Jolie

SERVICE ORIENTED PROGRAMMING

Panoramica

Jolie è il primo linguaggio *general-purpose* per la *programmazione a (micro) servizi*

Servizi Semplici: scritti da zero, wrapper a servizi esistenti terzi ed eterogenei (e.g. Web Services, PostgreSQL, etc)

Orchestratori: realizzano servizi orchestrando (combinando, coordinando, …) servizi esistenti (anche dinamicamente)

Composizione di sistemi: Composizione/integrazione di SOA (service ortiented architectures) esistenti (e non necessariamente scritti in Jolie) grazie ad primitive come *Aggregazione e Redirezione*

Web Applications: HTTP/AJAX/REST etc. sono solo canali/protocolli di accesso ad un *servizio* esposto dal sistema

Panoramica

Solide basi teoriche

- Incarnazione dell'algebra di processi SOCK (2005) (service-oriented comunication kernel)
- Sistema di tipi comportamentali (session types)
- …

Utilizzato nel «mondo reale»

- ItalianaSoftware s.r.l. (system integration)
- EOS (web)
- GPA Group (business automation)

La vostra nuova homepage: jolie-lang.org

Hello world

```
include "console.iol"

main{
    println@Console( "Hello, world!" )()
}
```

Console è un servizio:

- Inviamo un messaggio al servizio “Console” sul canale “println”
  - **channel@service(message)**
- Il messaggio è la stringa “Hello, world!”
- () solicit response i.e. ack
  - **channel@service(message)(response)**
  - **double@math(5)(x)**

Esecuzione:

```
$> jolie hello.ol
Hello, world!
```


Behaviour & Deployement

Ogni programma Jolie definisce un servizio

È composto da due parti:

- Behaviour:
  - Implementazione delle funzionalità del servizio
  - La comunicazione è gestita in modo astratto
  - E.g. «chiedi al servizio di terminale di stampare questa stringa»
- Deployement:
  - Informazioni per trovare e invocare servizi
  - Completa il behaviour
  - E.g. chi è il terminale e come va contattato
  - Definisce la struttura di una SOA

Esempio: behaviour

Server

```
main{
    twice( number )( response ) {
        response = number * 2
    }
}
```

Client

```
main{
    twice@TwiceService(5)(response);
    println@Console(response)
}
```

Esempio: deployement

Tutte le primitive di comunicazione dipendono dal concetto di Porta

Una porta definisce:

1. Locazione (e.g. URL)
2. Protocollo (e.g. SOAP)
3. Interfaccia

di un servizio e può essere di output o di input (client-server)

1 e 2 forniscono un *binding concreto* tra un programma Jolie e un servizio

3 definisce il contratto (o tipo nel sistema di tipi di Jolie) del servizio (cf. IDX)

Esempio: deployement

Interface:

```
interface TwiceInterface {
    RequestResponse: twice(int)(int)
}
```

Server port:

```
inputPort TwiceService {
    Location: "socket://localhost:8000"
    Protocol: sodep
    Interfaces: TwiceInterface
}
```

Client port:

```
outputPort TwiceService {
    Location: "socket://localhost:8000"
    Protocol: sodep
    Interfaces: TwiceInterface
}
```


Tipi di dato semplice

```
a = 5;
a = "Hello"
```

Jolie è un sistema a tipizzazione dinamica (e non serve dichiarare le variabili)

Ma i tipi esistono (e.g. nelle interfacce)

Tipi di base:

- bool, int, long, double, string, void, any, raw (bitstring, ma non come in Erlang ☹)

Cast:

- n = double("1.3")

Test:

- expr instanceof type

Una variabile è definita solo a seguito di un assegnamento

- is_defined(var)
- undef(var)

Array

Ogni variabile è un array
- a = 5;
  println@Console(a[0])() // -> 5

Non sono omogenei
- a[0] = 5;
  a[1] = "cinque"

Indici arbitrari
- a["zero"] = a[0]

Dimensione
- #a

Composizione

Composizione sequenziale:
- print@Console("Hello, ")();
  println@Console("world!")()
- L'operatore «;» sequenzia comandi, non li termina

Composizione parallela:
- print@Console("Hello, ")() | println@Console("world!")()

I blocchi sono racchiusi in { }

Annidamenti arbitrario:
- {A ; B} | {C ; D}

L'operatore | ha priorità
- A ; B | C ; D

Input condizionale

[InputStatement1] { CodeBranch1 }
[InputStatement2] { CodeBranch2 }
…
[InputStatementN] { CodeBranchN }

