Design and Implementation of a Behaviorally Typed Programming System for Web Services

Dissertação de Mestrado

Filipe David Oliveira Militão (26948)

Orientador: Prof. Doutor Luís Caires
Part 1 – Introduction (~10 min)

• Motivation

• What is a Behavioral Type?

• Why do we need Behavioral Types?

• Overview (programmer's perspective)

• Contributions
Motivation

- Increasing *software* complexity
  - requires more sophisticated tools
  - faster feedback on possible errors
  - cut back errors only detectable at runtime

- **Web Services**
  - many standards (WSDL, etc)
  - dynamic combination of services
    + automatic type compatibility checks
    - behavior “assumed” compatible

→ ease *Web Services* use/composition
→ statically check concurrent compositions
What is a Behavioral Type?

Imagine a monster with a strange habit of *squashing* cats and then cook them into *pancakes* or *tea* right before going to *sleep*.
What is a Behavioral Type?

Imagine a monster with a strange habit of **squashing** cats and then cook them into **pancakes** or **tea** right before going to **sleep**.

**Monster Type**
- squash(Cat)
- makePancakes(Cat)
- makeTea(Cat)
- sleep()
Imagine a monster with a strange habit of *squashing* cats and then cook them into *pancakes* or *tea* right before going to *sleep*.

**Monster Type**
- squash(Cat)
- makePancakes(Cat)
- makeTea(Cat)
- sleep()

**Behavior**
1º squash cat
2º pancakes or tea
3º sleep
What is a Behavioral Type?

Imagine a monster with a strange habit of *squashing* cats and then cook them into *pancakes* or *tea* right before going to *sleep*.

**Monster Type**
- squash(Cat)
- makePancakes(Cat)
- makeTea(Cat)
- sleep()

**Behavior**
1º squash cat
2º pancakes or tea
3º sleep

**Behavioral Type** = **Type** + **Behavior**
Why do we need Behavioral Types?

- statically check a program's correct flow of calls (ignoring possible trapped errors)

- **benefits**: avoids less obvious errors such as opened file/sockets not being safely closed after use (could lead to possible loss of data)

- Behavioral checking includes:
  - verifying termination in the use of a behavior (correct resource discard)
  - checking branches, loops and exceptions in a flexible way
  - deciding if/when a behavioral type can be replaced by another behavior
Don't Panic Airlines wants to create a simple Web Service for its customers and requires:

- all clients must be authenticated (logged in)
- it's possible to choose a special package, although some might be sold out
- in the case of booking a simple flight there's an additional option of also booking a return flight
- it should also be possible to list all available flights
- “at most, only one purchase per log in / session”
class DPA {

    login(String username, String password) {
    }

    logout() {
    }

    specialPackage(String type) throws SoldOut {
    }

    bookDestination(String dest) {
    }

    bookReturnFlight() {
    }

    printAllAvailableFlights() {
    }
}

(initial approach to the problem)
class DPA {

    login(string username, string password) {
    ... } 
logout() {
    ... }

    specialPackage(string type) throws SoldOut {
    ... }

    bookDestination(string dest) {
    ... }

    bookReturnFlight() {
    ... }

    printAllAvailableFlights() {
    ... } can be called freely 
}

only available on specific situations
In order to restrict the use of those methods, we define a specific usage protocol to be applied to anyone using the class.

This protocol is only related to the method's name, not their return type or arguments.
login ; logout
login;
  +
  specialPackage
  +
  stop

; logout
login;
&choose(
    (bookDestination; bookReturnFlight?) +
    specialPackage[SoldOut: choose]
    +
    stop
);
logout
class DPA {

usage login ;
&choose(
+ specialPackage[SoldOut: choose ]
+ stop
); logout

login(string username, string password) { ... }
logout(){ ... }
specialPackage(string type) throws SoldOut { ... }
bookDestination(string dest){ ... }
bookReturnFlight(){ ... }
printAllAvailableFlights(){ ... }
}
requestFlight(DPA s) {
    s.login("usr", "pwd");
    //...
    s.logout();
}

