

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

Candidato:
Michael Orrù

Relatrice:
Viviana Bono

16 Luglio 2021



Cos'è Magda?

- Un linguaggio orientato ai [mixin](#).
 - Mixin: alternativa all'ereditarietà multipla → ereditarietà multipla *linearizzata*.
 - Mixin = classe incompleta: un mixin può essere *applicato* a una o più classi che «lo completano».
 - Si possono ottenere nuovi mixin *componendoli* tra loro.
 - In Magda, invece, i mixin fanno anche le veci delle classi (ovvero gli oggetti sono creati a partire dai mixin).
 - In Magda, i mixin sono organizzati in gerarchie:
ereditarietà + composizione.
- Definisce campi e metodi in modo igienico.
- Modularizza i costruttori attraverso gli ini-moduli.

Cosa sono i mixin di Magda?

- OPERAZIONE DI COMPOSIZIONE.
- ~~OPERAZIONE DI APPLICAZIONE.~~
- EREDITARIETÀ.
- ISTANZIAZIONE.

Sintassi dichiarazione di mixin

mixin *MixinName* **of** *MixinExpression*

field:MixinExpression; ...

MethodDeclaration; ...

IniModuleDeclaration; ...

end

Esempio operazione di composizione:

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

```
  new String metA()  
  begin  
    return "nothing done";  
  end  
end
```

```
mixin B of Object =  
  fieldB: String;
```

```
  new String metB()  
  begin  
    return "nothing done";  
  end  
end
```

```
//Main instructions  
new A,B[]; // operazione di composizione
```

Cos'è Magda?

- Un linguaggio orientato ai mixin.
 - Mixin: alternativa all'ereditarietà multipla → ereditarietà multipla *linearizzata*.
 - Mixin = classe incompleta: un mixin può essere *applicato* a una o più classi che «lo completano».
 - Si possono ottenere nuovi mixin *componendoli* tra loro.
 - In Magda, invece, i mixin fanno anche le veci delle classi (ovvero gli oggetti sono creati a partire dai mixin).
 - In Magda, i mixin sono organizzati in gerarchie:
ereditarietà + composizione.
- Definisce campi e metodi in modo [igienico](#).
- Modularizza i costruttori attraverso gli ini-moduli.

Igienicità degli identificatori

- Al nome degli identificatori viene anticipato il nome del mixin in cui sono contenuti.
- Impedisce possibili name clash accidentali, tramite modificatori new e override.

Esempio di overriding accidentale:

	Java	Magda
classe/mixin A	<pre>public int met(par){ // do A job; }</pre>	<pre>new Integer met(par){ // do A job; }</pre>
Classe/mixin B extends A	<pre>public int met(par){ // do B job; }</pre>	<pre>new Integer met(par){ // do B job; }</pre>
Main instructions	<pre>(new A()).met(par); (new B()).met(par);</pre>	<pre>(new A[]).A.met(par); (new A,B[]).A.met(par);</pre>

Cos'è Magda?

- Un linguaggio orientato ai mixin.
 - Mixin: alternativa all'ereditarietà multipla → ereditarietà multipla *linearizzata*.
 - Mixin = classe incompleta: un mixin può essere *applicato* a una o più classi che «lo completano».
 - Si possono ottenere nuovi mixin *componendoli* tra loro.
 - In Magda, invece, i mixin fanno anche le veci delle classi (ovvero gli oggetti sono creati a partire dai mixin).
 - In Magda, i mixin sono organizzati in gerarchie:
ereditarietà + composizione.
- Definisce campi e metodi in modo igienico.
- Modularizza i costruttori attraverso gli [ini-moduli](#).

Ini-moduli Vs costruttori

	Java (costruttori)	Magda (ini-moduli)
Point fields: - rPolarCoord - thPolarCoord - xCartCoord - yCartCoord	public Point(dist, ang) ... public Point(x, y) ...	optional Point(dist, ang) ... optional Point(x, y) ...
ColorPoint extends Point fields: - rRGB - gRGB - bRGB - hexDec	public ColorPoint(dist, ang, r, g, b) {super(dist, ang);} public ColorPoint(dist, ang, hex) {super(dist, ang);} public ColorPoint(x, y, r, g, b) {super(x, y);} public ColorPoint(x, y, hex) {super(x, y);}	optional ColorPoint(r, g, b)... optional ColorPoint(hex) ...

