EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security

Ran Canetti[1,2], Alley Stoughton[1], Mayank Varia[1]

[1] Boston University [2] Tel Aviv University

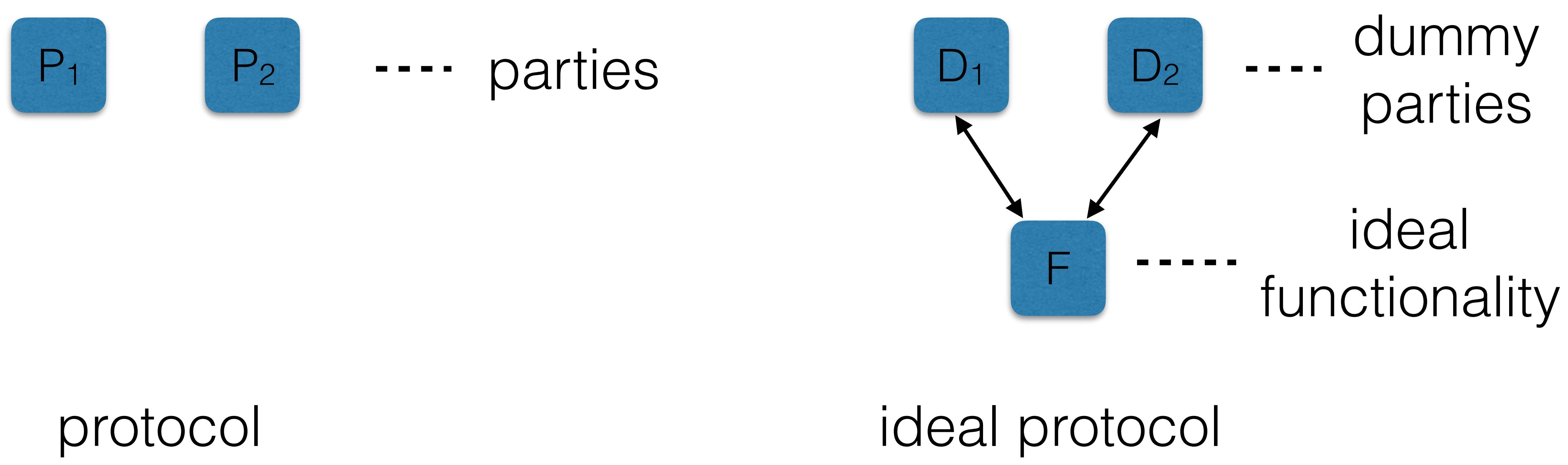
32nd IEEE Computer Security Foundations Symposium
June 25-28, 2019, Hoboken, NJ, USA

Universally Composable Security

- Universally Composable (UC) Security (Canetti, …) is a refinement of the real/ideal paradigm supporting modular proof development
- In UC, a protocol interacts with
  - an *environment*, which supplies protocol inputs and consumes protocol outputs, and
  - an *adversary*, which is given certain powers to observe or corrupt the protocol
- The environment and adversary (may) communicate
- UC uses a *coroutine style* of message passing in which control is transferred along with data

Universally Composable Security

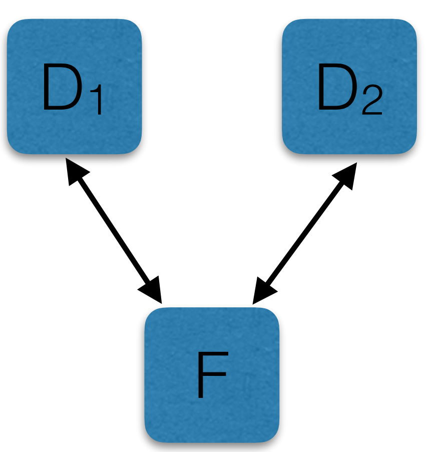
- A *protocol* consists of some number of protocol parties
- An *ideal protocol* consists of an *ideal functionality* combined with *dummy parties* transferring inputs/outputs to/from the ideal functionality
  - Specifies desired functionality, plus leakage to simulator



Universally Composable Security

- Because we always work with an ideal functionality and its dummy parties *as a unit*, and we needed a *neutral term* for a protocol or an ideal protocol, we settled on

protocol

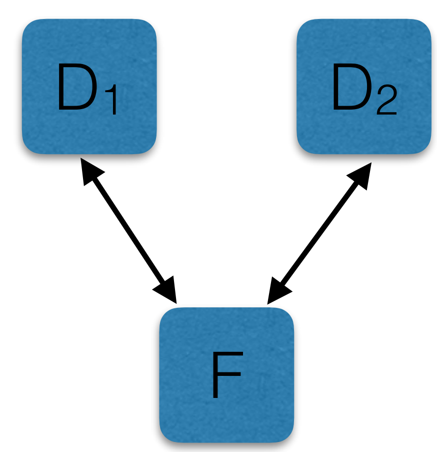


ideal protocol

Universally Composable Security

- Because we always work with an ideal functionality and its dummy parties *as a unit*, and we needed a *neutral term* for a protocol or an ideal protocol, we settled on:
  - calling ideal protocols *ideal functionalities*, and
  - calling protocols real functionalities
- Thus a *functionality* is either a real or ideal functionality

real functionality



ideal functionality

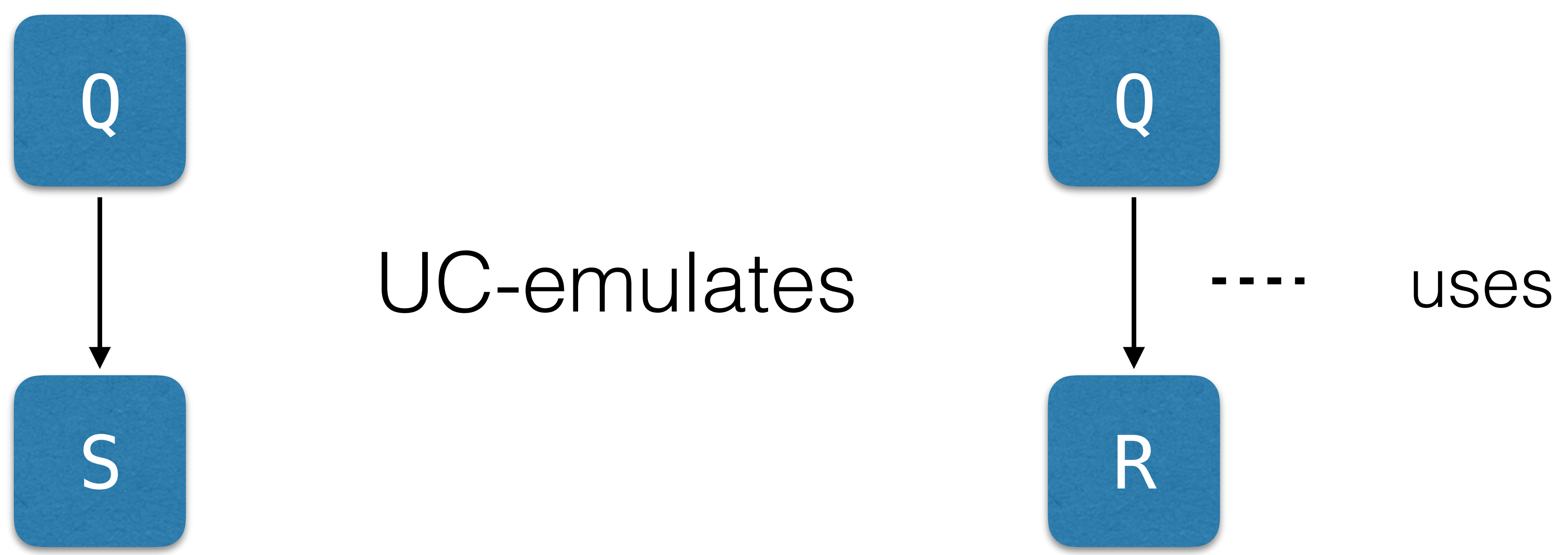
Universal Composable Security

- A real functionality `RF` *UC-emulates* an ideal functionality `IF` iff, there is an efficient, black box simulator `Sim`, such that, for all efficient adversaries `Adv`, and for all efficient environments `Env`, `Env` can't tell if it is interacting with
  - `RF/Adv` (the real game), or
  - `IF/Sim(Adv)` (the ideal game)
- More precisely, the environment yields a boolean judgment, and we want the absolute value of the difference between the probabilities of the environment returning true in the real and ideal games to be small
- This definition is the same when the second functionality is also a real functionality
- UC-emulation is trivially transitive: the simulators compose

Universal Composable Security

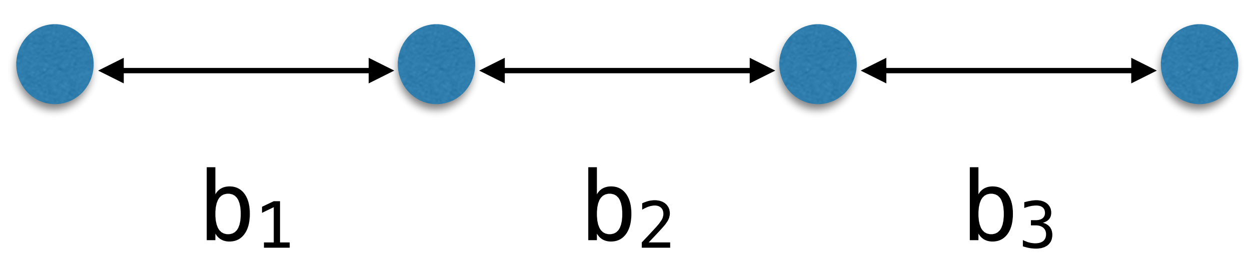
- The UC Composition Theorem says that:
  - if **S** UC-emulates **R**, and **Q** is a functionality using **R**,

  then if we change **Q** to use **S** instead of **R** (this is the UC composition operator), the result will UC-emulate **Q**



Sequence of Games Approach

- In general, it takes some number of steps to connect real and ideal games
  - Each step establishes an upper bound on the ability of the environment to discriminate between the two games
  - The sum of these upper bounds is an upper bound on the ability of the environment to discriminate between the real and ideal games
  - Steps may be proved by reductions, up-to bad reasoning, code motion, …



Proof Mechanization

- Several frameworks have been developed for mechanizing cryptographic security proofs in the sequence of games approach:
  - CryptoVerif (Blanchet) is semi-automated, guided by hints
  - FCF (Petcher & Morrisett) is embedded in Coq
  - CryptHOL (Basin, Lochbihler & Sefidgar) is embedded in Isabelle/HOL
  - EasyCrypt (Barthe, Grégoire, Strub, …, Stoughton, …) is a standalone proof assistant, with a fairly small and well-studied TCB
- We're using EasyCrypt partly because it directly handles modules — including abstract ones like adversaries — with their own local, private state

EasyCrypt's Modules

- Modules consist of global variables and procedures
- Modules may be parameterized, e.g., by adversaries or environments
- Procedures are written in a simple imperative language, with while loops and random assignments (choosing values from probability sub-distributions)

EasyCrypt's Logics

- EasyCrypt has four logics:
  - a Probabilistic Relational Hoare Logic (pRHL) for proving relations between pairs of games
  - a Probabilisitic Hoare logic (pHL) for proving probabilistic facts about single games
  - an ordinary Hoare logic (HL)
  - an ambient higher-order logic for proving mathematical facts and connecting judgements from the other logics

EasyCrypt's Proofs and Theories

- Proofs are structured as sequences of lemmas
- Lemmas are proved using tactics, as in Coq
- EasyCrypt theories may be used to group definitions, modules and lemmas together
- Theories may be specialized via cloning

UC in EasyCrypt

- We are in the early stages of researching how UC security may be mechanized in EasyCrypt
- A major challenge is how to deal with UC's coroutine style of communication in EasyCrypt's procedural programming language
- Our approach is to give functionalities, the adversary and parts of the environment *addresses* (lists of integers), and to build abstractions that route *messages* to their destinations
    - The empty list, `[]`, is the root address of the environment