\textbf{login} ; &\textbf{choose}(
  + specialPackage[\textbf{SoldOut: choose} ] + stop ) ; \textbf{logout}
requestFlight(DPA s) {

    s.login("usr", "pwd");
    s.printAllAvailableFlights();

    if (? ){
        //choice 1
    }
    else{
        //choice 2
    }
}

s.logout();

login ; &choose(
    + specialPackage[SoldOut: choose ] + stop ) ; logout
requestFlight(DPA s){
    s.login("usr","pwd");
    s.printAllAvailableFlights();
    if (?){
        s.bookDestination("Lisbon");
        if (?){ s.bookReturnFlight(); }
    }
    else{
        try{
            s.specialPackage("around the world 80");
        }catch(SoldOut out){
            //never mind then...
        }
    }
    s.logout();
}

login ; &choose(
    + specialPackage[SoldOut: choose ] + stop ) ; logout
Contributions

• Design of the programming language yak

• Design and formalization of a behavioral type system

• Implementation of a fully functional proof-of-concept prototype
Contributions

• Design of the programming language **yak**
  - simple (minimalistic)
  - Java “inspired” (similar syntax)
  - apply main features of the type system

• Design and formalization of a behavioral type system

• Implementation of a fully functional proof-of-concept prototype
Contributions

- Design of the programming language **yak**
- Design and formalization of a behavioral type system
  - behavioral termination
  - behavioral ownership
  - branching
  - loops
  - exceptions (new approach in behavioral types)
  - ...
- Implementation of a fully functional proof-of-concept prototype
Contributions

● Design of the programming language **yak**

● Design and formalization of a behavioral type system

● Implementation of a fully functional proof-of-concept prototype
  - language parser
  - interpreter
  - run-time system (WS using HTTP+XML)
  - type checker (based on DFA manipulation)
  - examples
  - available for download
Part 2 - How it works

- Protocol
- Program's Structure
- Type System
Protocol (I)

- Describes sequences of (allowed) behavioral calls
- Any protocol may include:
  - method's names
  - exceptions types
  - recursion labels

- empty behavior: `stop` (behavior of basic types)
- operators:
  - `a + b` choice
  - `a ; b` sequence
  - `a*` repetition
  - `&label(a;stop+label)` (limited) recursion
  - `a[Error: b];c` exceptions
Protocol (II)

- Can express more complex behaviors like “repeat on error”:

  &start( hello[NoReply: start];goodbye )

- + operator → “external” choice
  - The programmer may choose freely any of the given options

- exceptions → “internal” choice
  - The internal logic of the class decides to change the allowed protocol and “announces” the change as an exception

- Internally, the protocol is converted to a Deterministic Finite Automaton (DFA)
Program's Structure
Program's Structure

All static variables must be `#stop`

Note: basic values are all `stop` (`boolean#stop`, etc)
Program's Structure - Distribution

@local

interface C

class Q

class I

static X

class A

interface G

class W

@remote

interface P

class R

class L

static k

static y

static o

HTTP+XML
Program's Structure – Distribution Example

//client

interface Hello @“localhost:8180”
class RemoteHello @“localhost:8180”
class Main{
    main(){
        Hello newer = new RemoteHello();
        Lib.println( newer.say() );
    }
}

//server @localhost:8180

interface Hello{
    string say();
}
class RemoteHello{
    string say(){
        return “I'm remote”;
    }
}

REST inspired URL format:

(protocol)://(ip:port)/yak/Type/Instance#/Method

type interface: http://localhost:8180/yak/RemoteHello
instance: http://localhost:8180/yak/RemoteHello/1
method invocation: http://localhost:8180/yak/RemoteHello/1/say
Program's Structure

Zooming on a single class
Program's Structure – Class internals (I)

class Q

- **fields** (always private)
- **methods** (always public)
Program's Structure – Class internals (II)

class Q

behavioral methods

non behavioral methods (free use)

usage
Example – File interface

```java
interface File{
    usage &start(
        ( openRead; read* ) +
        ( openWrite; write* ) +
        ( openReadWrite; (read+write)* )
    ; close
    )[ openRead, openWrite, openReadWrite
        -> FileNotFoundException: stop+
        (changeFile; start) |
        read, write
        -> IOException: close ]

    changeFile(string name);

    openRead() throws FileNotFoundException;
    openWrite() throws FileNotFoundException;
    openReadWrite() throws FileNotFoundException;

    string read() throws IOException;
    write(string content) throws IOException;
    write(string content, integer offset) throws IOException;

    close();

    integer size();
    string name();

    free methods
}
```
Program's Structure – Fields permissions (I)

Non behavioral methods can be called in any context.

