell'istruzione *super[...]*, vengono valutati.

5. I nomi dei parametri di output saranno mappati sui valori valutati nello step precedente e saranno aggiunti all'insieme dei parametri di inizializzazione.

6. I restanti ini-moduli della sequenza di inizializzazione verranno ora valutati. Per trovare il prossimo modulo, candidato all'attivazione e all'esecuzione, si userà il nuovo insieme di parametri di inizializzazione andatosi a creare con lo svolgimento dei punti 1 e 5.

7. Infine, l'insieme di istruzioni che segue l'istruzione *super[...]* viene valutato.

```
1  -- Ini-mods
2  evalMods :: [IniModule] -> Eval ()
3
4  evalMods [] = do  -- (Map.size pv) == 0, case END
5    pv <- fmap configPmVls config
6    case (Map.size pv) == 0 of
7      True -> return () -- enforced by typeCheck, but
                            is right check the case
8      False -> error $ "Evaluation error: not all initialization
                 parameters were consumed \n\n Param Values: "
                 ++ (show pv)
9
10 evalMods (m : ms) = do
11   c <- config
12   h <- pure $ configHeap c
13   pv <- pure $ configPmVls c
14   case isSubList (moduleInParams m) pv of
15     True  -> do
16       parsOld <- pure $ configPmVls c -- case EXECUTION
17       pars' <- pure $ deduce (moduleInParams m) parsOld
                  -- first step
18       actuals <- evalInParams (moduleInParams m) (moduleMixin m)
19       addr <- pure $ fst $ Map.findMax h -- obj this mapped
20       objt <- pure $ ObjMixin addr
21       envOld <- pure $ configEnv c
22       ctxOld <- pure $ configCtx c
23       env <- pure $ initInput (moduleInParams m) actuals
               (initEnv m) -- second step
24       ctx <- pure $ Module objt m
```

```
25      put $ c { configEnv = env, configCtx = ctx }
26      case moduleBodyA m of -- third step
27          Just i -> do evalInstr i
28                       return ()
29          Nothing -> return ()
30      h' <- fmap configHeap config
31      c <- config
32      put $ c {configHeap = h', configEnv = env, configCtx = ctx}
33      expects <- evalOutParams $ moduleSuper m -- fourth step
34      pars'' <- pure $ computeOut (moduleSuper m) expects pars'
                -- fifth step
35      c <- config
36      put $ c { configPmVls = pars'' }
37      evalMods ms -- sixth step
38      c <- config
39      put $ c { configEnv = env, configCtx = ctx }
40      case moduleBodyB m of -- seventh step
41          Just i -> do evalInstr i
42                       return ()
43          Nothing -> return ()
44      return ()
45    False -> evalMods ms -- case SKIP
46
47  where
48    computeOut :: ParamAssignment -> [Value] -> ParamValues
                  -> ParamValues
49    computeOut [] [] p = p
50    computeOut (((mix, field), e) : xs) (v : vs) p =
51      case Map.member field p of
52        True  -> computeOut xs vs
                   (Map.update (const $ Just v) (field) p)
53        False -> computeOut xs vs (Map.insert (field) v p)
54
55    evalOutParams :: ParamAssignment -> Eval [Value]
56    evalOutParams [] = do return []
57    evalOutParams (((mix, field), e) : ps) = do
58      v <- evalExpr e
59      vs <- evalOutParams ps
60      return (v:vs)
```

```
61
62    initInput :: [Identifier] -> [Value] -> Environment
                -> Environment
63    initInput input vs (Left env) =
64      let actuals = zip (map (idName) input) vs in
65        Left $ Map.union (Map.fromList actuals) env
66
67    evalInParams :: [Identifier] -> String -> Eval [Value]
68    evalInParams [] _ = do return []
69    evalInParams (p : ps) mix = do
70      c <- config
71      pv <- pure $ configPmVls c
72      v <- pure $ Map.lookup ((idName p)) pv
73      case v of
74        Just v -> do
75          vs <- evalInParams ps mix
76          return (v : vs)
77        Nothing -> error $ "Impossible to find input parameter "
                    ++ ((idName p)) ++ " in current Param Values"
                    ++ "\n\nParamValues: " ++ (show pv)
78
79    initEnv :: IniModule -> Environment
80    initEnv mod =
81      let localIds = moduleLocals mod in
82        Left $ Map.fromList $ map (\x -> (idName x, ObjNull))
          localIds
83
84    deduce:: [Identifier] -> ParamValues -> ParamValues
85    deduce [] p = p
86    deduce (i : iis) p =
87      case Map.member (idName i) p of
88        True -> deduce (iis) (Map.delete (idName i) p)
89        False -> error $ "Function 'deduce' fatal error: " ++
                  (idName i) ++ " not in current Param Values"
90
91    isSubList :: [Identifier] -> ParamValues -> Bool
92    isSubList [] _ = True
93    isSubList (j : js) pv =
94      case Map.lookup (idName j) pv of
```

```
95          Just a -> isSubList js pv
96          Nothing -> False
```

5 Sviluppi futuri

Accenniamo ora ad alcune tra le più importanti lacune del linguaggio Magda e del nostro interprete.

Nell'attuale versione di Magda tutti i metodi sono visibili da ogni punto del programma. In una futura versione sarebbe giusto poter limitare la visibilità delle componenti di un mixin (campi, metodi e ini-moduli). In particolare sarebbe utile poter distinguere tra componenti private e componenti pubbliche.