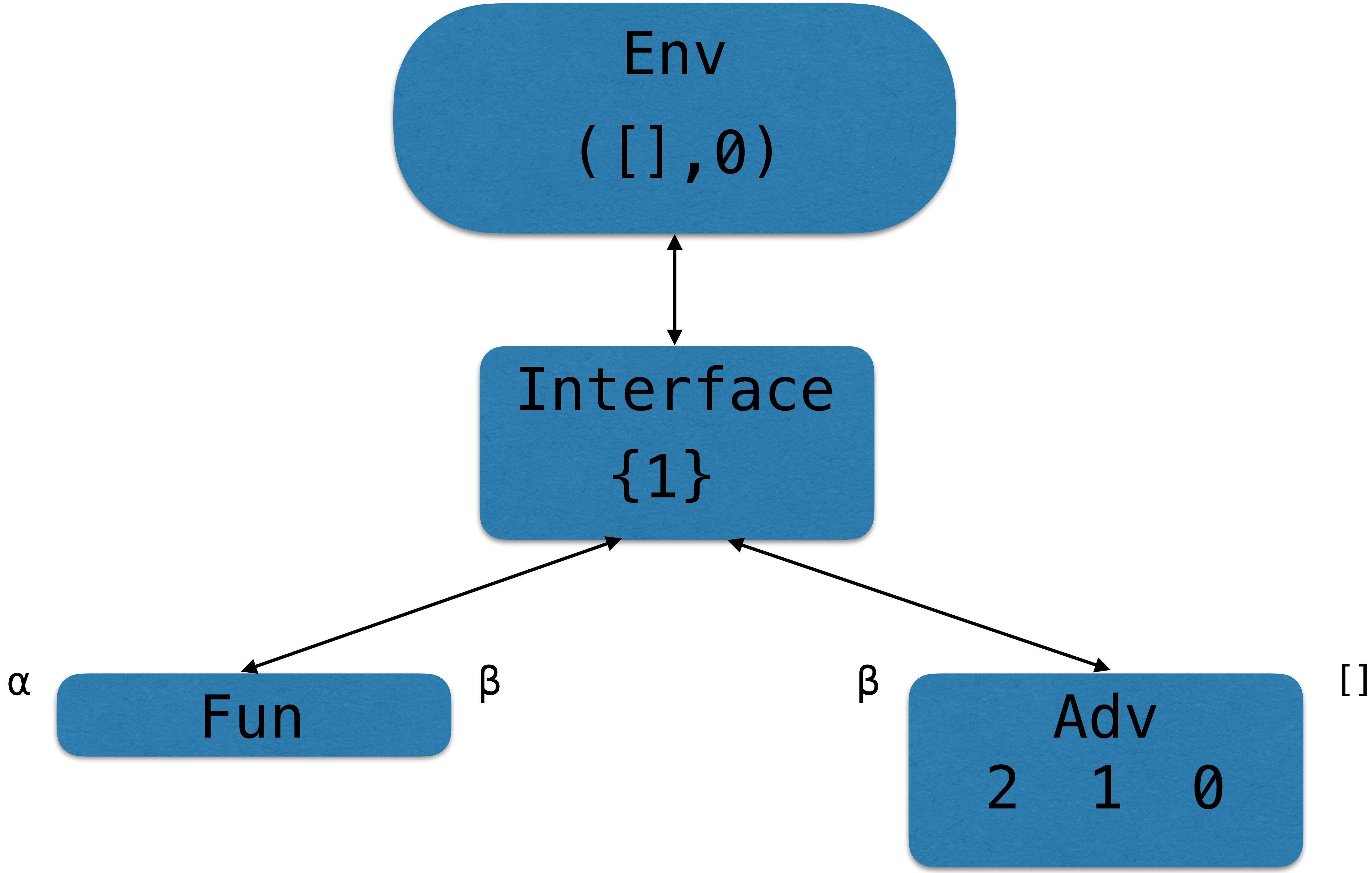
UC in EasyCrypt

- On top of the addressing system, we have a simple naming scheme based on *ports* (α, i), where α is an address, and i is an integer (a *port index*)
  - `([], 0)` is the environment's default port
  - Each of a functionality's parties has some number of ports
- Messages can be
  - "*direct*" — providing functionality inputs or reporting functionality outputs; or
  - "*adversarial*" — communication between environment and adversary, or functionality and adversary

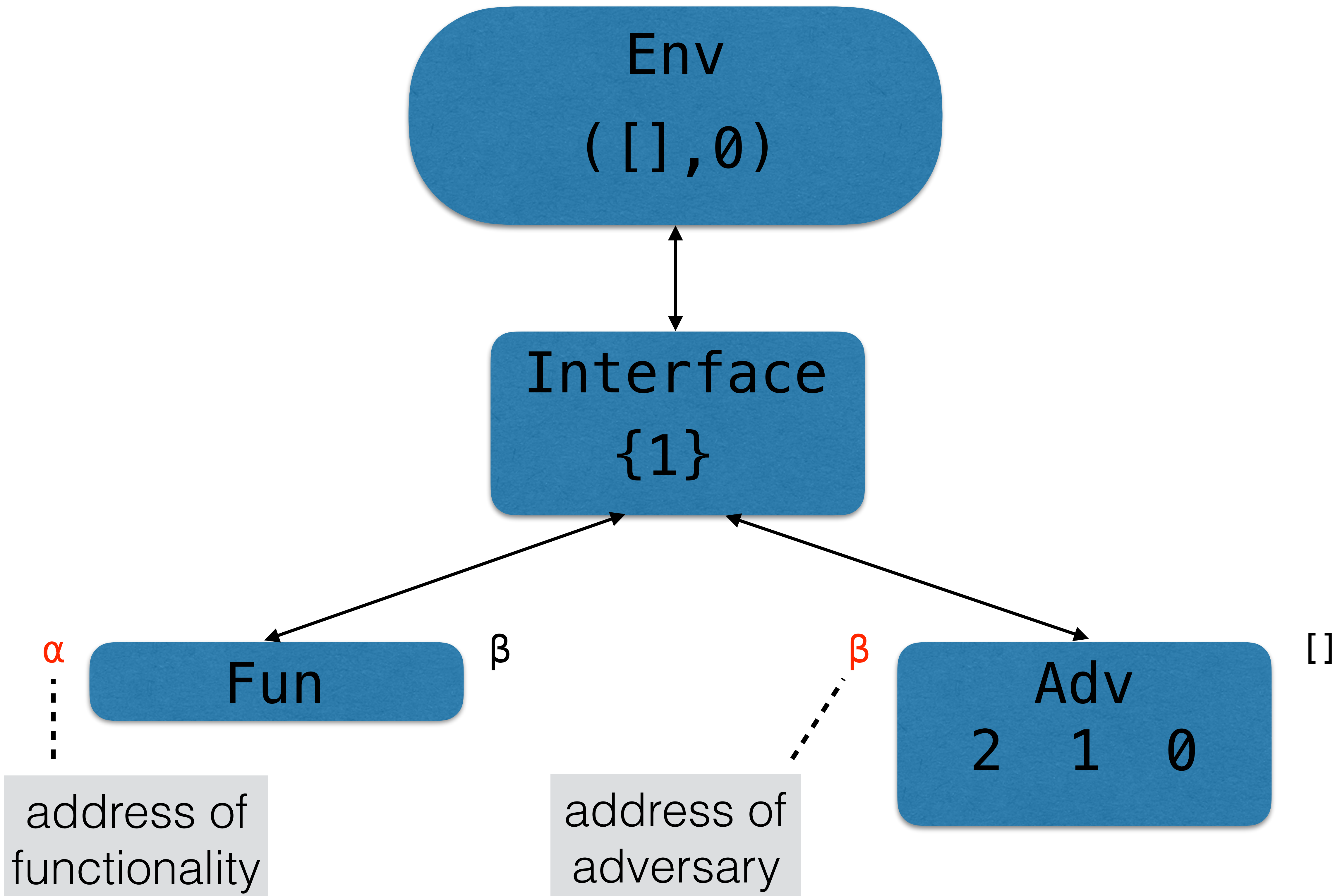
Functionalities in EasyCrypt

- We realize functionalities as modules
  - The parties of a functionality live within a single module
- Functionalities may have sub-functionalities, with sub-addresses
  - A parent functionality can choose which messages from the environment to forward to its sub-functionalities
- Modules in EasyCrypt may be parameterized, allowing the UC composition operator to be realized as module application
- Multiple instances of functionalities can be statically created using EasyCrypt's cloning mechanism