Therefore, to avoid inconsistencies they see all fields as constants with a stop behavior.
Behavioral methods are called in specific contexts.

To verify each method correctly uses the class' fields we do a **consistency check** so that each method uses a field's behavior correctly.
Consistency check

class C - Consistency Check

```java
class C{
    usage a;b[integer: c];c
    A#a;(b;(b+c)+(d;(d+c))) v;
}
C()
    v = new A();
}
a()
    v.a();
}
b() throw integer{
    if( ? ){
        v.d();
        throw 1337;
    }else{
        v.b();
    }
}
c()
    v.c();
}
v: A#stop
```

Consistency check diagram:
For additional flexibility, the usage protocol is only related to anyone using the class from the “outside”. Code inside the class' body is allowed to call any of the class' methods.

code “importing” recursion
```java
class C{
    C c;
    ...
    m(){
        if(...){
            c.a();
            n();
        }
        ...
    }
    ...
    n(){
        c.b();
        ...
    }
    ...
}
```
Recursion

**source code**

```java
class C{
    usage m
    T#a t;
    C(){
        t = new T();
    }
    m(){
    if( ? ){
        m();
    }else{
        t.a();
    }
    }
}
```

**pre-pass**

```java
m(){
    ?;
    t.a();
}
```

**base-pass**

```java
m(){ // t = T#a
    if( ? ){
        // t = T#a
        m();
        // t = T#stop
    }else{
        // t = T#a
        t.a();
    }
    // t = T#stop
}
```

**full-pass**

```java
m(){ // t = T#a
    if( ? ){
        // t = T#a
        m();
        // t = T#stop
    }else{
        // t = T#a
        t.a();
    }
}
```
Program's Structure

class Q

Zooming on a single method
Any local variable must fulfill its behavior before the method ends.
Program's Structure

```
A a = new A(); // a -> A#a;b;c
m(a); // a -> A#c
a.c(); // a -> A#stop
```
Program's Structure - Ownership

method

arg1 → owned A#a;b;c
arg2 → owned B#c

this.field = arg1;

arg1 → A#stop
arg2 → B#stop

return arg2;
Program's Structure - Ownership

// only 1 unique (full) owner
A a = new A(); // a → A#a;b;c
a.a(); // a → A#b;c
A#b;c b = a; // a → A#stop b → A#b;c
A#stop c = b; // b → A#b;c c → A#stop

method

this.field = arg1;

arg1 → owned A#a;b;c
arg2 → owned B#c

return arg2;
Program's Structure

- Method

  - if-else

- Method

  - while/repeat

- Method

  - exceptions (try-catch + throw)
Exceptions (try-catch + throw)

m()
A# a;(b;b)+c;d v = new A();

try{
  v.a();
  if( ? ){
    v.b();
  }
  v.c(); v: A#d
}

INTERSECTION
catch(integer e){
  v.b(); v: A#d
};

v.d();

v: A#stop

a;((b;b)+c);d → 

a

b

b

c

d

→ c

→ d

→
Subtyping

Replacing a behavioral type with another, while still obeying behavioral expectations

call A

call B

Done.
Subtyping

Replacing a behavioral type with another, while still obeying behavioral expectations.

call A

call B

Done.
interface Order{
    usage review*;buy?
    /* ... */
}

class TravelOrder{
    usage (packageAlaska+packageArtic)[SoldOut: stop]+
        (flight;hotel); (review*; buy?)
    /* ... */
}

class HotelOrder{
    usage bookGroup+bookPenthouse+bookRoom* ;
        breakfast? ; dinner? ; (review*; buy?)
    /* ... */
}

can hold HotelOrders or TravelOrders as long as their only remaining behavior is
    (review*;buy?)
    missing (subtype-wise) behavioral
    methods will never be called
    (can't be used anymore)

class User{
    map<Order> orders;
    /* ... */
}
Protocol Simulation (I)

- **choices:**
  - “main” more / “temp” less
  - → *Hidden choices in “temp”*

- **exceptions:**
  - “main” less / “temp” more
  - → «*Useless*» catches
Protocol Simulation (II)
Type System