Il primo input completato con successo disabilita gli altri

Viene eseguito il branch corrispondente

E.g.
```
[ buy( stock )( response ) {
    buy@Exchange( stock )( response )
} ] { println@Console( "Buy order forwarded" )()}

[ sell( stock )( response ) {
    sell@Exchange( stock )( response )
}] { println@Console( "Sell order forwarded" )()}
```

If, for, while …

```
if ( condition ) {
    ...
} [else {
    ...
}]


while( condition ) {
    ...
}


for ( init-code-block, condition, post-cycle-code-block ) {
    ...
}
```


Procedure

define consente di associare un nome ad un blocco di codice:

```
define procedureName{
      ...
}
```

Non possiedono scoping locale:

-
```
define sumProcedure{
    sum = x + y
}
main{
    x = 1;    y = 2;
    sumProcedure
}
```


Strutture dati

Strutture dati

- tree-like come XML, JSON
- sono record non tipati (niente metodi, campi privati etc.)

E.g.:

- Jolie:
  -
```
animals.pet[0].name = "cat";
animals.pet[1].name = "dog";
animals.wild[0].name = "tiger";
animals.wild[1].name = "lion"
```
- JSON:
  -
```
animals : {
      "pet" : [
          { "name" : "cat" },
          { "name" : "dog" }
      ],
      "wild" : [
          { "name":"tiger" },
          { "name":"lion" }
      ]
  }
```

Strutture dati

Non necessitano di dichiarazione

Ogni variabile può essere estesa arbitrariamente

Strutture chiave-valore annidate

E.g.:

- `println@Console( animals.( “pe” + "t" )[ 0 ].name )()`
- ```
  foreach ( kind : animals ){
    for ( i = 0, i < #animals.( kind ), i++ ){
      println@Console(animals.(kind)[i].name)()
    }}
  ```

La keyword “with” (VB?) consente di risparmiare un po’ di battute

- ```
  with ( animals ){
    .pet[ 0 ].name = "cat";
    .pet[ 1 ].name = “dog";
  }
  ```


Strutture dati

- “<<” deep copy
- “=“ shallow copy
- “->” alias

```
with ( a.b.c ){
        .d[ 0 ] = "zero";
        .d[ 1 ] = "one";
        .d[ 2 ] = "two";
        .d[ 3 ] = "three"
};
currentElement[ 0 ] -> a.b.c.d[ i ];
```


Costanti

Jolie supporta la definizione di costanti (anche globali)

```
constants {
   const1 = val1,
   const2 = val2,
   ...
   contsN = valN
}
```

Molto utili per configurare port (cf. Macro C e .h)

Porte

Una porta è una tripla (locazione, protocollo, interfacce)

Locazione: *medium[:parametri]*
- socket (TCP/IP)
- btl2cap (bluetooth)
- rmi (Java RMI)
- localsocket (Unix Local Socket)

Protocollo
- http/https
- soap
- sodep
- Xml/json rpc

(Esercizio: interfacciare Jolie con altri esempi visti e.g. via SOAP, xmlrpc)

Interfacce

Un'interfaccia è una collezione di (tipi e) operazioni asincrone o meno

```
interface identifier {
    OneWay:
        ow_name1( t1 ),
        ...,
        ow_nameN( tN )
    RequestResponse:
        rr_name1( t11 )( t12 ),
        …,
        rr_nameN( tN1 )( tN2 )
}
```


Interfacce (2)

I tipi di dato sono tree-like

```
type T: tipo_base {
    .subnode_1: tipoBase o tipoUtente
    .subnode_2[min,max]: tipo
    .subnode_n?: {?}
}
```

E.g.:
- type myType: string {
  .x[1, *]: mySubType
  .y[1, 3]: void {
  .value*: double
  .comment: string
  }
  .z?: void { ? }
  }

(Esempio…)

Binding dinamico

Località e protocollo di una porta di **output** sono assegnabili dinamicamente

Le porte sono definite parzialmente, e.g:

```
outputPort P {
        Interfaces: PrinterInterface
}
```

E trattate come variabili e.g.:

```
main{
    P.location = "socket://p:80/";
    P.protocol = "sodep"
}
```

Il deployement è definibile/adattabile dinamicamente

Esempio: Service directory

```
interface RIf { RequestResponse: getBinding( string )( Binding ) }
```