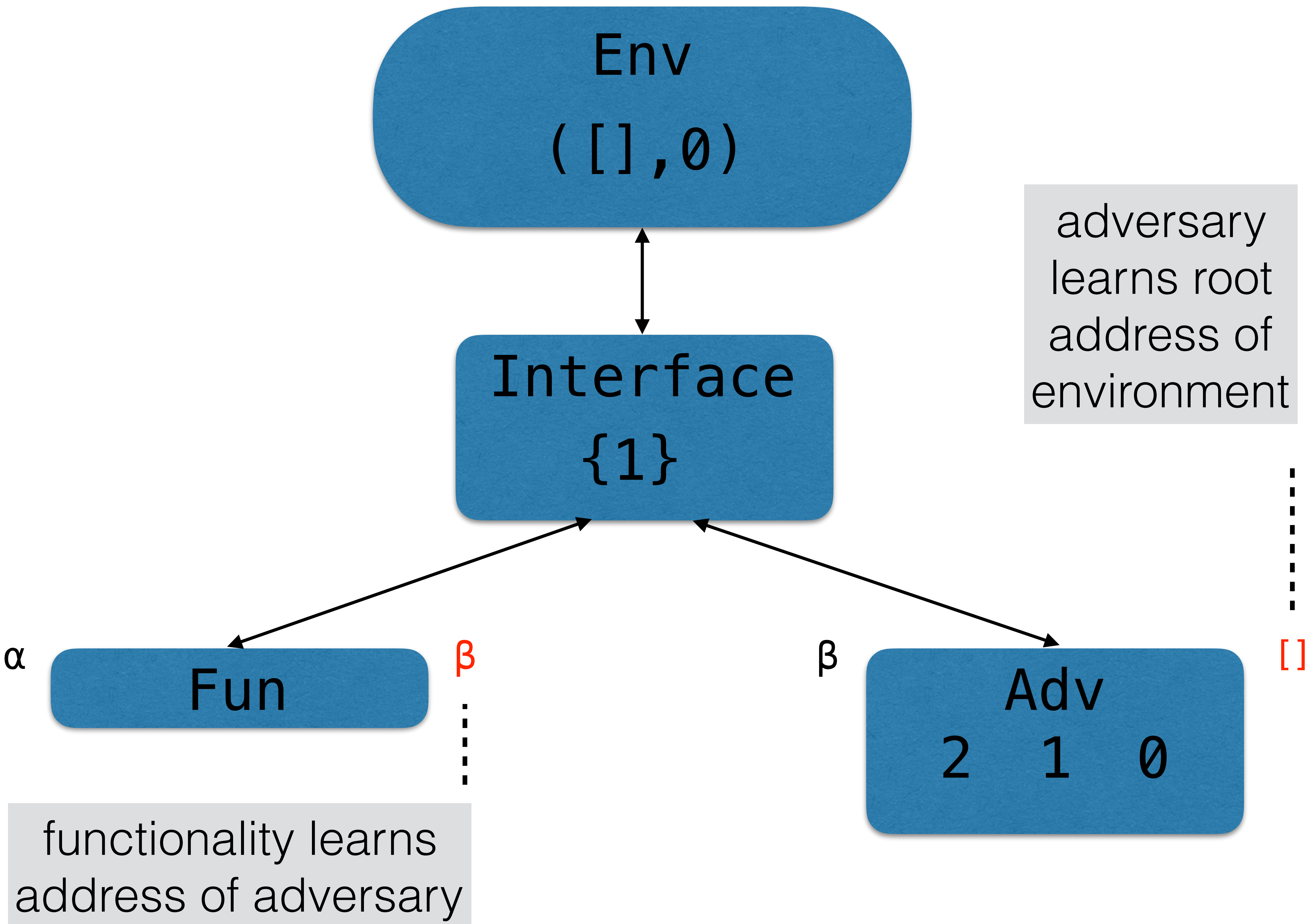
Interface Firewall



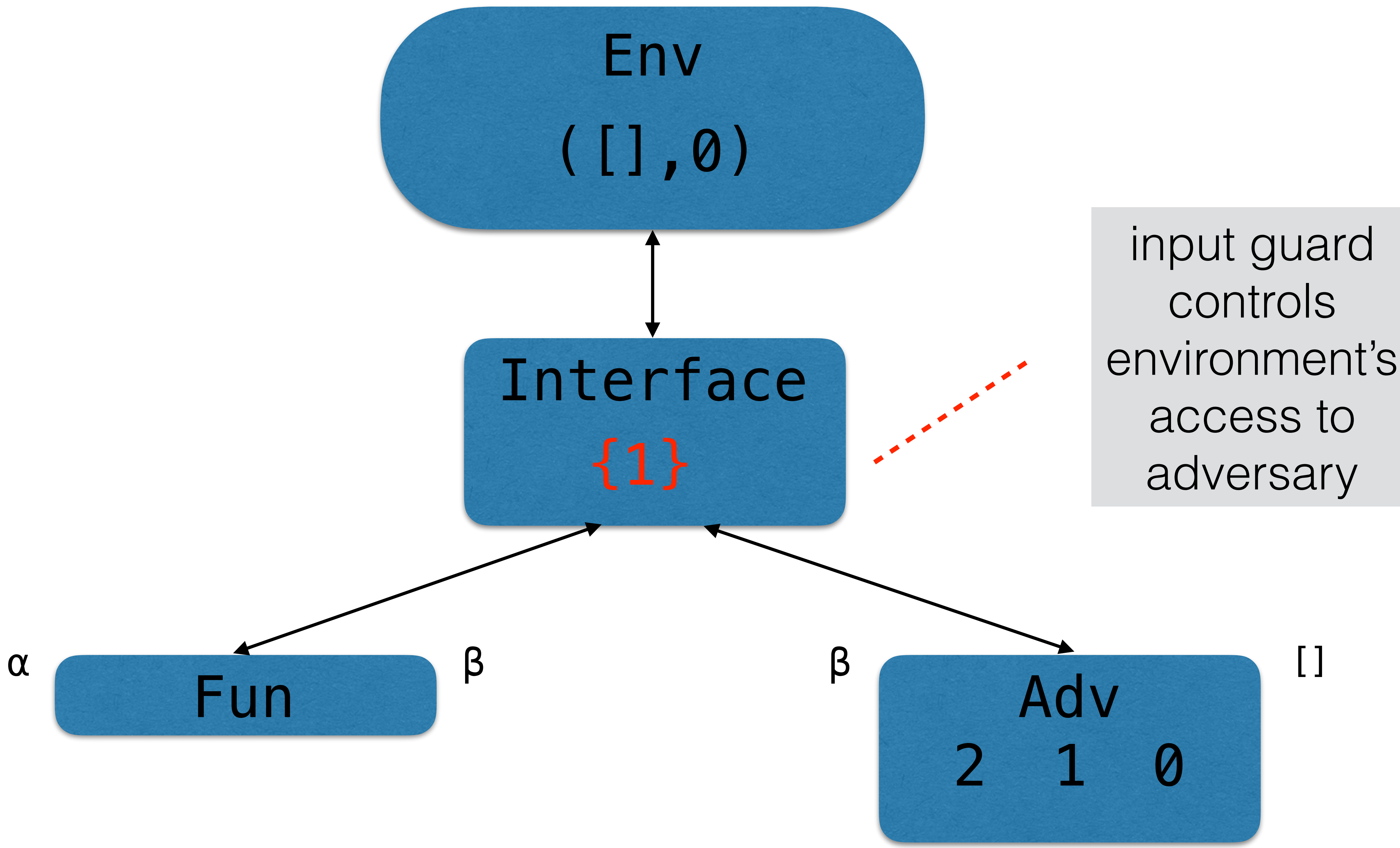
Interface Firewall



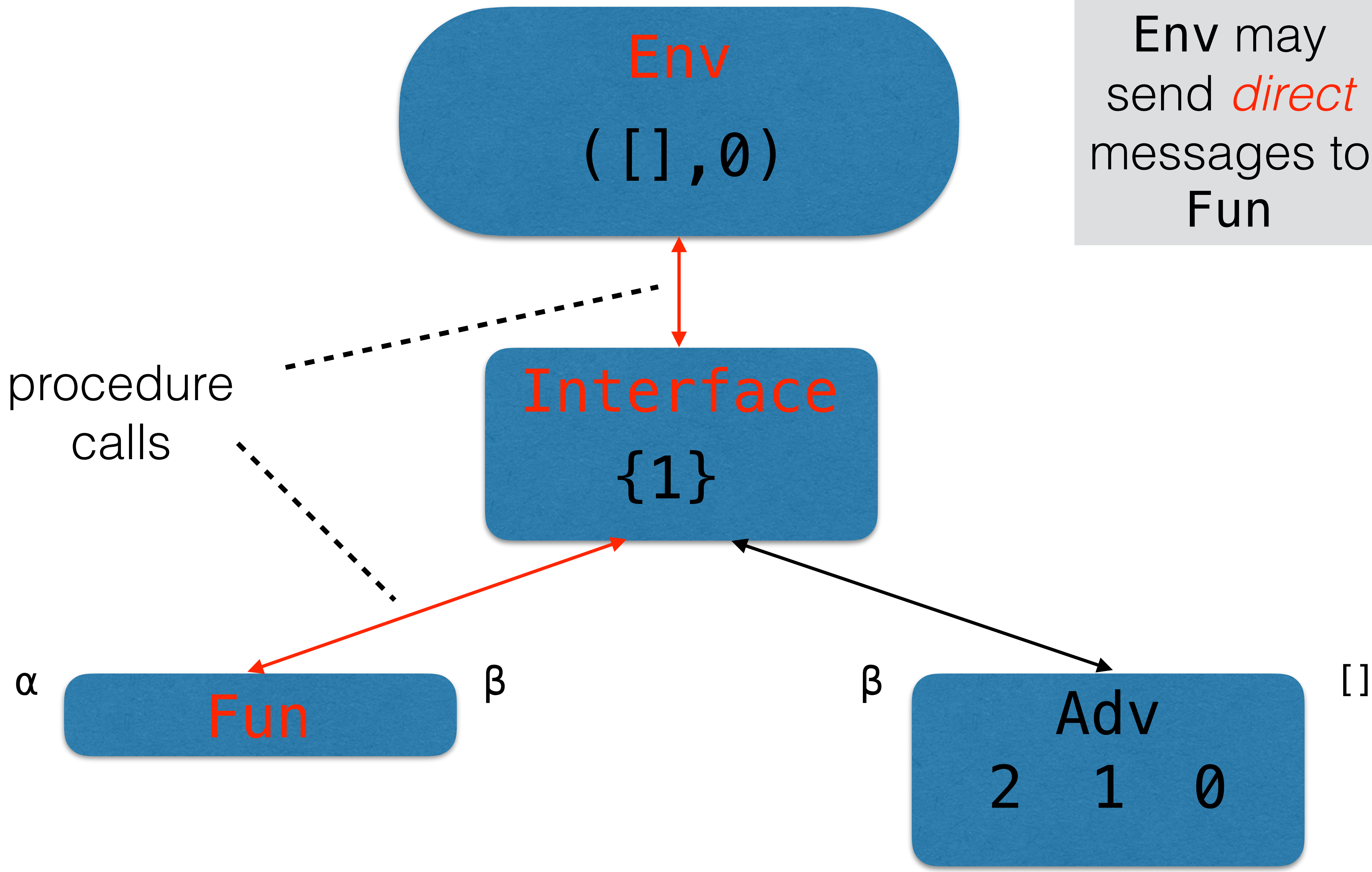
Interface Firewall



Interface Firewall



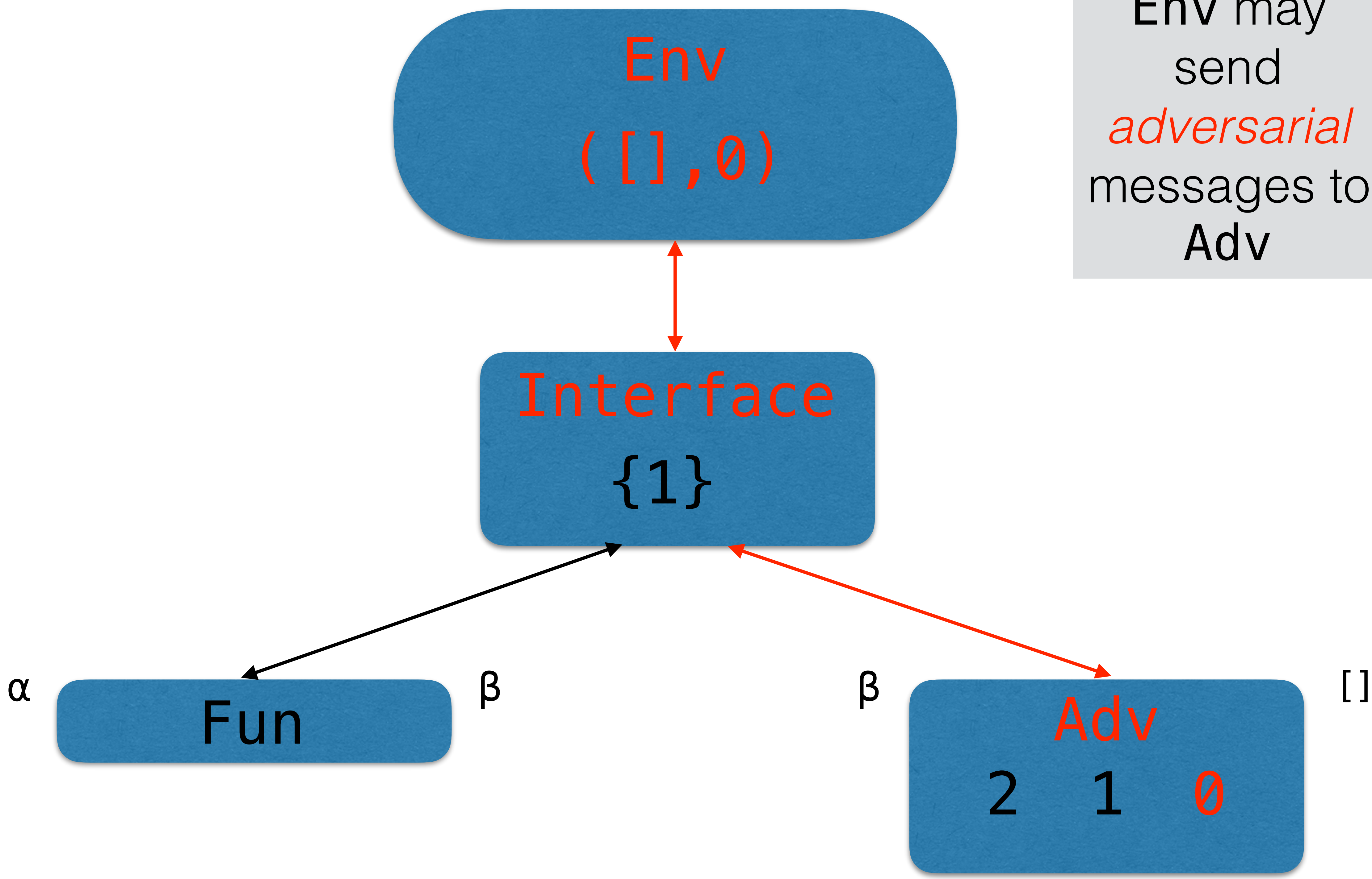
Interface Firewall



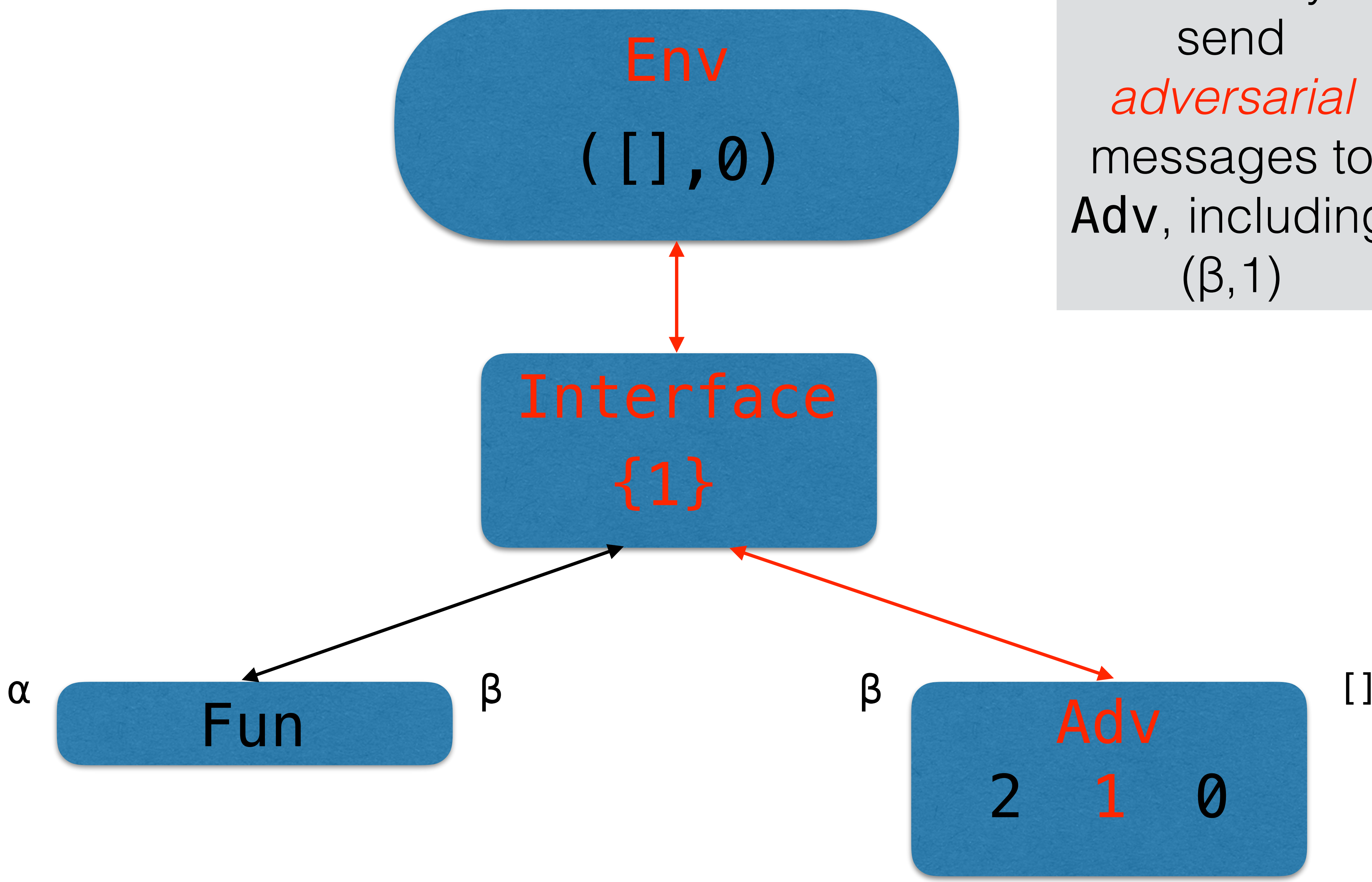
Env may send *direct* messages to **Fun**