- Simplified syntax (no “syntax sugar”)
- Only the core features of the language
- Basic typing judgment:

\[
\Delta_{\text{before}} \vdash E : T_{\text{result}} \mapsto \Delta_{\text{after}}
\]
Type System – if else

\[
\begin{align*}
\Delta \vdash E_{\text{cond}} : \text{boolean} \quad \Delta_{\text{cond}} \vdash E_{\text{if}} : T &\quad \Delta_{\text{if}} \quad \Delta_{\text{cond}} \vdash E_{\text{else}} : T \quad \Delta_{\text{else}} \\
\Delta \vdash \text{if}(E_{\text{cond}}) \ E_{\text{if}} \ \text{else} \ E_{\text{else}} : T &\quad \Delta_{\text{if}} \ \cap \ \Delta_{\text{else}}
\end{align*}
\]
Type System – try-catch and throw

\[ \Delta_{\text{try}} = \Delta \cup \langle \Omega; \Theta \rangle \cup (\Delta \leadsto N : \Delta_N) \]

\[ \Delta_{\text{try}} \vdash E_{\text{try}} : T_{\text{try}} \mapsto \Delta'_{\text{try}} \text{ stopped}(T_{\text{try}}) \]

\[ \Delta'_{\text{try}} = \Delta' \cup \langle \Omega; \Theta \rangle \cup (\Delta \leadsto N : \Delta_N) \]

\[ \Delta_{\text{catch}} = \Delta_N \cup \langle \Omega; \Theta \rangle \cup (n : N \#\text{stop} \times N \#\text{stop}) \]

\[ \Delta_{\text{catch}} \vdash E_{\text{catch}} : T_{\text{catch}} \mapsto \Delta'_N \cup \langle \Omega; \Theta \rangle \text{ stopped}(T_{\text{catch}}) \]

\[ \Delta \cup \langle \Omega; \Theta \rangle \vdash \text{try } E_{\text{try}} \text{ catch}(N \ n) \ E_{\text{catch}} : \text{void} \mapsto \Delta' \cap \Delta'_N \cup \langle \Omega; \Theta \rangle \]

(throw)

\[ \Delta \vdash E : T \mapsto \Delta' \text{ stopped}(T) \quad T = N \#P \]

\[ \langle \Omega; \Theta \rangle \in \Delta' \quad (\Delta'_{\text{try-catch}} \leadsto N : \Delta'_{\text{catch-N}}) \in \Theta \]

\[ \Delta' = \Delta'_{\text{try-catch}} \cup \Delta'_{\text{unreachable}} \text{ stopped}(\Delta'_{\text{unreachable}}) \quad \Delta_{\text{catch-N}} \triangleleft \Delta'_{\text{try-catch}} \]

\[ \Delta \vdash \text{throw } E : \text{void} \mapsto \emptyset \]
Part 3 – Closing Points

- Related Work
- Conclusions
- Future Work
Behavioral verification is a very broad topic.

There are several different approaches to the same core problem.

A quick overview of some of the most closely related work...
Related Work (II)


- **Futoshi Iwama, Atsushi Igarashi, and Naoki Kobayashi.** Resource usage analysis for a functional language with exceptions. 2006.

+ complex protocol expressiveness (even tough somewhat confusing)
+ concurrency
+ (some) exception handling

- ML based (functional language)
- no practical algorithms for checking (only formal type system)
Related Work (III)


```java
[WithProtocol("raw", "bound", "connected", "down")]
class Socket {
    [Creates("raw")]
    public Socket (...);

    [ChangesState("raw", "bound")]
    public void Bind (EndPoint localEP);

    [ChangesState("raw", "connected"). ChangesState("bound", "connected")]
    public void Connect (EndPoint remoteEP);

    [InState("connected")]
    public int Send (...);

    [InState("connected")]
    public int Receive (...);

    [ChangesState("connected", "down")]
    public void Shutdown (SocketShutdown how);

    [Disposes(State.Any)]
    public void Close ();
}
```

+ pre/post + state-machine + subtyping and parameter check
- no exception handling
- requires extensive annotations

Figure 5: A state-machine protocol for sockets.
Related Work (IV)