Magda nel tempo

2010 - Jarek Kuśmierek, tesi di dottorato

A Mixin Based Object-Oriented Calculus: True Modularity in Object-Oriented Programming.
Pre-processore Magda-Java.

2013 - Marco Naddeo, sviluppo di un IDE per il linguaggio Magda.

2019 - Mattia Fumo, prima versione dell'interprete Magda-Haskell.

2020 - Gioele Tallone, il typechecking viene aggiunto all'interprete.

2021 - Michael Orrù, all'interprete viene aggiunto il supporto
all'inizializzazione di nuovi oggetti.

Interprete Haskell: esempi pre/post-modifiche

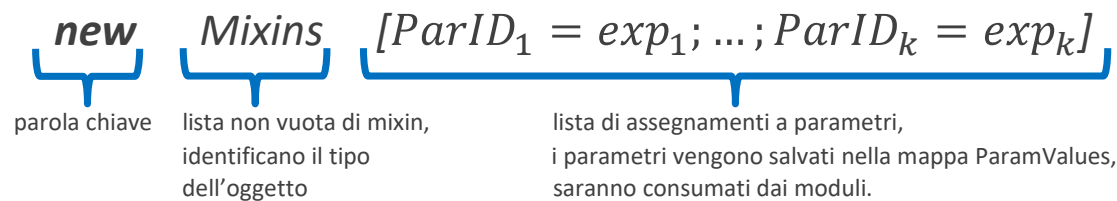
Pre-modifica:

```
mixin A of Object =  
  fieldA: String;  
  
  new Object setField(field: String)  
  begin  
    this.A.fieldA := field;  
    return null;  
  end  
end  
  
(new A[]).A.setField("parametro");
```

Post-modifica:

```
mixin A of Object =  
  fieldA: String;  
  
  required A (field: String) initializes ()  
  begin  
    this.A.fieldA := field;  
    super[];  
  end  
end  
  
new A[A.field := "parametro"];
```

Istruzione di creazione



Creiamo nuovi oggetti componendo mixin e li inizializziamo partendo dai parametri di inizializzazione $ParID_i = exp_i$.

I nomi $ParID_i$ sono importanti perché vanno a selezionare i moduli da eseguire.

Processo di inizializzazione – passi:

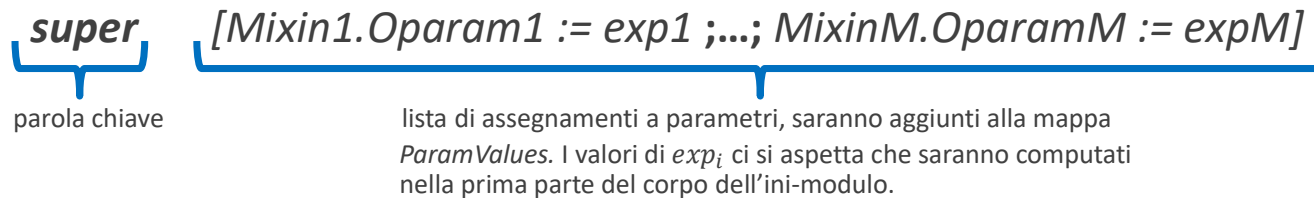
- Un oggetto vuoto, con tutti i campi settati a *null* viene creato.
- Un indirizzo viene riservato per eseguire lo store dell'oggetto.
- Il processo di creazione termina e inizia quello di inizializzazione.

Moduli di inizializzazione

Sintassi:

```
( required / optional ) Mixin (lparam1: Type1 ; ...; lparamN: TypeN)  
    initializes (Mixin1.Oparam1 ; ...; MixinM.OparamM)  
    VarName:Type ; ...; VarName:Type ;  
begin  
    Instr ; ...  
    super[Mixin1.Oparam1 := exp1 ;...; MixinM.OparamM := expM];  
    Instr ; ...  
end
```

Istruzione: *super*[...]



