Alcino Cunha

SPECIFICATION AND MODELING

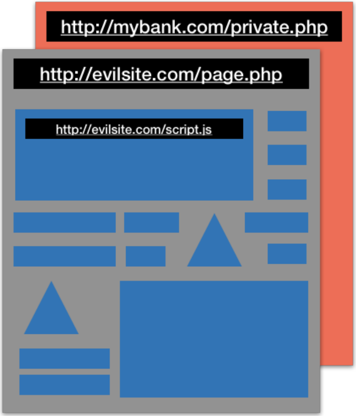
TYPE SYSTEM

Universidade do Minho & INESC TEC

2019/20

SAME ORIGIN POLICY

SAME ORIGIN POLICY



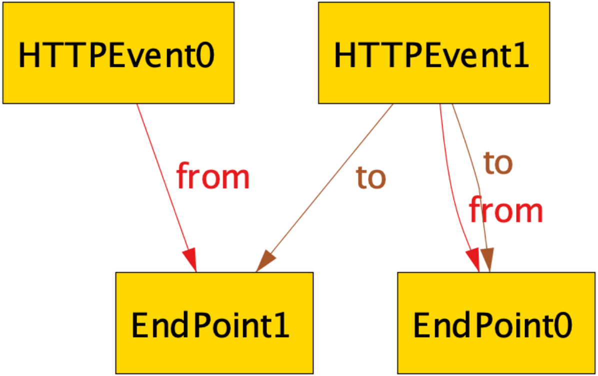
Understand and verify the policy:

- Resources can only access resources from the same origin

END-POINTS AND HTTP EVENTS

```
sig EndPoint {}

sig HTTPEvent {
    from : set EndPoint,
    to : set EndPoint
}
```

END-POINTS AND HTTP EVENTS



END-POINTS AND HTTP EVENTS

```
sig EndPoint {}

sig HTTPEvent {
    from : set EndPoint,
    to : set EndPoint
}

fact {
    all h : HTTPEvent | one h.from and one h.to
}
```

MULTIPLICITIES

MULTIPLICITIES IN SIGNATURE DECLARATIONS

$m \in \{$ **some** , **lone** , **one** $\}$

m **sig** A { }

$\equiv$

sig A { }

fact { *m* A }

MULTIPLICITIES IN FIELD DECLARATIONS

$m \in \{$ **set** , **some** , **lone** , **one** $\}$

```
sig A { r : m B }
```

$\equiv$

```
sig A { r : set B }
fact { all a : A | m a.r }
```

if $m \neq$ **set**

```
sig A { r : B }  ≡  sig A { r : one B }
```

MULTIPLICITIES IN FIELD DECLARATIONS

$m, n \in \{$ **set** , **some** , **lone** , **one** $\}$

sig A { r : B m -> n C }

$\equiv$

sig A { r : B **set** -> **set** C}

fact { **all** a : A, b : B | n b.(a.r) } if $n \neq$ **set**

fact { **all** a : A, c : C | m (a.r).c } if $m \neq$ **set**

sig A { r : B -> C} $\equiv$ **sig** A { r : B **set** -> **set** C}

MULTIPLICITIES IN FORMULAS

$m, n \in$ { **set** , **some** , **lone** , **one** }

Φ **in** *A m* -> *n B*

```
sig A { r : B m -> n C }
```

$\equiv$

```
sig A { r : B set -> set C}
fact { all a : A | a.r in B m -> n C }
```