Interface Firewall

Env may send *adversarial* messages to **Adv**

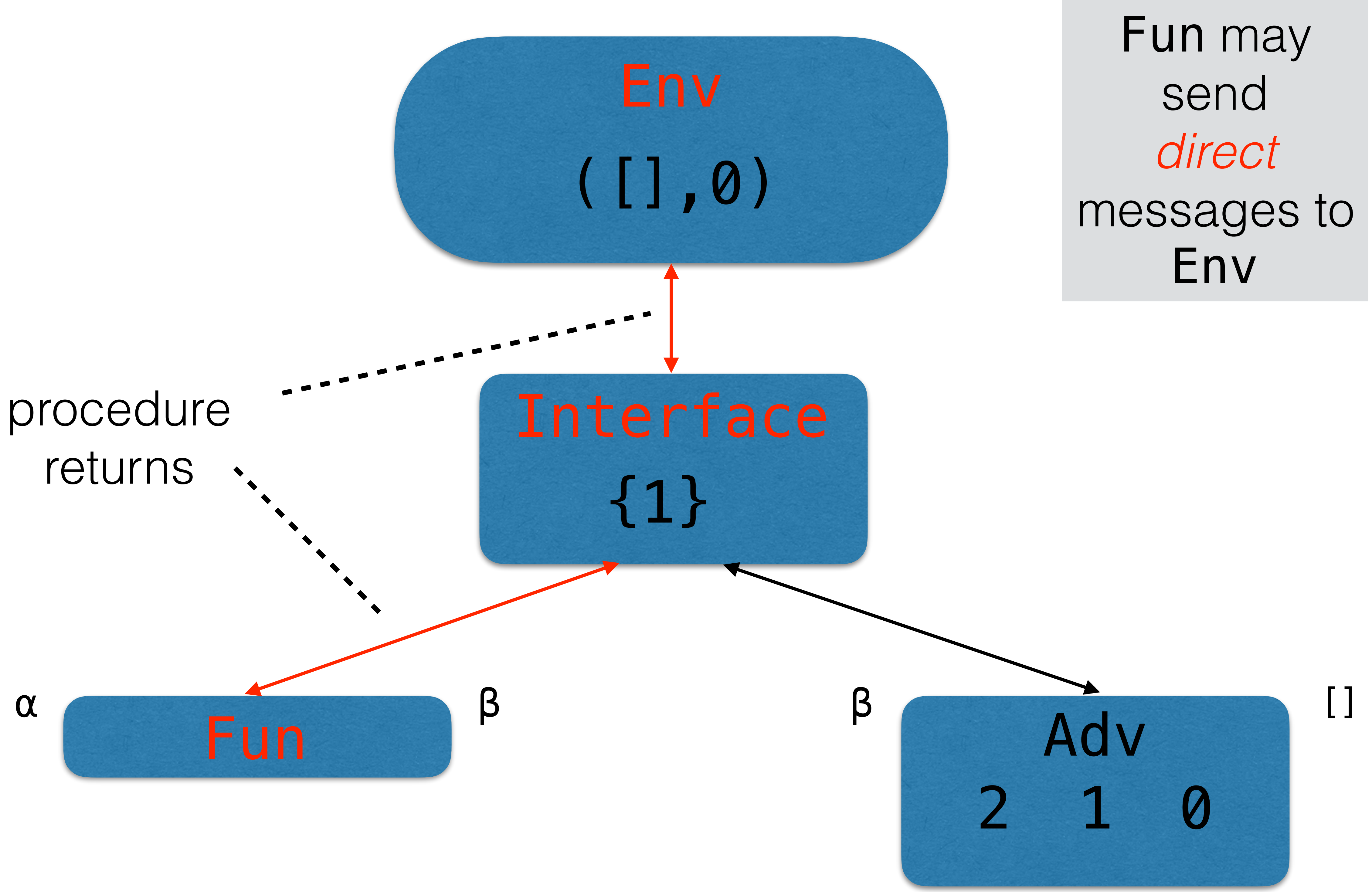


Interface Firewall

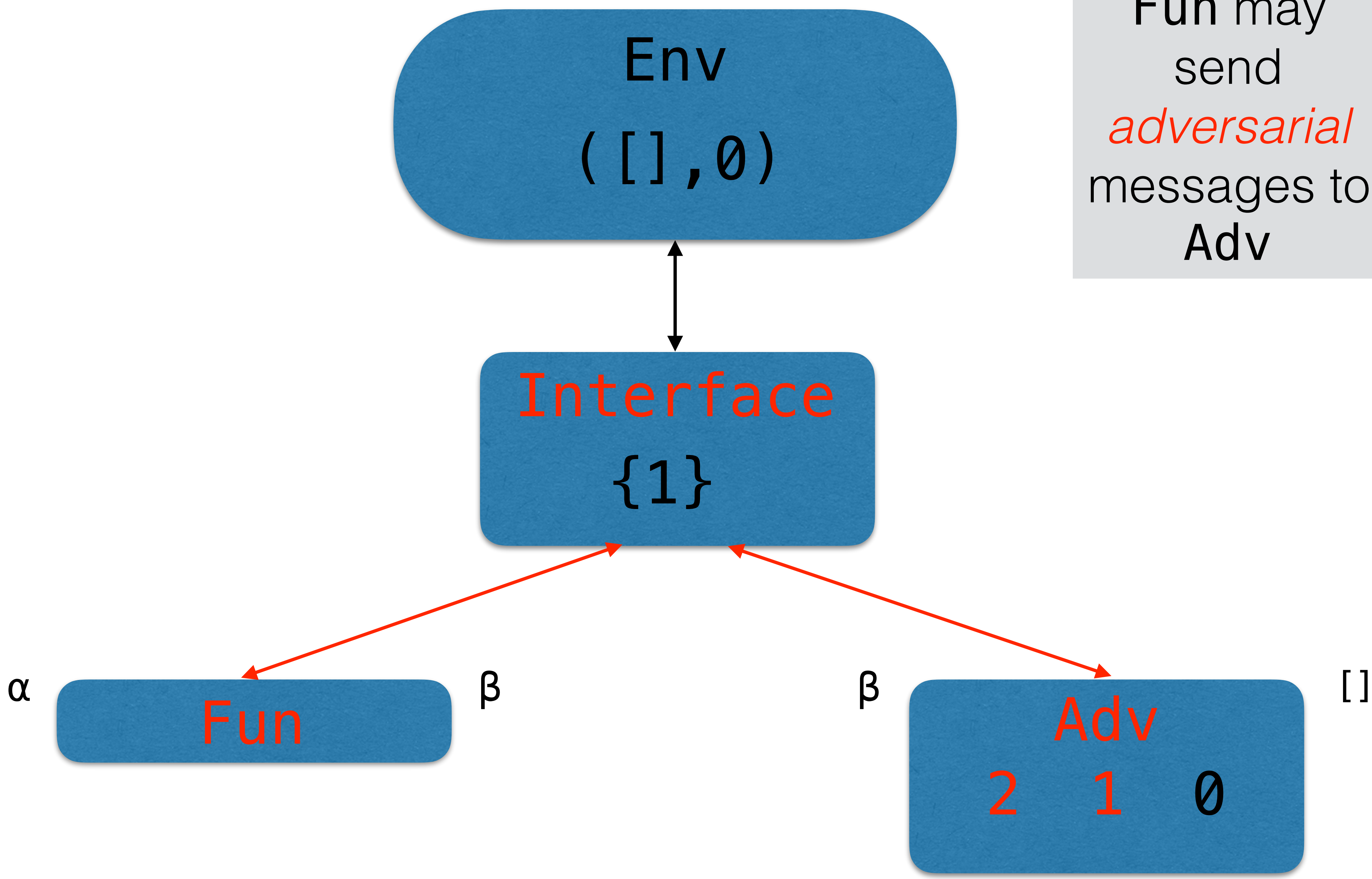


Env may send *adversarial* messages to **Adv**, including (β,1)

Interface Firewall

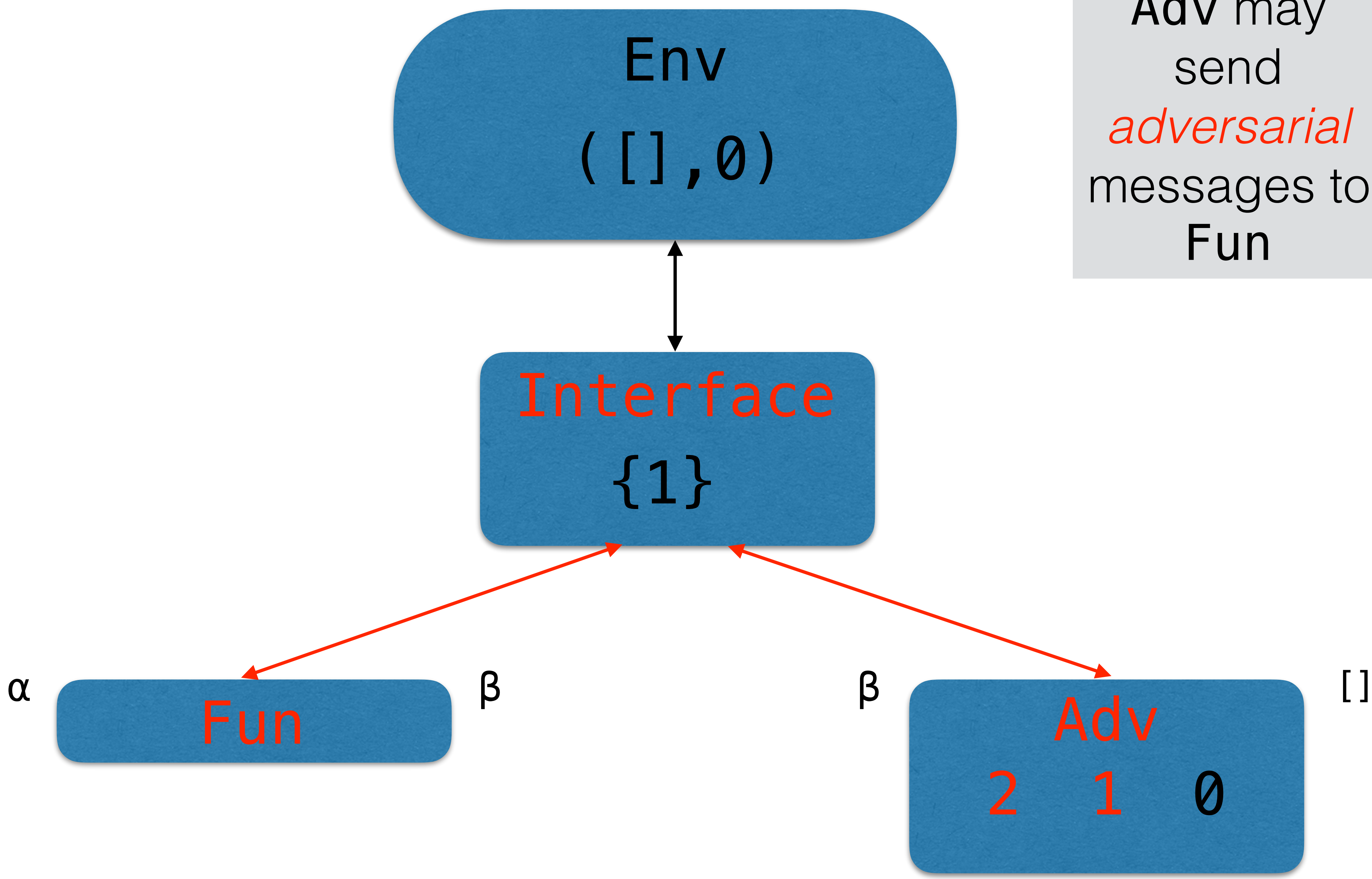


Interface Firewall



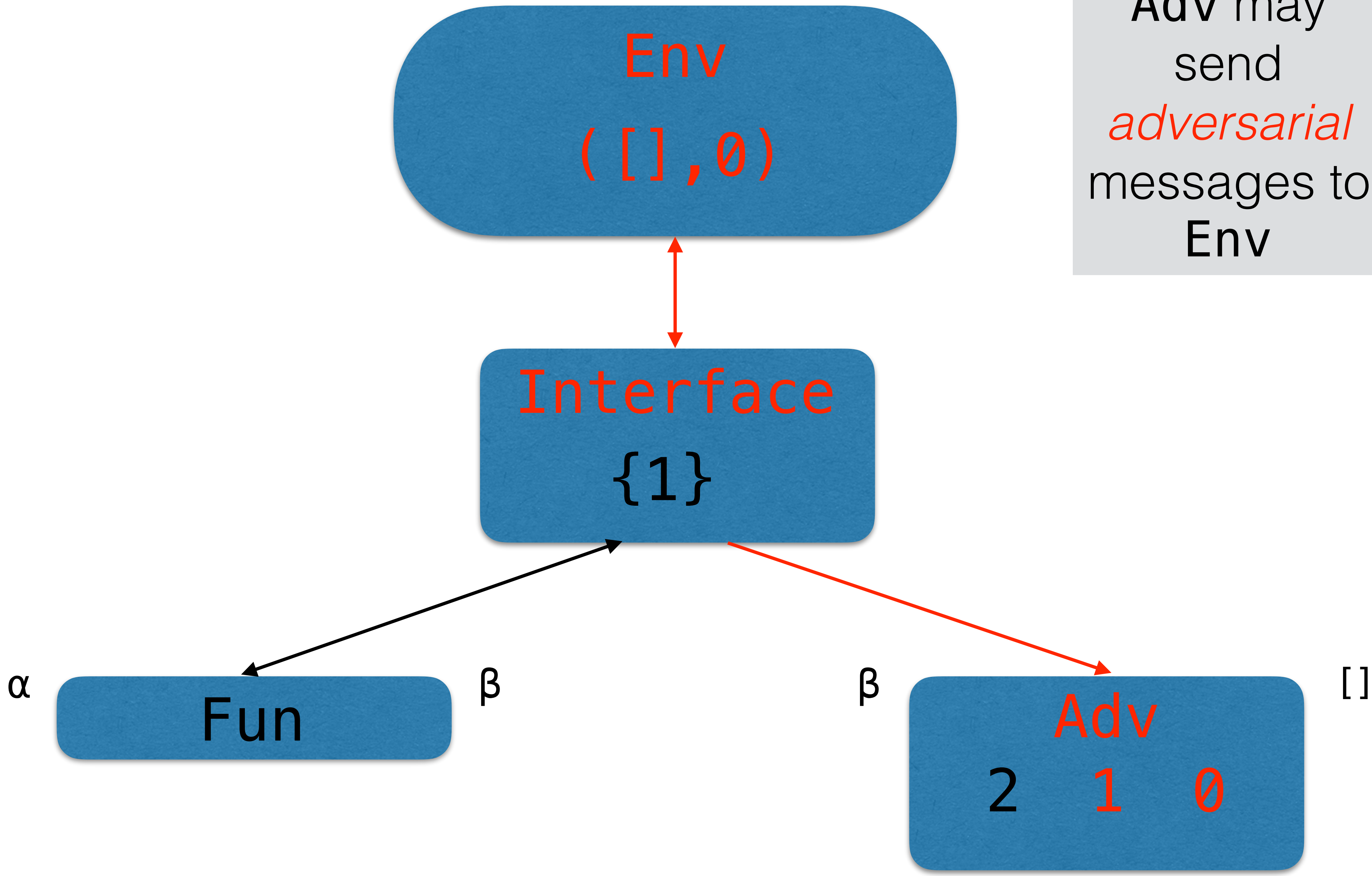
Fun may send *adversarial* messages to **Adv**