Il lavoro svolto riguardante l'istruzione *super*[...] rientrava nell'ambito del type checking:

- I nomi dei parametri *Oparam1*, ..., *OparamM*, usati all'interno delle parentesi quadre, devono combaciare con i nomi dei parametri di output dell'ini-modulo in cui l'istruzione si trova.
- i parametri attuali *exp1*, ..., *expK* hanno lo stesso tipo dei parametri di output dell'ini-modulo in cui l'istruzione *super* si trova.

Come vengono attivati gli ini-moduli

- Un modulo di inizializzazione viene attivato da una lista di parametri che memorizziamo in una struttura detta *ParamValues*.
- *ParamValues* viene inizializzata con i parametri presenti nell'istruzione `new[...]`.
Esempio: `new Point [Point.x = 1, Point.y = 4];`
- Un ini-modulo attivato viene eseguito, andando a modificare i campi dell'oggetto, lo stato del programma e la mappa *ParamValues* nel seguente modo:
$$ParamValues = ParamValues - mod.inputPars + mod.outputPars$$
- Un ini-modulo attivato eseguirà il suo corpo fino all'istruzione `super[...]` successivamente il processo di inizializzazione cerca un altro ini-modulo da attivare. Il restante corpo dell' ini-modulo viene eseguito dopo tutti gli altri ini-moduli attivati.
- Il processo di ricerca di un potenziale ini-modulo da attivare non si ferma quando la mappa *ParamValues* è vuota: un ini-modulo che ha l'insieme vuoto come insieme di parametri di input viene sempre attivato dal processo di inizializzazione.

Interprete Haskell: concetti aggiunti e rivisitati

- Ini-moduli: l'interprete è in grado di parsificare, eseguire un controllo di tipo e valutare questi costrutti.
- Istruzione di creazione (`new Mixins [...]`): all'interno delle parentesi quadre è ora possibile indicare dei parametri di inizializzazione che corrispondono a dei parametri degli ini-moduli, in modo da attivarli.
Esempio: `new Point [Point.x = 1, Point.y = 4];`
- Istruzione `super[...]`:
 1. Sempre presente all'interno di un ini-modulo.
 2. Permette di specificare ulteriori parametri di inizializzazione.
 3. Usata per risalire la gerarchia di mixin.
- Configurazione di un programma Magda:
 1. Il concetto «contesto d'esecuzione» è stato esteso.
 2. La mappa `ParamValues` è stata aggiunta.

Modello della memoria

Contesto d'esecuzione:

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

ParamValues:

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

```
data Value = ObjNull  
  | ObjBool Bool  
  | ObjInt Integer  
  | ObjString String  
  | ObjThis  
  | ObjMixin Int  
deriving (Show, Eq)
```

- Context viene definito come un nuovo data type, il contesto mantiene sempre il riferimento al metodo o ini-modulo in esecuzione, utilizziamo il valore Top per indicare che ci troviamo nel main.

- ParamValues viene definito come un nuovo tipo, un'etichetta che rappresenta una mappa che associa identificatori ad indirizzi.

Moduli di inizializzazione - Haskell

```
data IniModuleScope = IniRequired | IniOptional
  deriving Eq
```