Attualmente l'interprete non segnala a dovere gli errori lanciati a tempo di esecuzione. Infatti, l'interprete esegue il programma equivalente in codice Haskell, gli errori, se occorrono, vengono riportati rispetto al codice Haskell e non direttamente rispetto al codice Magda. Questo problema costringe un potenziale utilizzatore di Magda a conoscere sia il linguaggio funzionale Haskell, sia la struttura interna dell'interprete, senza la quale sarebbe difficile capire in quale step del processo d'interpretazione si è verificato l'errore. Un utente che si vede notificare un errore della forma:

```
1 magda: Pattern match failure in do expression at Eval.hs:134:3-10
```

è in grado di dedurre solamente che il problema si è verificato durante lo step di valutazione, dopodiché, per avere più informazioni, dovrebbe andare a controllare quale operazione viene valutata alla riga 134 del file Eval.hs.
Lo stesso tipo di problema, in forma più limitata, è presente anche nel precedente preprocessore Magda-Java sviluppato da Jarosław Kuśmierek e perfezionato da Mauro Mulatero. Il preprocessore esegue inizialmente un'analisi lessicale sul codice Magda per produrre un albero semantico che descrive l'intero programma. Sull'albero ottenuto dallo step precedente viene eseguito un controllo sui tipi. Successivamente, l'albero semantico, che descrive il codice Magda, viene tradotto in codice Java. La corrispondenza non è uno ad uno, in quanto, in Java, non posse-

diamo alcuni elementi che caratterizzano il linguaggio Magda. L'ostacolo è stato superato rappresentando tutti gli elementi del linguaggio Magda come oggetti Java. Infine, il codice Java, prodotto nello step precedente, viene compilato. Si noti che il controllo di tipo viene svolto su un albero sintattico che descrive il codice Magda quindi possibili errori trovati in questa fase vengono riportati rispetto al codice Magda. Una volta che l'albero viene tradotto, invece il codice Magda è perso e gli errori, in seguito, verranno lanciati rispetto al codice Java. I controlli preventivi, fatti dal preprocessore, aiutano a correggere gli errori e a non portarli fino alle fasi di compilazione e esecuzione del codice Java. Tuttavia non esiste una prova formale completa che il preprocessore produca codice Java completamente corretto in tutti i casi. Come abbiamo detto, invece il nostro interprete esegue tutti i controlli sul codice Haskell: potrebbe essere interessante sviluppare un processo di type checking direttamente sul codice Magda.

Magda attualmente è da considerarsi un linguaggio puramente accademico, il suo utilizzo, in ambiti esterni alla didattica, è sabotato da piccoli dettagli che un programmatore cerca sempre nei linguaggi da adottare. Il supporto, che un editor di testo, attraverso plug-in scaricabili, offre alla stesura di codice, è un fattore che viene spesso ricercato. Nel 2013 Marco Naddeo ha sviluppato un plug-in per Eclipse, scritto in Xtext, per supportare il programmatore Magda. Il plug-in prodotto integra tutte le comodità necessarie per la stesura di programmi di grandi dimensioni: il completamento automatico, l'hyperlinking, la colorazione della sintassi, ... Offrendo, inoltre, la possibilità di compilare ed eseguire il codice Magda attravero il preprocessore Magda-Java.
Paragonato al lavoro svolto da Marco Naddeo e Jarosław Kuśmierek l'interprete Magda-Haskell è ancora in fase embrionale, integrarne la struttura in un plug-in, simile a quello già sviluppato, potrebbe renderne l'utilizzo più facile e più appetibile.

Ci si potrebbe chiedere perchè svolgere lo stesso lavoro sull'intrerprete Magda-Haskell quando esiste già un compilatore utilizzabile più completo. La descrizione delle componenti di un linguaggio richiede numerose definizioni ricorsive, Haskell consente interpretazioni molto concise di tutti gli elementi del linguaggio Magda. Inoltre, fornisce la libreria *Text.ParserCombinators.Parsec* rendendo il lavoro di

stesura di un lexer e di un parser incredibilmente semplice, veloce da sviluppare ed elegante. Queste caretteristiche permetterebbero di rendere la sperimentazione di nuove carateristiche di Magda più agevole. Scrivere un compilatore, o un interprete nel nostro caso, è essenzialmente la stesura di un algoritmo dove il codice Magda è il nostro input e il codice Haskell è il nostro output, non esiste miglior modo per rappresentare un algoritmo, se non attraverso il paradigma di programmazione funzionale.

Riferimenti bibliografici

[BKM12] Viviana Bono, Jarek Kusmierek, and Mauro Mulatero. Magda: A new language for modularity. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 560–588. Springer, 2012.

[Fum19] M. Fumo. Haskell interpreter for Magda. *Tesi di Laurea in Informatica. Università degli Studi di Torino*, 2019.

[Kah87] G. Kahn. Natural semantics. in proc. In *STACS'87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[Kus10] J.D. Kusmierek. A mixin based object-oriented calculus: True modularity in object-oriented programming. *PhD thesis, University of Warsaw*, 2010.

[Lei01] Daan Leijen. Parsec, a fast combinator parser. *University of Utrecht Dept. of Computer Science*, 2001.

[Nad13] M. Naddeo. Sviluppo di un integrated development environment per il linguaggio Magda. *Tesi di Laurea Specialistica in Sistemi per il Trattamento dell'Informazione. Univeristà degli Studi di Torino*, 2013.

[rep] Haskell interpreter for Magda. `https://gitlab.com/magda-lang/hi-magda/-/tree/mo-extensions`.

[Tal20] G. Tallone. Un processore Magda-Haskell con typechecking. *Tesi di Laurea in Informatica. Università degli Studi di Torino*, 2020.