Interface Firewall



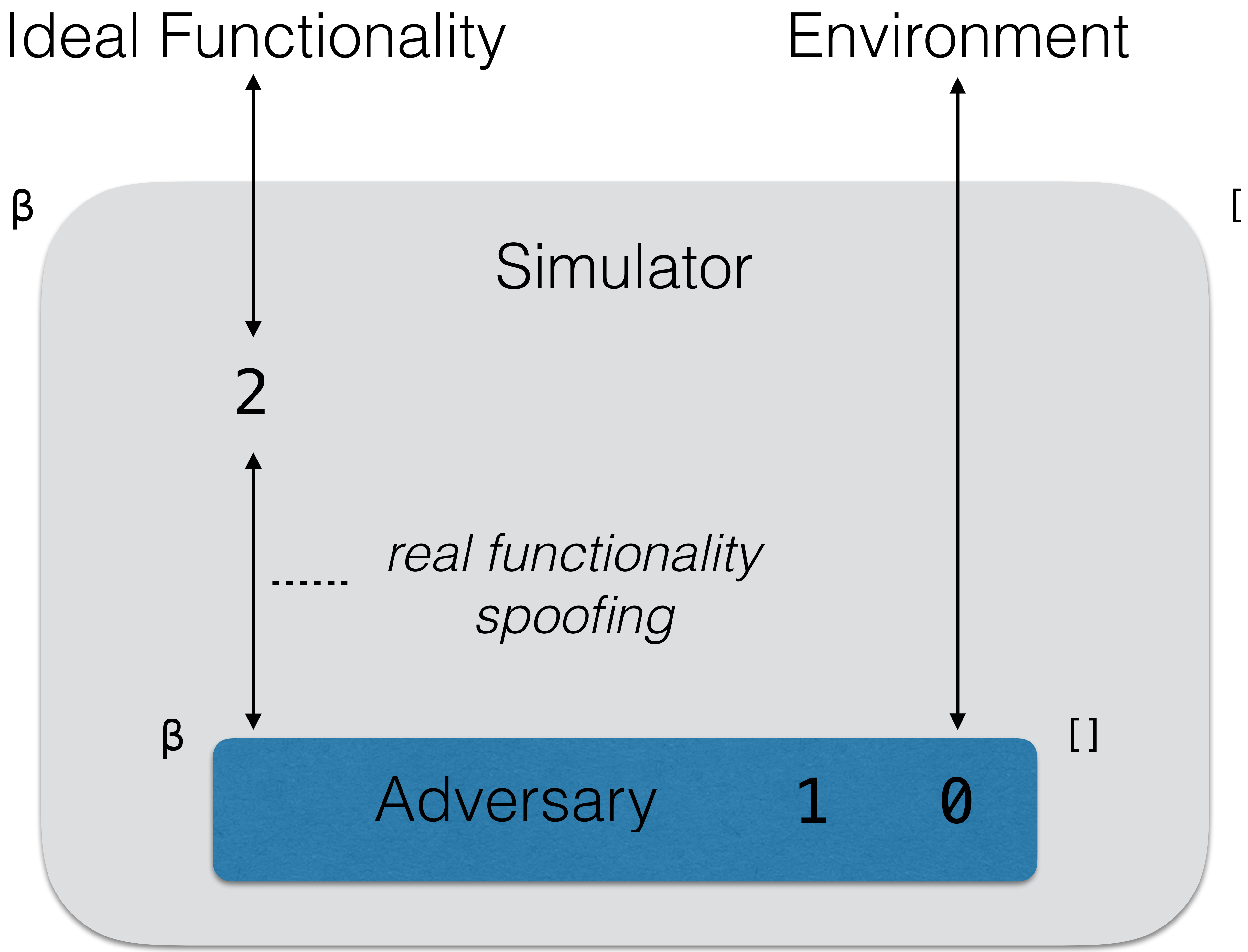
Adv may send *adversarial* messages to **Fun**

Interface Firewall

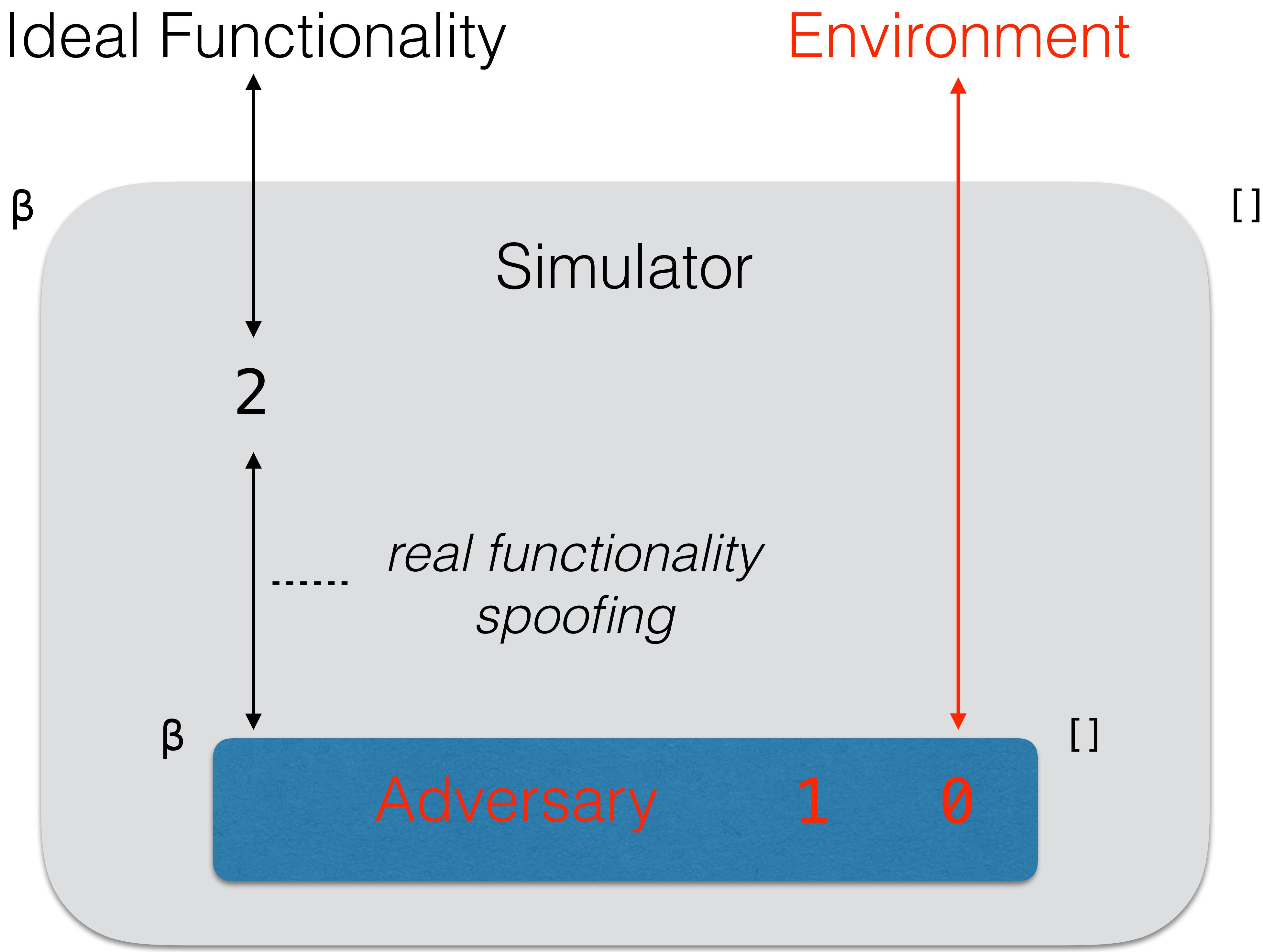


Adv may send *adversarial* messages to **Env**

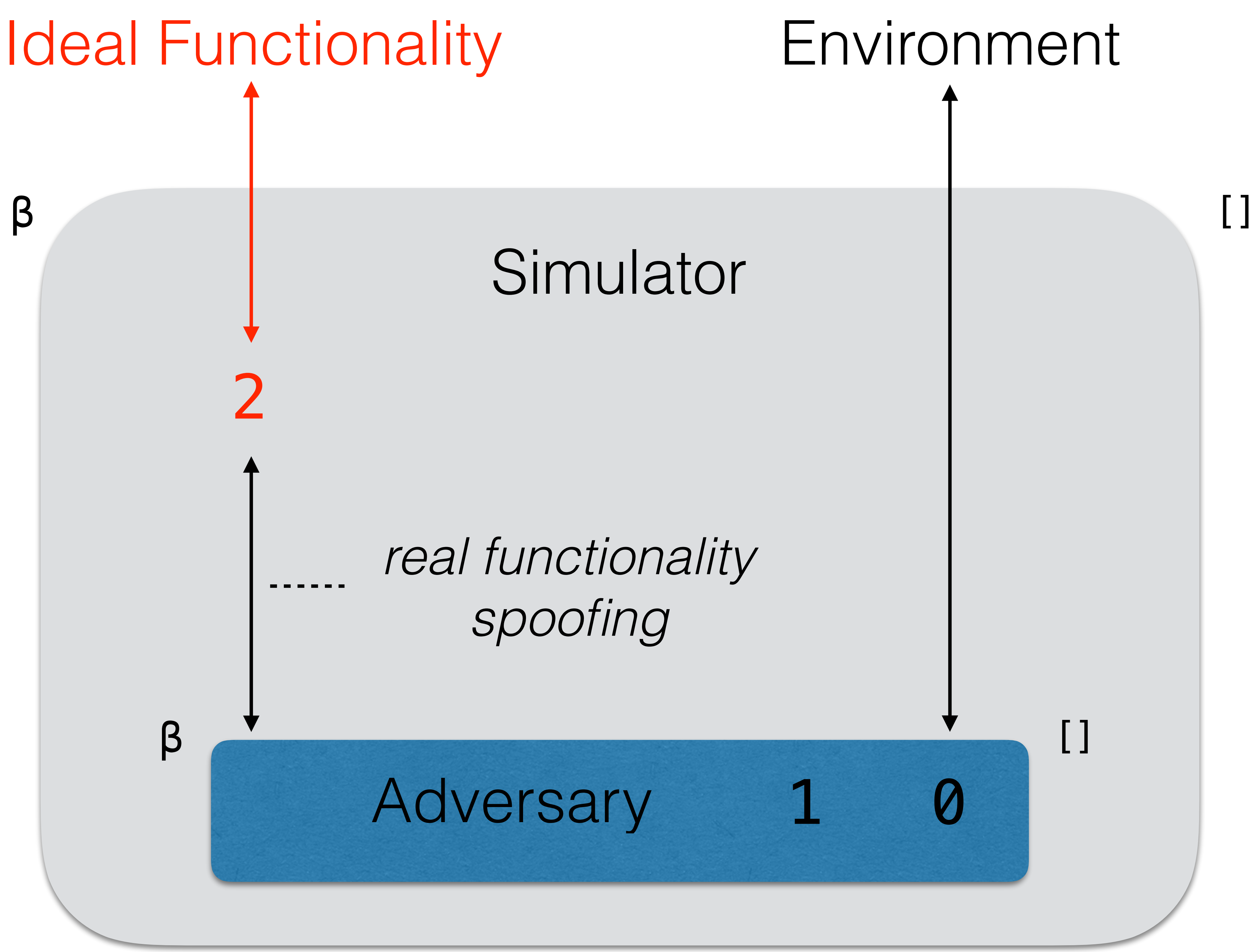
Simulators


Ideal Functionality
Environment
β
[]
Simulator
2
real functionality
spoofing
β
[]
Adversary
1
0

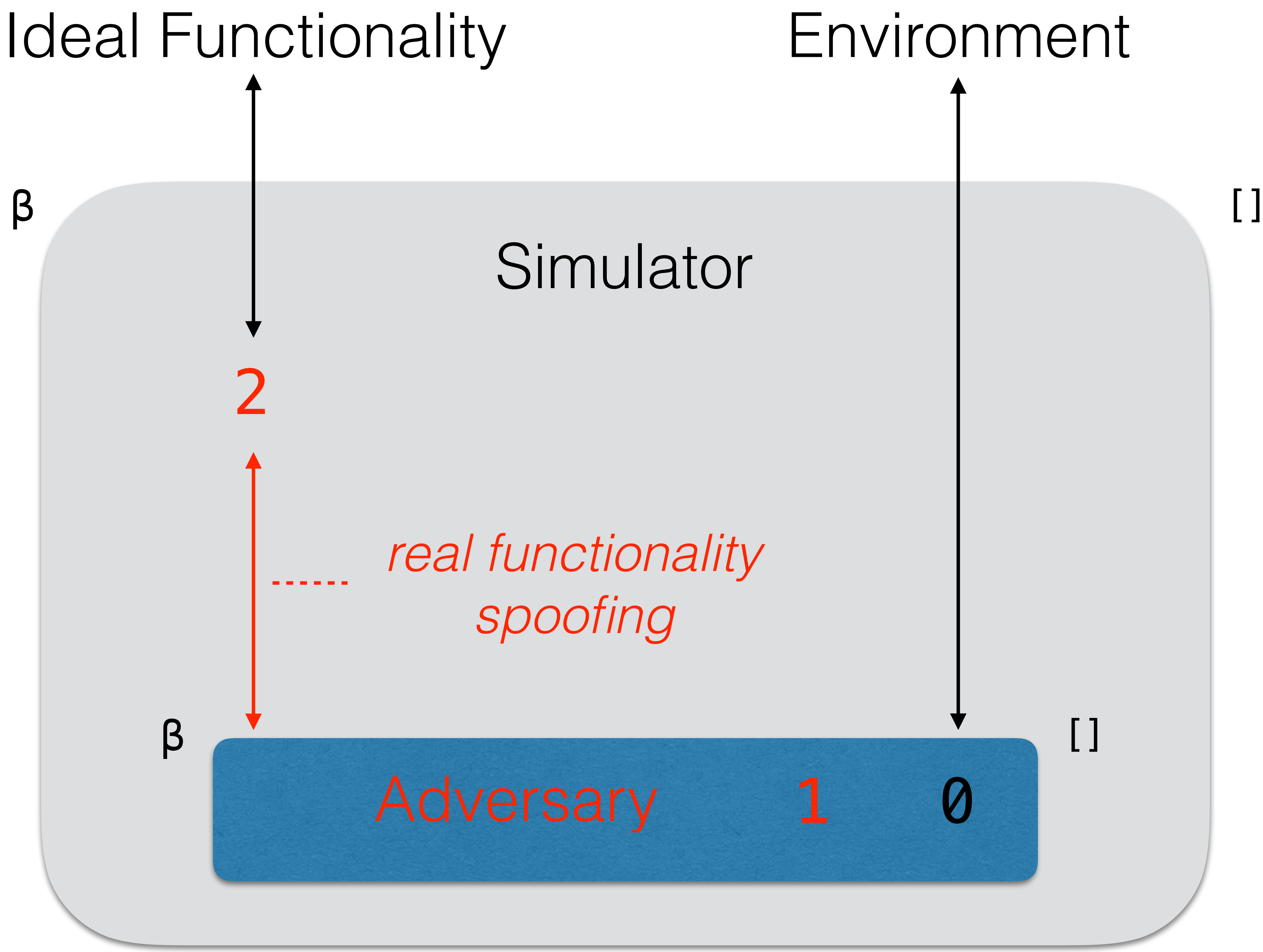
Simulators


Ideal Functionality
Environment
β
[]
Simulator
2
real functionality
spoofing
β
[]
Adversary
1
0