ServiceDirectory server:

```
main{
    getBinding( name )( b ){
        if ( name == "LaserPrinter" ){
            b.location = "socket://p1.com:80/";
            b.protocol = "xmlrpc"
        } else if ( name == "InkJetPrinter" ) {
            b.location = "socket://p2.it:80/";
            b.protocol = "soap"
        }}}
```

Client:

```
outputPort Printer {
    Interfaces: PrinterInterface
}

main{
    getBinding@ServiceDirectory( "LaserPrinter" )( Printer );
    printText@Printer( "My text" )
}
```


Processi

Jolie supporta il riutilizzo dei behaviour mediante la primitiva di deployement *execution modality*

- execution { **single** } default, al temine l'istanza è rimossa
- execution { **sequential** } riavvia il behaviour al termine (nuova istanza)
- execution { **concurrent** } avvia un istanza per richiesta (cf socket accept)

(cf. behaviours in Erlang/OTP)

La procedura speciale init{} è eseguita (una sola volta) prima del main

Istanze separate (niente variabili condivise)

- Variabili globali distinte dal prefisso *global.*
- Sincronizzazione mediante blocchi *synchronized*(identifier){…}

(Esempio)

Correlatori

Le sessioni possono contenere più istanze dello stesso behaviour

Il routing all'interno delle sessioni si basa su correlatori

i.e. identificatori univoci per ogni istanza

Jolie generalizza il concetto con i **c**orrelation **set**s

È compito del behaviour dichiarare quali componenti dei messaggi fanno parte di un dato correlation set ed eventuali alias:

- **cset** { sid: OpMessage.**sid** OpMessage.MetaData.sid }

```
main {
    login( request )( response ){
        username = request.name;
        response.sid = csets.sid = new;
        // code
    }}
```


Gestione degli errori

Gestione degli errori reminiscente del *try … catch*

- Gli **scope** delimitano blocchi di codice e presentano un nome univoco
- Uno scope può installare più **fault handlers**
- Questi intercetteranno tutti i **faults** all'interno del loro scope

```
scope( scope_name ){
    install (
          fault_name1 => println@Console("ARGH!"),
          fault_name2 => warning@Log(fault_name2)
    );
      …
    throw( fault_name2 )
}
```

(install ha la massima priorità nel caso di composizione parallela)

Gestione degli errori

Fallimenti non catturati dal contesto di un RequestResponse sono inoltrati al «chiamante» e **vanno dichiarati**

E.g.:

```
type fault_number_type: void{
    .number: int
    .fault_error: string
}

interface Guess {
    RequestResponse: guess throws
       fault_number(fault_number_type)
}
```

Gestione degli errori

Gli errori si propagano **terminando** scope paralleli

```
scope( scope_name ){
    install (
          fault => println@Console("fault"),
    );
     …
}|throw( fault )
```

Installare handler da eseguire in caso di terminazione:

```
install ( this => println@Console("termiantion") );
```

È possibile dichiarare fallback handlers:

```
install ( default => println@Console("catch-all") );
```


Gestione degli errori

I fault handler possono essere installati dinamicamente

-
```
install(someFault => println@Console( "handler 1" )());
…
install(someFault => println@Console( "handler 2" )());
…
| throw(someFault)
```

L'installazione di un nuovo handler sovrascrive il precedente

Il gestore preesistente è raggiungibile mediante il place-holder **cH**:

-
```
install(someFault => println@Console( "handler 1" )());
…
install(someFault => {cH;
   println@Console( "handler 2" )()) };
…
| throw(someFault)
```


Fault handler e variabili

-
```
i = 1;
  while( true ){
   sleep@Time(1)();
   install(someFault => {cH;
     println@Console( "handler " + i )()) };
   i++
}
```

Problema: i è valutata al momento della gestione del fault e tutte le stampe produrranno lo stesso messaggio.