BESTIARY

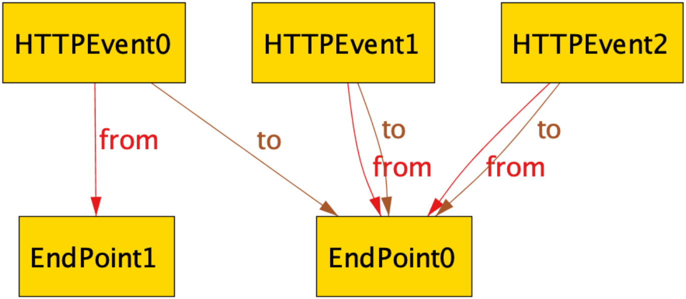
```
r in A -> some B        // r is entire
r in A -> lone B        // r is simple
r in A some -> B        // r is surjective
r in A lone -> B        // r is injective

r in A -> one B         // r is a function (entire + simple)
r in A lone -> some B   // r is a representation (entire + injective)
r in A some -> lone B   // r is an abstraction (simple + surjective)

r in A lone -> one B    // r is an injection (function + representation)
r in A some -> one B    // r is an surjection (function + abstraction)

r in A one -> one B     // r is a bijection (injection + surjection)
```

SAME ORIGIN POLICY

END-POINTS AND HTTP EVENTS

```
sig EndPoint {}

sig HTTPEvent {
    from : one EndPoint,
    to : one EndPoint
}
```

END-POINTS AND HTTP EVENTS



SERVERS AND CLIENTS

```
sig EndPoint {}

sig Server in EndPoint {}
sig Client in EndPoint {}

fact {
    no Server & Client
    EndPoint = Server + Client
}
```

SUB-TYPING

EXTENSION SIGNATURES

- All extensions of a signature are disjoint
- Signatures that are not extended are known as *atomic*

```
sig A {}

sig B,C extends A {}
```

≡

```
sig A {}

sig B,C in A {}

fact { no B & C }
```

ABSTRACT SIGNATURES

- An abstract signature has no atoms outside its extensions

```
abstract sig A {}
sig B,C extends A {}
```

≡

```
sig A {}
sig B,C in A {}
fact { no B & C }
fact { A = B + C }
```

ENUMERATIONS

```
abstract sig A {}

one sig B,C,D extends A {}
```

$\equiv$

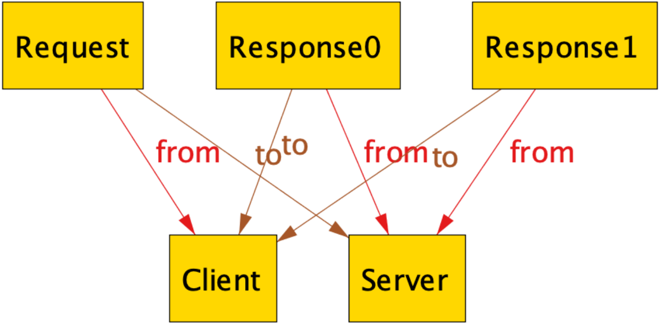
```
enum A {B,C,D}
```

SAME ORIGIN POLICY

SERVERS, CLIENTS, REQUESTS, AND RESPONSES

```
abstract sig EndPoint {}
sig Server,Client extends EndPoint {}

abstract sig HTTPEvent {
    from : one EndPoint,
    to : one EndPoint
}
sig Request, Response extends HTTPEvent {}

fact {
    Request.from + Response.to in Client
    Request.to + Response.from in Server
}
```

SERVERS, CLIENTS, REQUESTS, AND RESPONSES



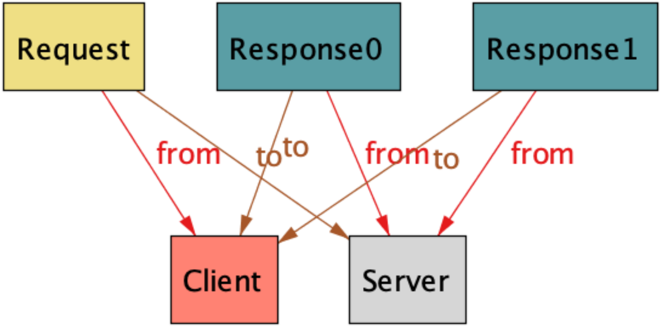
THEMES

THEMES

- Complex instances are difficult to understand
- It is possible to improve the visualisation with *themes*
  - Different colors and shapes for different entities
  - Hide irrelevant entities
  - Project over a signature
  - ...
- Good themes considerably simplify the validation task