Simulators


Ideal Functionality
Environment
β
[]
Simulator
2
real functionality
spoofing
β
[]
Adversary 1 0

Simulators


Ideal Functionality
Environment
β
[]
Simulator
2
real functionality
spoofing
β
[]
Adversary
1
0

Secure Message Communication

As a case study, we proved the security of secure message communication in a UC style, via a one-time pad agreed by the parties using Diffie-Hellman key exchange

Group **`key`** of Keys
(**`^^`**, **`id`**, **`inv`**, **`g`**)

k ^ q : key

Commutative Semi-group
`exp` of Exponents
(∗)

inj

proj (partial)

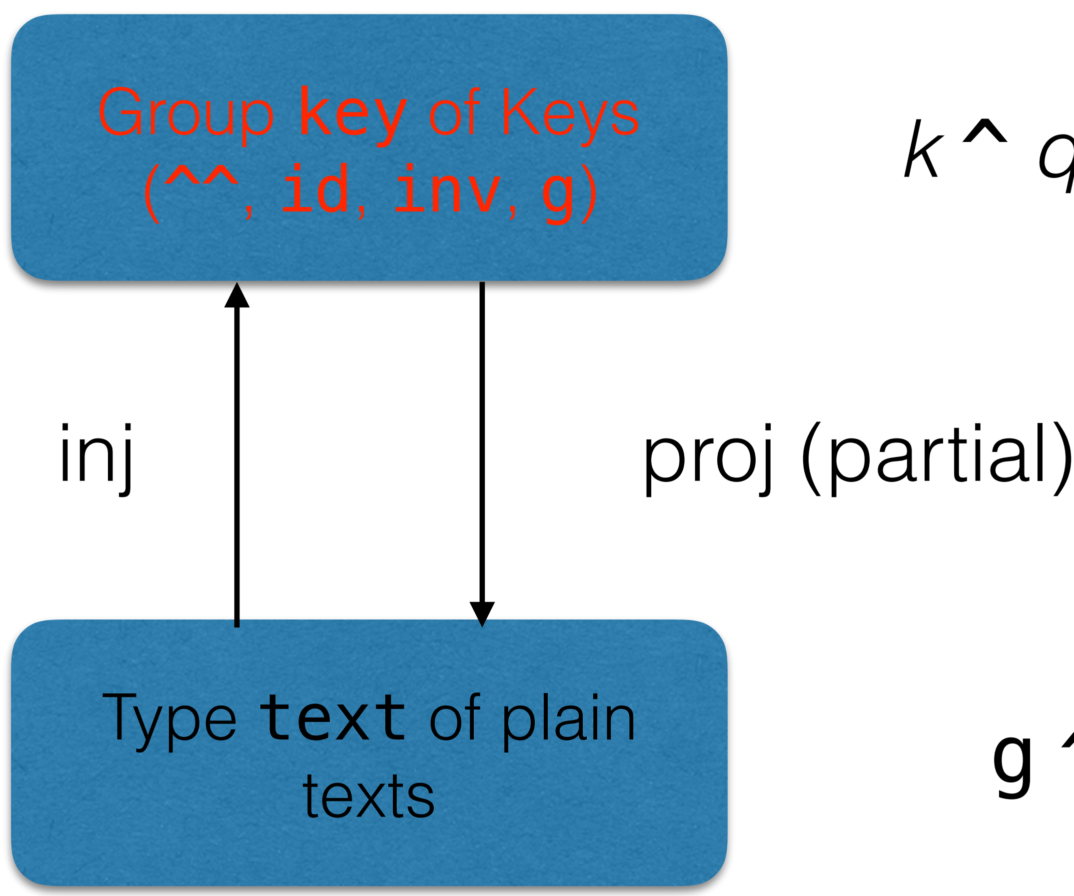
Type **`text`** of plain texts

`g` ^ q determines all keys, uniquely

`(` k **`^`**q**`)^`**r **`=`** k **`^(`** q∗r **`)`**

Secure Message Communication

As a case study, we proved the security of secure message communication in a UC style, via a one-time pad agreed by the parties using Diffie-Hellman key exchange



k ^ q : key

Commutative Semi-group
exp of Exponents
(∗)

g ^ q determines all keys, uniquely

(k ^ q)^ r = k ^(q ∗ r)

Secure Message Communication

As a case study, we proved the security of secure message communication in a UC style, via a one-time pad agreed by the parties using Diffie-Hellman key exchange

Group **`key`** of Keys
(**`^^`**, **`id`**, **`inv`**, **`g`**)

inj

proj (partial)

Type **`text`** of plain texts

k **^** q : key

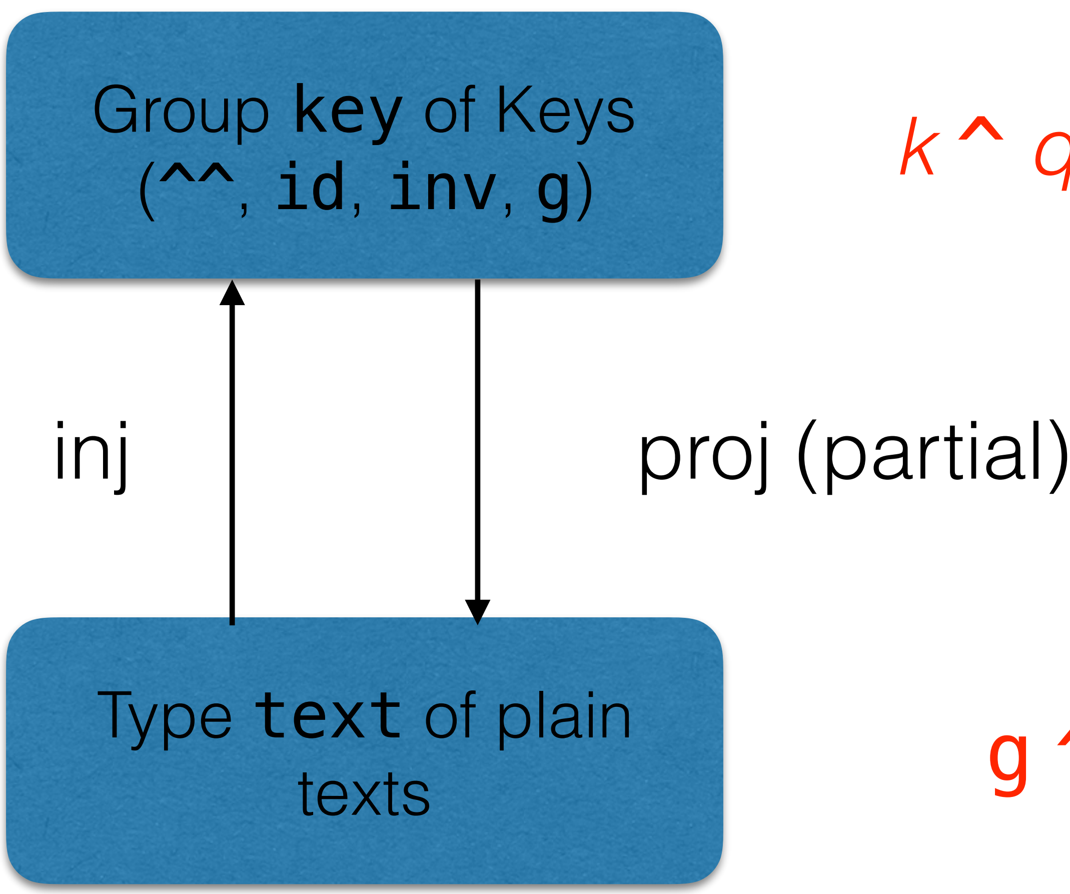
Commutative Semi-group
`exp` of Exponents
($*$)

`g` **^** q determines all keys, uniquely

`(`k **`^`**q**`)^`**r **`=`** k **`^(`**$q$$*$$r$**`)`**

Secure Message Communication

As a case study, we proved the security of secure message communication in a UC style, via a one-time pad agreed by the parties using Diffie-Hellman key exchange



k ^ *q* : key

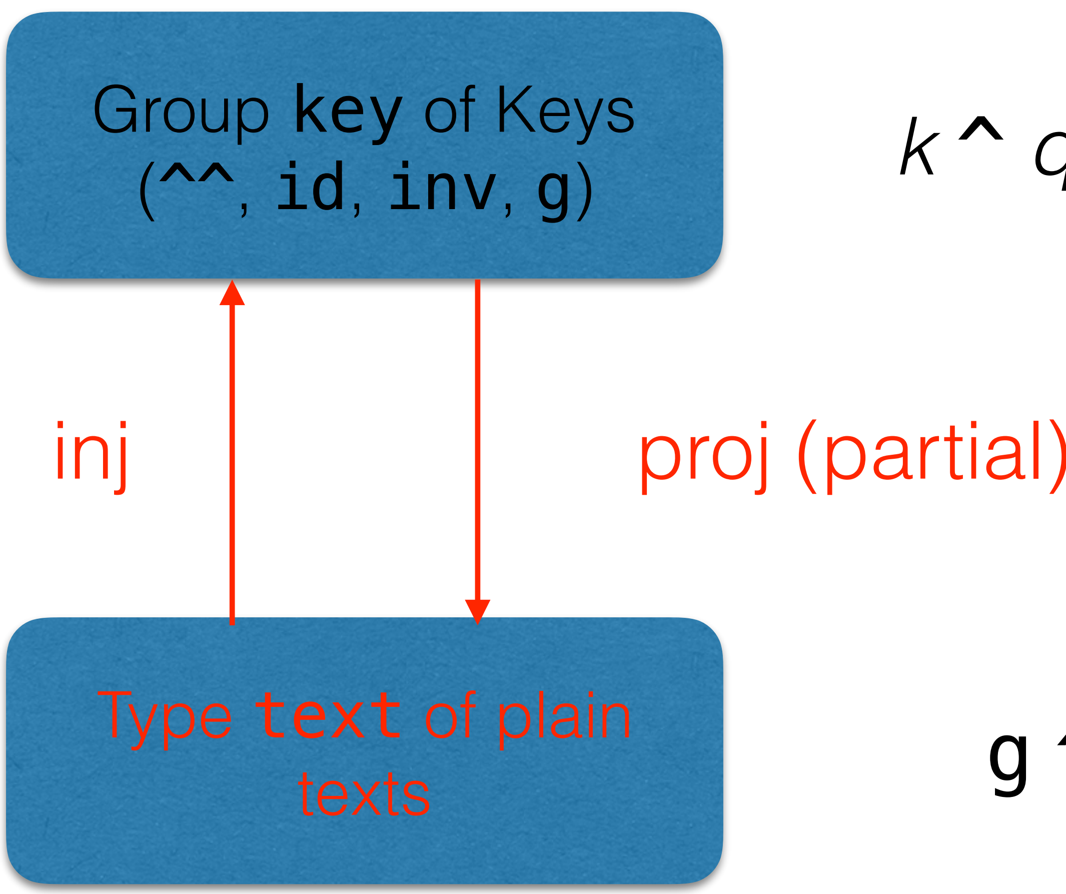
Commutative Semi-group
exp of Exponents
(∗)

g ^ *q* determines all keys, uniquely

$(k \,\texttt{\^{}}\, q) \,\texttt{\^{}}\, r \;=\; k \,\texttt{\^{}}\, (q * r)$

Secure Message Communication

As a case study, we proved the security of secure message communication in a UC style, via a one-time pad agreed by the parties using Diffie-Hellman key exchange



k ^ q : key

Commutative Semi-group
`exp` of Exponents
(∗)