L'operatore «**^**» consente di «congelare» lo stato di una variabile al momento dell'installazione di un fault handler

Gestione degli errori

La **compensazione** consente di gestire il recovery di scope completati con successo

- Lo scope è completato con successo,
- I suoi handler sono promossi allo scope genitore
  - esplicitamente invocando comp(nome_scope)
- In caso di recovery del genitore sono eseguite le compensazioni

```
install( a_fault =>
        println@Console("a_fault" )();
        comp( child_scope ) // <-- compensazione
    );
    scope( child_scope ){ … };
    throw( a_fault )
```


Porte e controlli di tipo

I messaggi inviati e ricevuti da ogni porta devono soddisfarne il tipo

In caso contrario è generato il fallimenti **TypeMismatch**

Operazioni *OneWay*

- Il controllo è fatto da mittente e destinatario
- Se fallisce il primo controllo il messaggio non è inviato e il fallimento è sollevato (solo lato mittente)
- Se fallisce il secondo controllo il messaggio è scartato e il mittente non è notificato

Porte e controlli di tipo

I messaggi inviati e ricevuti da ogni porta devono soddisfarne il tipo

In caso contrario è generato il fallimenti **TypeMismatch**

Operazioni *RequestResponse*

- Il controllo è fatto da mittente e destinatario sia per richiesta che risposta

	Richiesta	Risposta
Mittente	• Il messaggio non è spedito • TypeMismatch	• TypeMismatch
Destinatario	• Messaggio scartato • Warning su stdout • TypeMismatch al mittente	• TypeMismatch sollevata • TypeMismatch al mittente

Composizione architetturale

Behaviour
- Composizione sequenziale «;»
- Composizione parallela «|»

Deployment
- Strutturazione del contesto d'esecuzione di un servizio
  - Sotto-servizi nello stesso execution engine
  - Wrapping e hiding
- Gestione della topologia dei servizi
  - Binding dinamico
  - Aggregazione
  - Reindirizzamento
  - Couriers

Embedding

Consente di eseguire più servizi nella stessa VM
- **Embedder**: servizio «master» (main)
- **Embedded:** (sotto)-servizio

Primitiva di deployement
- embedded {
  Language : Path [in OutputPort]
  }
- Language = Jolie, Java o JavaScript (per ora)
- *(Laboratorio: aggiungere altri linguaggi)*

Supporto a parametri

Comunicazione **in-memory** mediante medium **local**

Embedding

È possibile variare dinamicamente un embedding

E.g.:

```
outputPort CounterService{
    Interfaces: CounterInterface
}

…

eInfo.type = "Jolie";
eInfo.filepath = "CounterService.ol";
loadEmbeddedService@Runtime(Info )(CounterService.location );

…
```

Aggregazione

Aggregazione di servizi nel deployement

```
inputPort TwiceService {
   Location: medium:parameters
   Protocol: protocol
   Interfaces: i1, ..., iN
   Aggregates: outputPort1, … , outputPortM
}
```

E.g. proxy, wrapper, embedder

Sessioni courier

Composizione di servizi indipendente dal contesto di esecuzione

Le sessioni courier sfruttano l'aggregazione

L'aggregatore **estende** servizi «esterni» e.g. aggiungendo

- Autenticazione
- Autorizzazione
- Mutua esclusione
- Etc…

Verifica/processa parte di messaggi «overloaded» e poi li inoltra al servizio aggregato.

Sessioni courier

Estensori di interfacce:

```
interface extender extender_id {
    OneWay: * | OneWayDefinition
    RequestResponse: * | RequestResponseDefinition
}
```

Installazione in porte aggregate e sessione courier:

```
inputPort AggregatorPort {
    Location: ...
    Protocol: ...
    Aggregates:
        outputPort_1 with extender_id1,
        ...
        outputPort_n with extender_idn
}
courier AggregatorPort {
    interface interface_id( request )[( response )]{
        …
        forward( request )[( response )]
    }
}
```

Sessioni courier + collezioni

Una collezione è un insieme di porte di output con lo stesso tipo (e.g. repliche)

Un aggregatore può estendere collezioni

```
inputPort AggregatorPort {
    Location ...
    Protocol ...
    Aggregates:
        { outputPort_11, outputPort_12, ... } with extender_id1,
        //  ...
        { outputPort_n1, outputPort_n2, ... } with extender_idn
}
```

E instradare dinamicamente i messaggi:

```
courier AggregatorPort {
    interface interface_id( request )[( response )]{
        ...
        forward (outputPort_i(request) | … | outputPort_j(request))
    }
}
```


Esercizi

1. Jolie e servizi terzi
   1. Java RMI
   2. XMLRPC
   3. SOAP
2. Key-Value store
   1. Centralizzato
   2. Lock/Unlock
3. Esecuzione sequenziale vs concorrente
4. Callbacks in Jolie?