SAME ORIGIN POLICY

SERVERS, CLIENTS, REQUESTS, AND RESPONSES



OVERLOADING

OVERLOADING

- Relations can be *overloaded*

```
sig A { }
sig B extends A { r : set A }
sig C extends A { r : set A }
```

- As long as domain signatures are disjoint

```
sig A { r : set A }
sig B extends A { r : set A }
```

A type error has occurred:
Two overlapping signatures cannot have two fields with the same name "r":

SAME ORIGIN POLICY

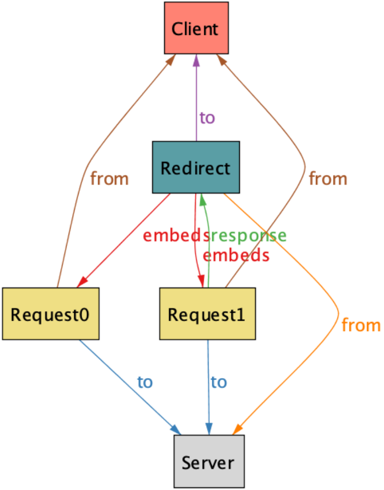
SERVERS, CLIENTS, REQUESTS, AND RESPONSES

```
abstract sig EndPoint {}
sig Server,Client extends EndPoint {}

abstract sig HTTPEvent {}
sig Request extends HTTPEvent {
    from : one Client,
    to : one Server
}
sig Response extends HTTPEvent {
    from : one Server,
    to : one Client
}
```

REDIRECTS AND LINKING REQUESTS TO RESPONSES

```
abstract sig EndPoint {}
sig Server,Client extends EndPoint {}

abstract sig HTTPEvent {}
sig Request extends HTTPEvent {
    from : one Client,
    to : one Server,
    response : lone Response
}
sig Response extends HTTPEvent {
    from : one Server,
    to : one Client,
    embeds : set Request
}
sig Redirect extends Response {}
```

REDIRECTS AND LINKING REQUESTS TO RESPONSES



TYPE ERRORS

TYPE ERRORS

- A good type system for modeling should support sub-typing and overloading
- But what should be the type errors in this setting?

Formula	Result
`some Request.response`	OK
`some Redirect.response`	Error
`no Redirect.embeds`	??
`some HTTPEvent.response`	??
`no (Request + Redirect).response`	??
`some HTTPEvent.to`	??

TYPE ERRORS

- An expression may trigger an *irrelevance* error
  - If it can be replaced by the empty relation without affecting the meaning of the enclosing formula
- An overloaded relation may trigger an *ambiguity* error
  - If it cannot be decided which case it refers to

Expression	Result	Why
`some Request.response`	OK	
`some Redirect.response`	Error	`Redirect.response` is irrelevant
`no Redirect.embeds`	OK	
`some HTTPEvent.response`	OK	
`no (Request + Redirect).response`	Error	`Redirect` is irrelevant
`some HTTPEvent.to`	Error	`to` is ambiguous

TYPES

- The type of an expression is a set of tuples of atomic types
- The type characterises the upper-bound of the expression - which tuples may be contained in it
- For every non abstract signature we assume the existence of an atomic type containing its *remainder*
  - `$Response` is the remainder of `Response`
  - It contains the atoms of `Response` that are not in `Redirect`
- Overloaded relations are treated as the union of all cases
  - `to` is an alias to `Request<:to + Response<:to`