`g` ^ q determines all keys, uniquely

`(`k **`^`**q**`)`**`^`r **`=`** k **`^(`** q∗r**`)`**

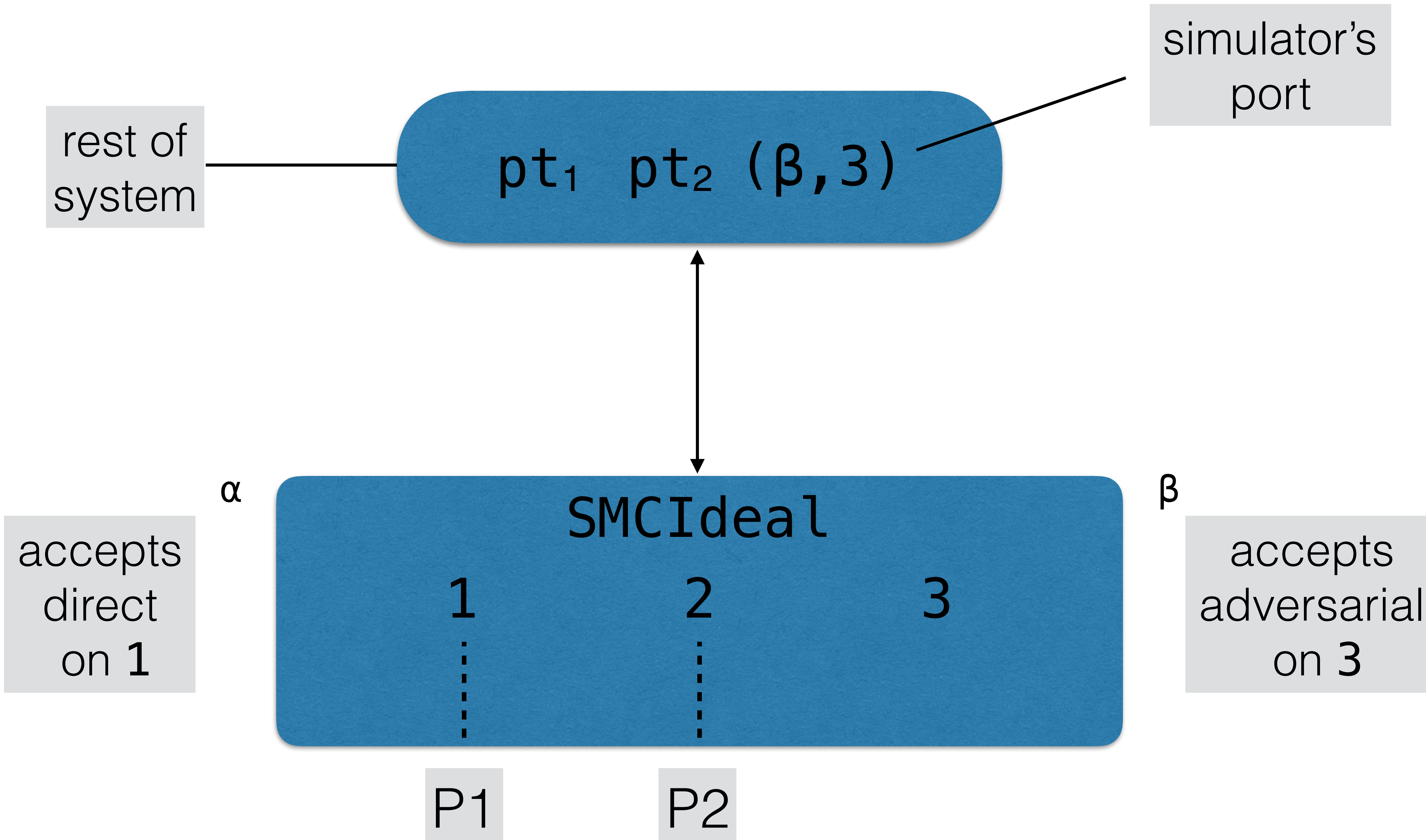
Secure Message Communication

- Two protocol parties: 1 and 2
  - P1 wants to securely transmit plain text *t* to P2
- P1 and P2 use Diffie-Hellman key exchange to agree on a key, *k* — *(see next slide)*
- P1 transmits *e* `= inj` *t* `^^` *k* to P2 — adversary observes but can't corrupt
- P2 gets decryption of *e* as `proj (`*e* `^^ inv` *k*`)`

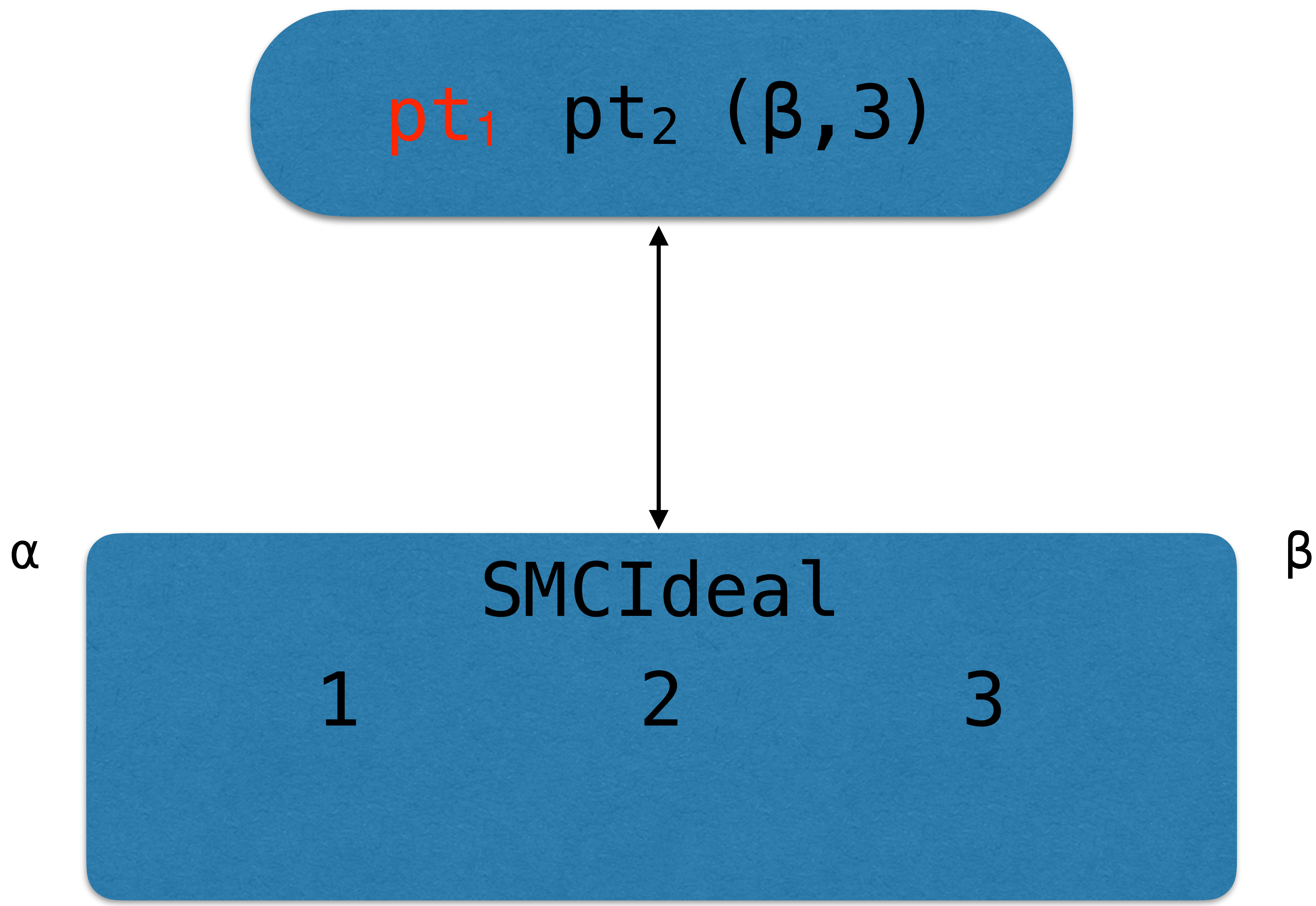
Diffie-Hellman Key Exchange

- P1 and P2 both have their own randomly generated secrets q_1, q_2 : exp
- P1 sends $\texttt{g\^{}}q_1$ to P2, which sends $\texttt{g\^{}}q_2$ to P1 — adversary observes these transmissions
- P1 then computes $(g\text{\^{}}q_2)\text{\^{}}q_1 = g\text{\^{}}(q_2 * q_1) = g\text{\^{}}(q_1 * q_2)$ as the shared key, k
- P2 then computes $(g\text{\^{}}q_1)\text{\^{}}q_2 = g\text{\^{}}(q_1 * q_2)$ as the shared key, k