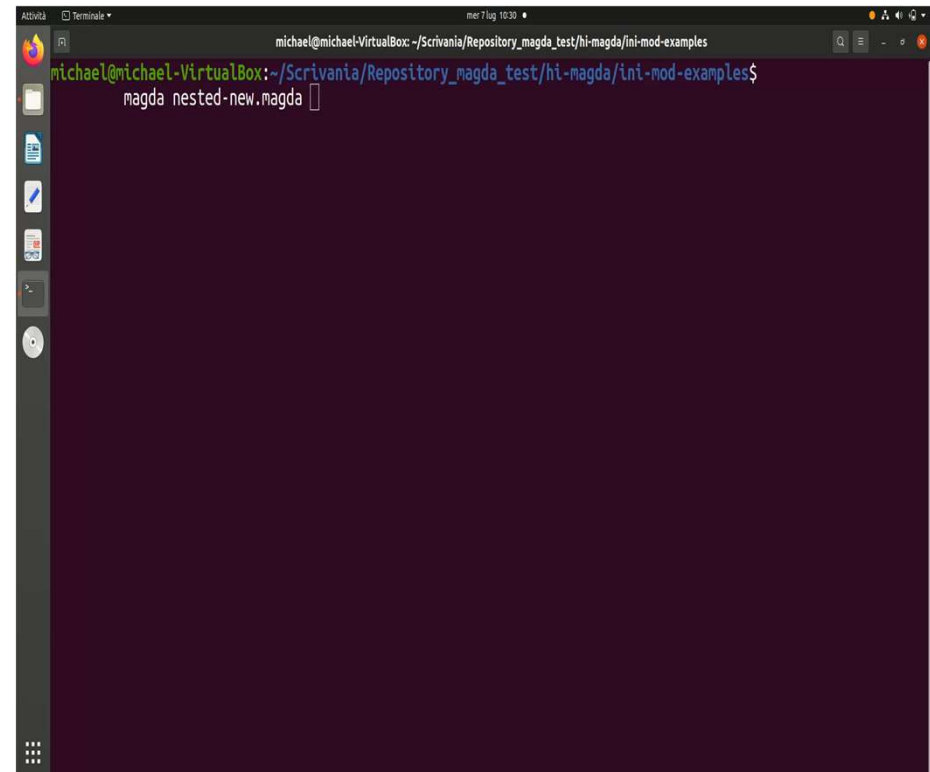
```
type ParamAssignment = [(String, String), Expression]
```

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


Processo di inizializzazione

Esempio 1:

```
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"]];
```



The screenshot shows a terminal window titled "Terminal" with the user "michael" at host "michael-VirtualBox". The current directory is `~/Scrivania/Repository_magda_test/hi-magda/ini-mod-examples`. The prompt is `michael@michael-VirtualBox:~/Scrivania/Repository_magda_test/hi-magda/ini-mod-examples$`. The user has entered the command `magda nested-new.magda`, and the cursor is at the end of the line.

Processo di inizializzazione

Esempio 2:

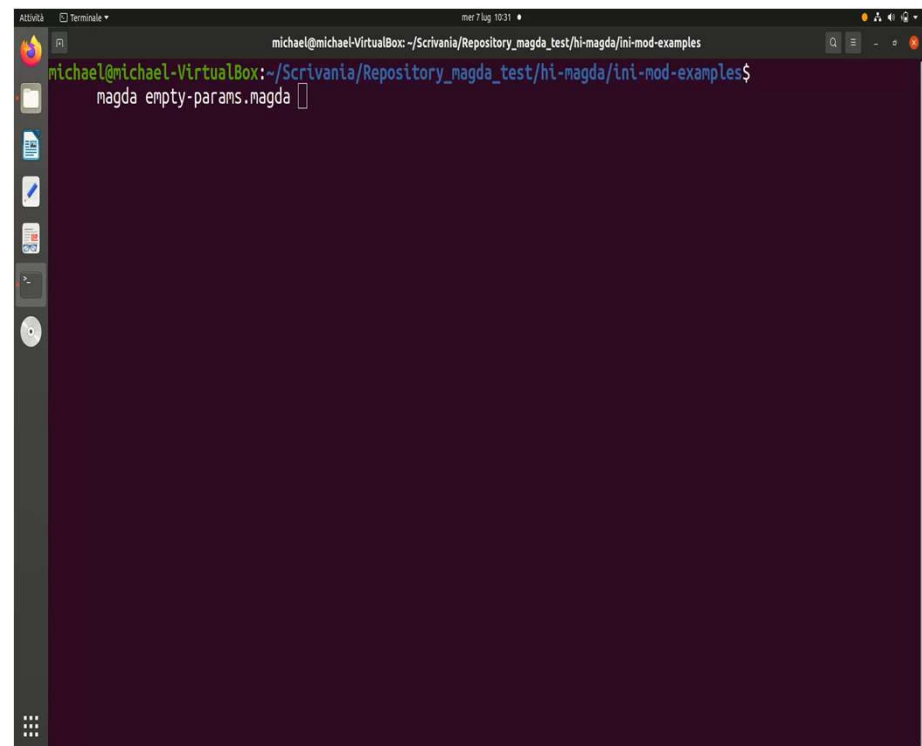
mixin Test of Object =

```
required Test() initializes ()  
begin  
  "Instruction 2".String.print();  
  super[];  
  "Instruction 3".String.print();  
end
```

```
optional Test() initializes ()  
begin  
  "Instruction 1".String.print();  
  super[];  
  "Instruction 4".String.print();  
end
```