+ pre/post + session-types
+ inheritance and subtyping

- more limited approach:
  - no self calls
  - no behavioral termination
  - no behavioral exceptions
  - no ownership

```java
enum OpenResult {OK, NOT_FOUND, DENIED;}
enum Bool {FALSE, TRUE;}

interface FileReadToEnd {
    session Init
    where Init = &{open: \{OpenResult.OK: Open,
                           OpenResult.NOT_FOUND: end,
                           OpenResult.DENIED: end\}
                   Open = &{eof: \{Bool.TRUE: Close, Bool.FALSE: Read\}
                   Read = &{read:Open}
                   Close = &{close:end}
    
    requires Init
    ensures \{OpenResult.OK: Open, OpenResult.NOT_FOUND: end,
                           OpenResult.DENIED: end\}

    Null open()
    
    requires Open
    ensures \{Bool.TRUE: Close, Bool.FALSE: Read\}

    Null eof()

    requires Read
    ensures Open

    String read()

    requires Close
    ensures end

    Null close()
}
```

**Fig. 1.** The interface of a file that must be read to the end-of-file.
Related Work (V)

- Raymond Hu, Nobuko Yoshida, and Kohei Honda. 
  **Session-based distributed programming in java.** 2008.

```java
protocol placeOrder {
    begin. // Commence session.
    ![ // Can iterate:
        ![String>. // send String
        ?(Double) // receive Double
    ]*.
    !{ // Select one of:
        ACCEPT: !<Address>.?(Date),
        REJECT:
    }
}
Order protocol: Customer side.

protocol acceptOrder {
    begin.
    ![ // Can iterate:
        ![String>. // send String
        ?(Double)
    ]*.
    ?{ // Select one of:
        ACCEPT: ?(Address).!<Date>,
        REJECT:
    }
}
Order protocol: Agency side.
```

+ session-type based
- focused on channel communication (only)
- complex syntax
- not language transparent
- pairwise composition of protocols
- no behavioral exceptions

```java
boolean decided = false;
... // Set journey details.
s_ca.outwhile(!decided) {
    s_ca.send(journDetails);
    Double cost = agency.receive();
    ... // Set decided to true or
    ... // change details and retry
}

s_ac.inwhile() {
    String journDetails = s_ac.receive();
    ... // Calculate the cost.
    s_ac.send(price);
}
```
- **Cosimo Laneve and Luca Padovani.**  
  The must preorder revisited. 2007.

\[
\begin{align*}
  \text{rec } x. \text{Login}.(\text{InvalidLogin}.x & \oplus \text{ValidLogin}.\text{rec } y. \\
  \quad & \text{Query}.\text{Catalog}.(y + \text{Logout} + \text{rec } z.\text{Purchase}. \\
  \quad & \quad \quad \text{Accepted} \oplus \text{InvalidPayment}.(z + \text{Logout}) \oplus \text{OutOfStock}.(y + \text{Logout}))
\end{align*}
\]

- **Giuseppe Castagna, Nils Gesbert, and Luca Padovani.**  

\[
\sigma \overset{\text{def}}{=} \text{Login}.(\text{InvalidLogin} \oplus \text{ValidLogin}.\text{Query}. \\
\text{Catalog}.(\text{Logout} + \text{AddToCart}.(\text{Logout} + \text{Buy}.( \\
\text{Logout} + \text{CreditCard}.(\text{Valid} \oplus \text{Invalid} \\
\quad + \text{BankTransfer}.(\text{Valid} \oplus \text{Invalid}))))))
\]

- (only) focused on the contract layer

+ flexible and interesting operations

+ sub-contract very similar to our behavioral sub-typing
Example – WS-CDL (wrapper)

```java
class Service{
    usage &l( login [ InvalidLogin: l ] ;
        &q( query;
            ( q + logout +
                &p( purchase[ InvalidPayment: p+logout | OutOfStock: q+logout ] ) ) )
    ) )

    login(string username, string password)
        throws InvalidLogin { ... }

    Catalog query(string query) { ... }

    string purchase(Purchase purchase)
        throws InvalidPayment, OutOfStock { ... }

    logout() { ... }
}
```

Image from: Laneve, Padovani. *The must preorder revisited – an algebraic theory for web services contracts*
Conclusions

+ minimalistic experimental language
+ formal description of the type system
+ working prototype (and publicly available)
  ( parser + interpreter + type checker + run-time system )
+ some interesting examples
Future Work

> soundness proof

- concurrency

- prototype improvements:
  query (object pool) by protocol
  protocol expressiveness
  error messages friendliness
  improve/simplify code base

...
The End.

Yak prototype

(http://ctp.di.fct.unl.pt/yak/)