TYPE INFERENCE

- The type inference mechanism determines the type of all relational expressions
  - If the type is empty the expression is irrelevant and an error is reported
  - In an overloaded relation, only one of the disjunct cases can be relevant, otherwise an ambiguity error is reported
- The type inference mechanism is guided by the abstract syntax tree and proceeds in two phases
  - A first bottom-up phase computes the *bounding types* $\Phi \uparrow T$
  - These are refined by the second top-down phase to compute the *relevance types* $\Phi \downarrow T$

BOUNDING TYPE INFERENCE

- The bounding types of the declared signatures and relations are inferred from their declarations
- The bounding types of compound expressions are computed from the bounding types of sub-expressions using the same operator

BOUNDING TYPE INFERENCE

```
Request ↑ {(Request)}

Redirect ↑ {(Redirect)}

Request + Redirect ↑ {(Request),(Redirect)}

response ↑ {(Request,Redirect),(Request,$Response)}

(Request + Redirect).response ↑ {(Redirect),($Response)}
```

RELEVANCE TYPE INFERENCE

- The relevance type of the outermost expression is equal to its bounding type
- The relevance type of sub-expressions are computed by determining which tuples of its bounding type contributed to the relevance type of the parent expression

RELEVANCE TYPE INFERENCE

(Request + Redirect).response ↓ {(Redirect),($Response)}

response ↑ {(Request,Redirect),(Request,$Response)}

Request + Redirect ↑ {(Request),(Redirect)}

response ↓ {(Request,Redirect),(Request,$Response)}

Request + Redirect ↓ {(Request)}

Request ↑ {(Request)}

Redirect ↑ {(Redirect)}

Request ↓ {(Request)}

Redirect ↓ {}

SAME ORIGIN POLICY

REDIRECTS AND LINKING REQUESTS TO RESPONSES

```
fact RequestResponse {
    -- Every response is associated with exactly one request
    all r : Response | one response.r

    -- Every response is to the endpoint its request was from,
    -- and from the endpoint its request was to
    all r : Response | r.to = response.r.from and
                       r.from = response.r.to

    -- A request cannot be embedded in a response to itself
    all r : Request | r not in r.^(response.embeds)
}
```

TRACKING ORIGINS

```
abstract sig HTTPEvent {
    origin : one EndPoint
}
```

TRACKING ORIGINS

```
fact Origin {
    -- A redirect has the same origin as the original request
    all r : Redirect | r.origin = (response.r).origin
    -- The origin of other responses is the server they came from
    all r : Response-Redirect | r.origin = r.from

    -- The origin of a non-embedded request is the endpoint it came from
    all r : Request | no embeds.r implies r.origin in r.from
    -- Otherwise it is the same origin of the embedding response
    all r : Response, e : r.embeds | e.origin = r.origin
}
```

TRACKING ORIGINS

```
pred EnforceOrigins [s : Server] {
    -- A server enforces the origin header if
    -- it allows incoming requests only if they originate
    -- at that server or at the client that sent the request
    all r : Request {
        r.to = s implies r.origin = r.to or r.origin = r.from
    }
}
```

TRACKING CAUSALITY

```
sig Server extends EndPoint {
    causes : set HTTPEvent
}
```

TRACKING CAUSALITY

```
fact Causality {
    -- An event is caused by a server if and only if
    -- it is from that server, or is embedded in a response
    -- that the server causes
    all e : HTTPEvent, s : Server {
        e in s.causes
        iff
        (e.from = s
         or
         some r : Response | e in r.embeds and r in s.causes)
    }
}
```

CHECKING SECURITY

```
assert Secure {
    -- Assuming the client never sends requests directly
    -- to "bad" servers, a "good" server that is enforcing the origin
    -- header cannot receive a request caused by a "bad" server
    all good, bad : Server {
        (EnforceOrigins[good] and
        no r : Request | r.to = bad and r.origin in Client)
        implies
        no r : Request | r.to = good and r in bad.causes
    }
}
check Secure for 5 HTTPEvent, 3 EndPoint
```

COUNTER-EXAMPLE