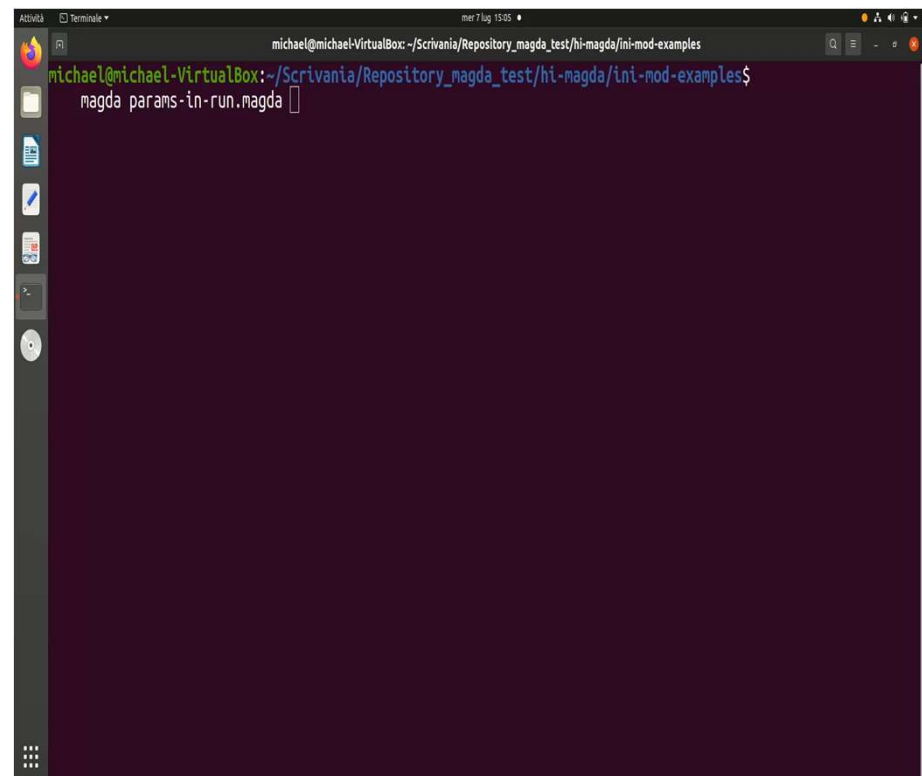
end

(new Test[]);



Istruzione *super[...]* - Funzionamento

```
mixin A of Object =  
  fieldRequired: String; fieldOptional: Integer;  
  
  required A() initializes ()  
    local:String;  
  begin  
    local := "hello";  
    super[];  
    "module 1 executed".String.print();  
  end  
  
  required A(fieldRequired : String) initializes ()  
  begin  
    this.A.fieldRequired := fieldRequired;  
    super[];  
    "module 2 executed".String.print();  
  end  
  
  optional A(fieldOptional : Integer) initializes (A.fieldRequired)  
  begin  
    this.A.fieldOptional := fieldOptional;  
    super[A.fieldRequired := "hello"];  
    "module 3 executed".String.print();  
  end  
end  
  
new A [A.fieldRequired := "hello"];  
"\n".String.print();  
new A [A.fieldOptional := 100000];
```



```
michael@michael-VirtualBox: ~/Scrivania/Repository_magda_test/hi-magda/init-mod-examples$  
magda params-in-run.magda
```

Sviluppi futuri e conclusioni

- Limitare la visibilità delle componenti di un mixin. Sarebbe utile poter distinguere tra componenti private e componenti pubbliche.
- Integrare l'interprete in un plug-in simile a quello già sviluppato da Marco Naddeo lo renderebbe più accessibile.
- Perché scrivere un interprete Haskell quando esiste un pre-processore Magda-Java?