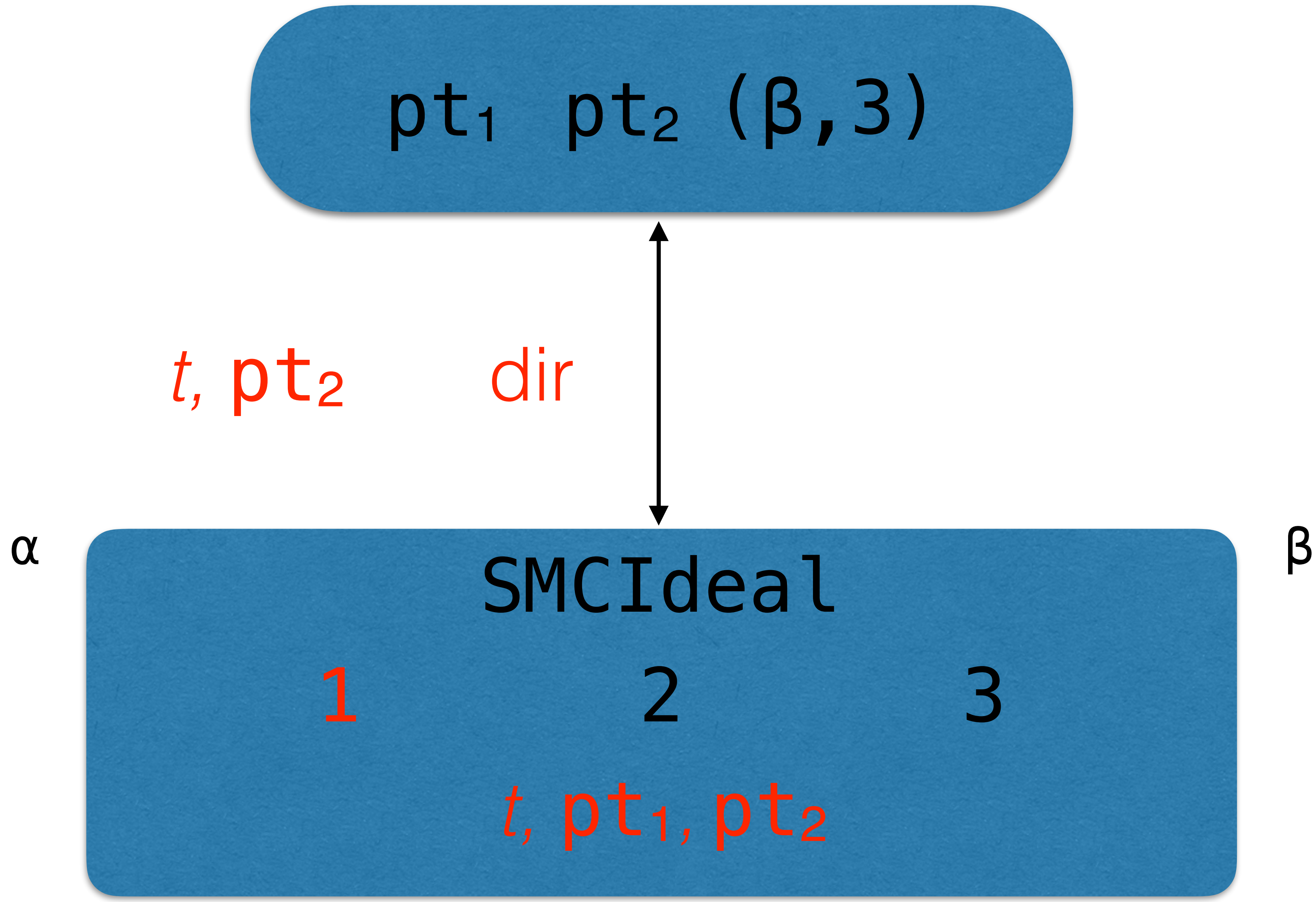
SMC Ideal Functionality



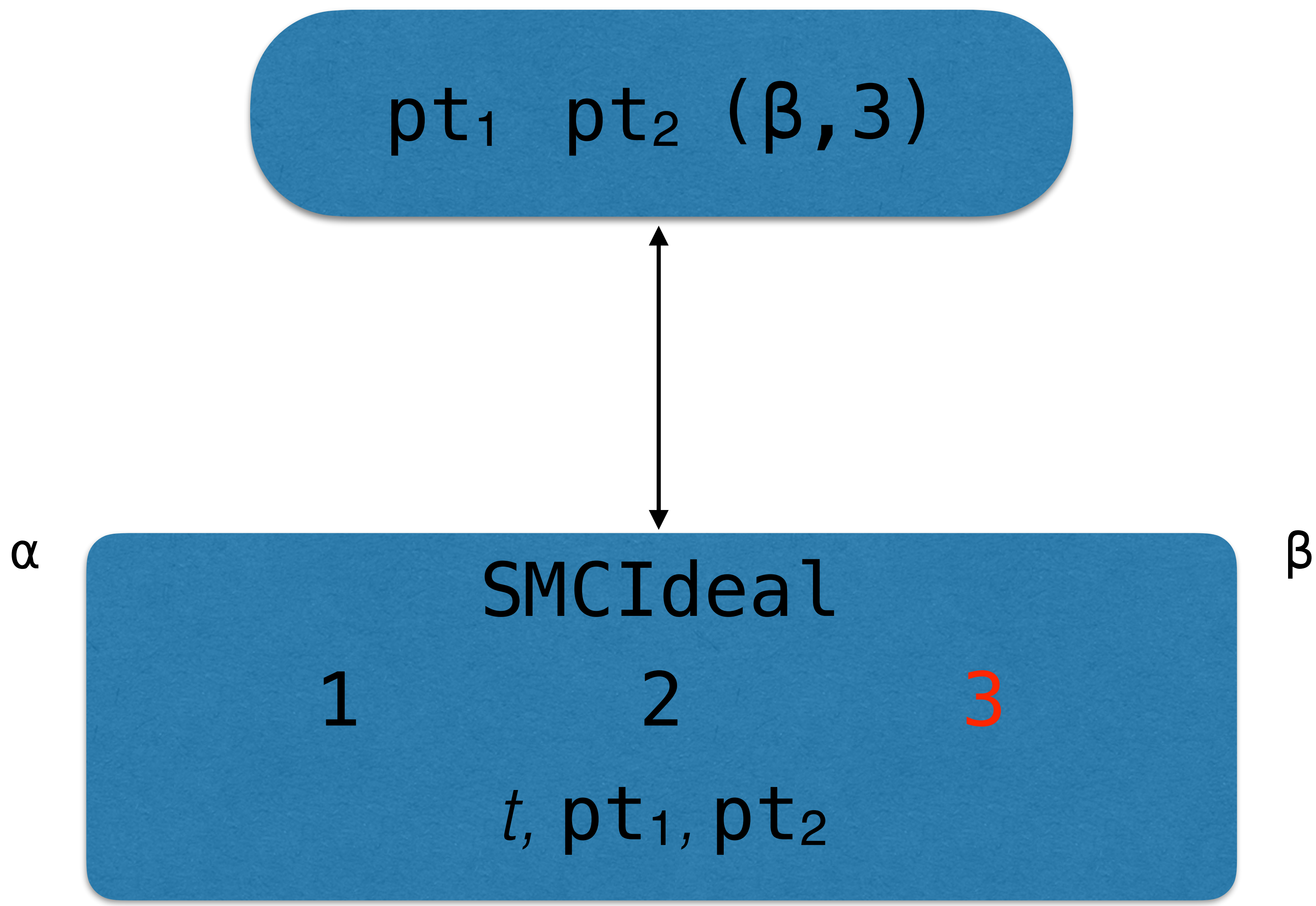
SMC Ideal Functionality



SMC Ideal Functionality

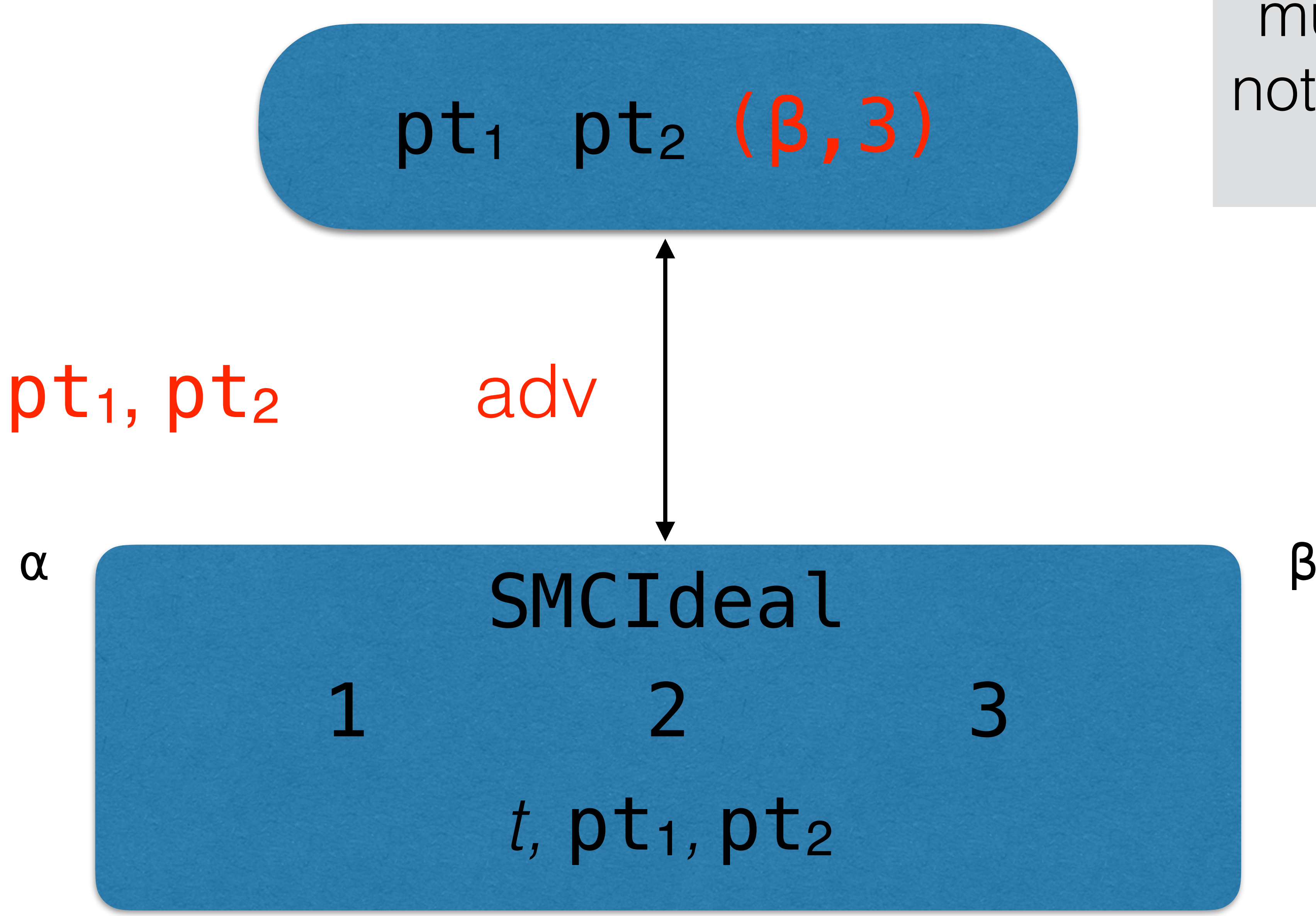


SMC Ideal Functionality

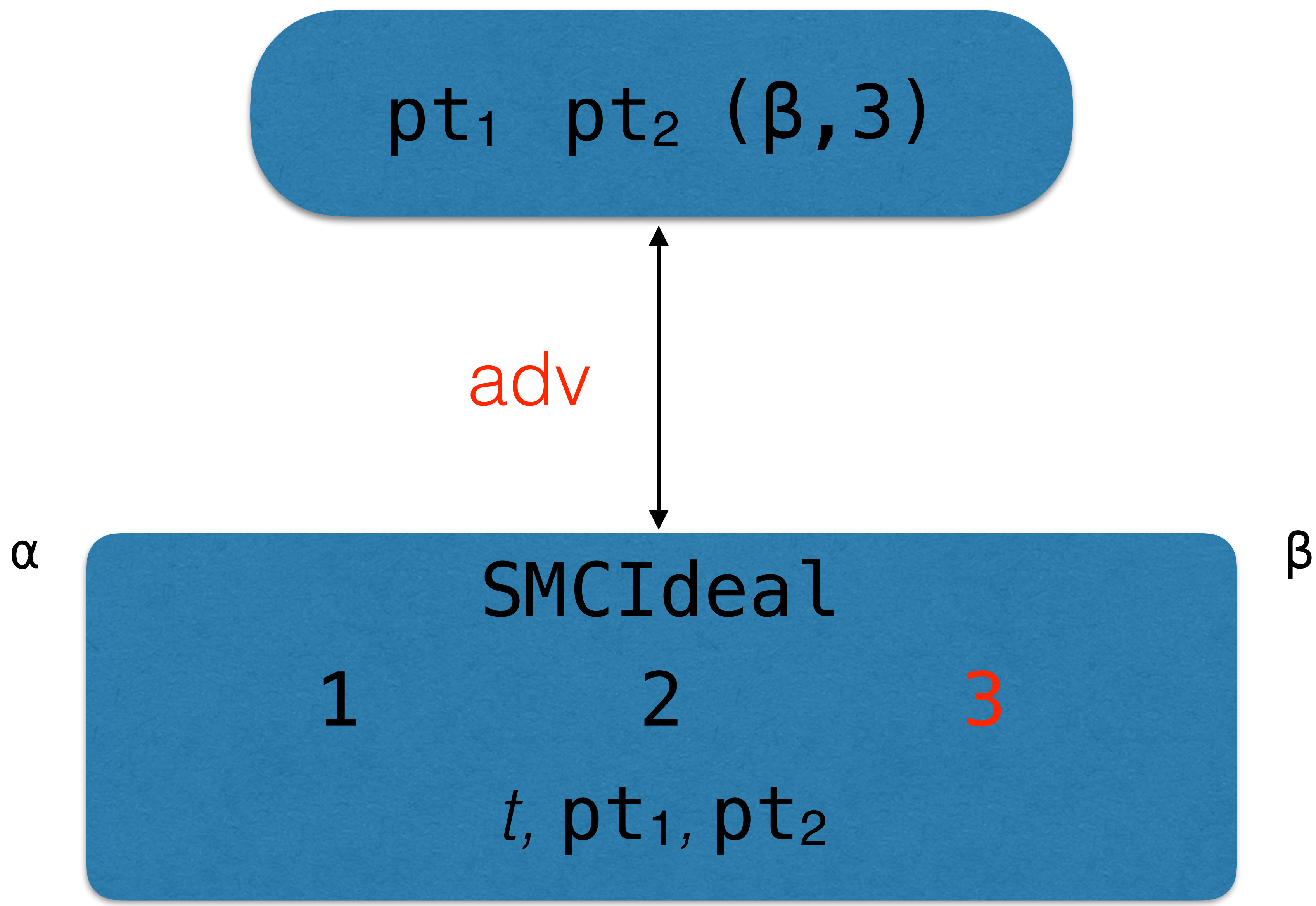


SMC Ideal Functionality

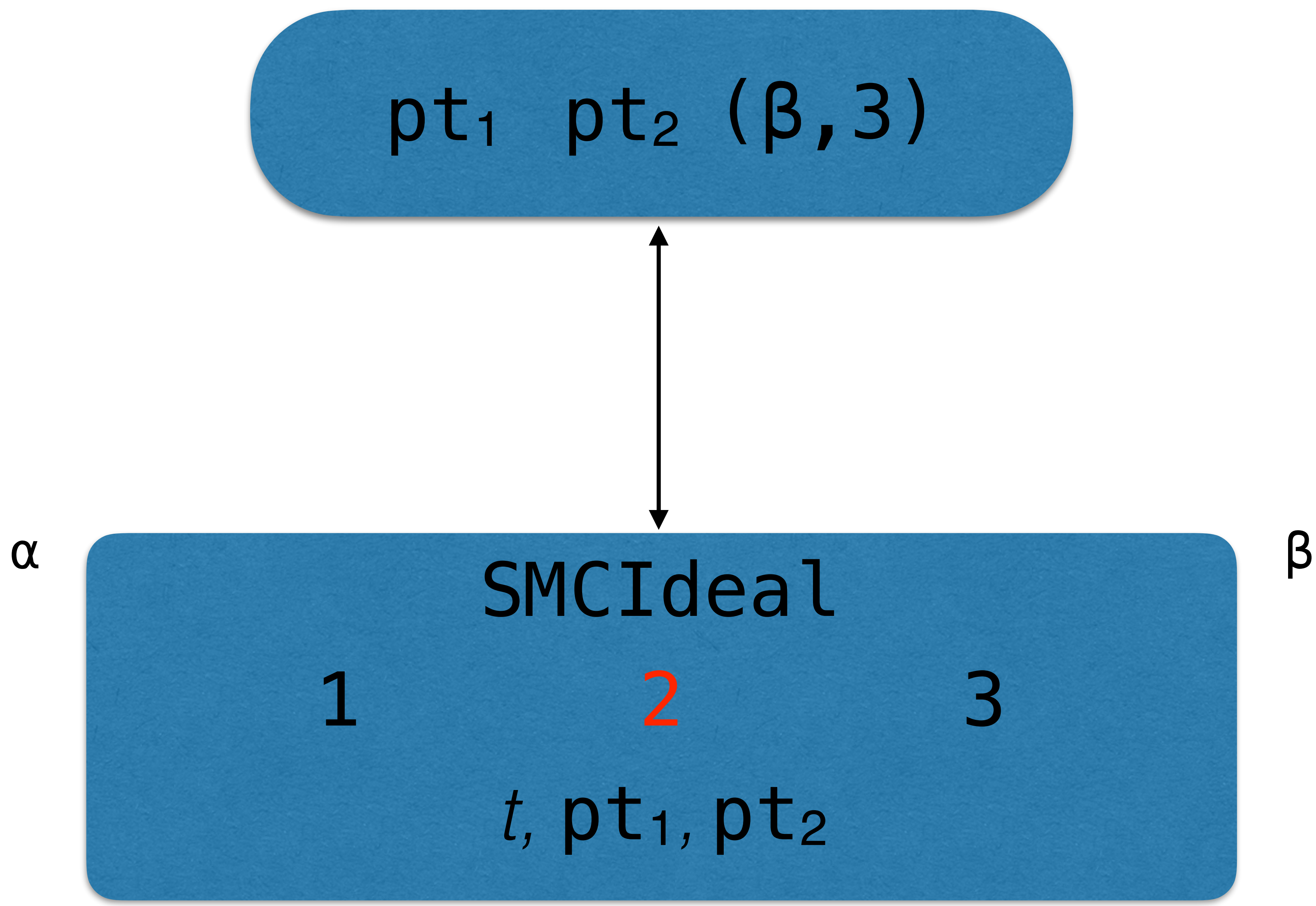
simulator must work not knowing *t*



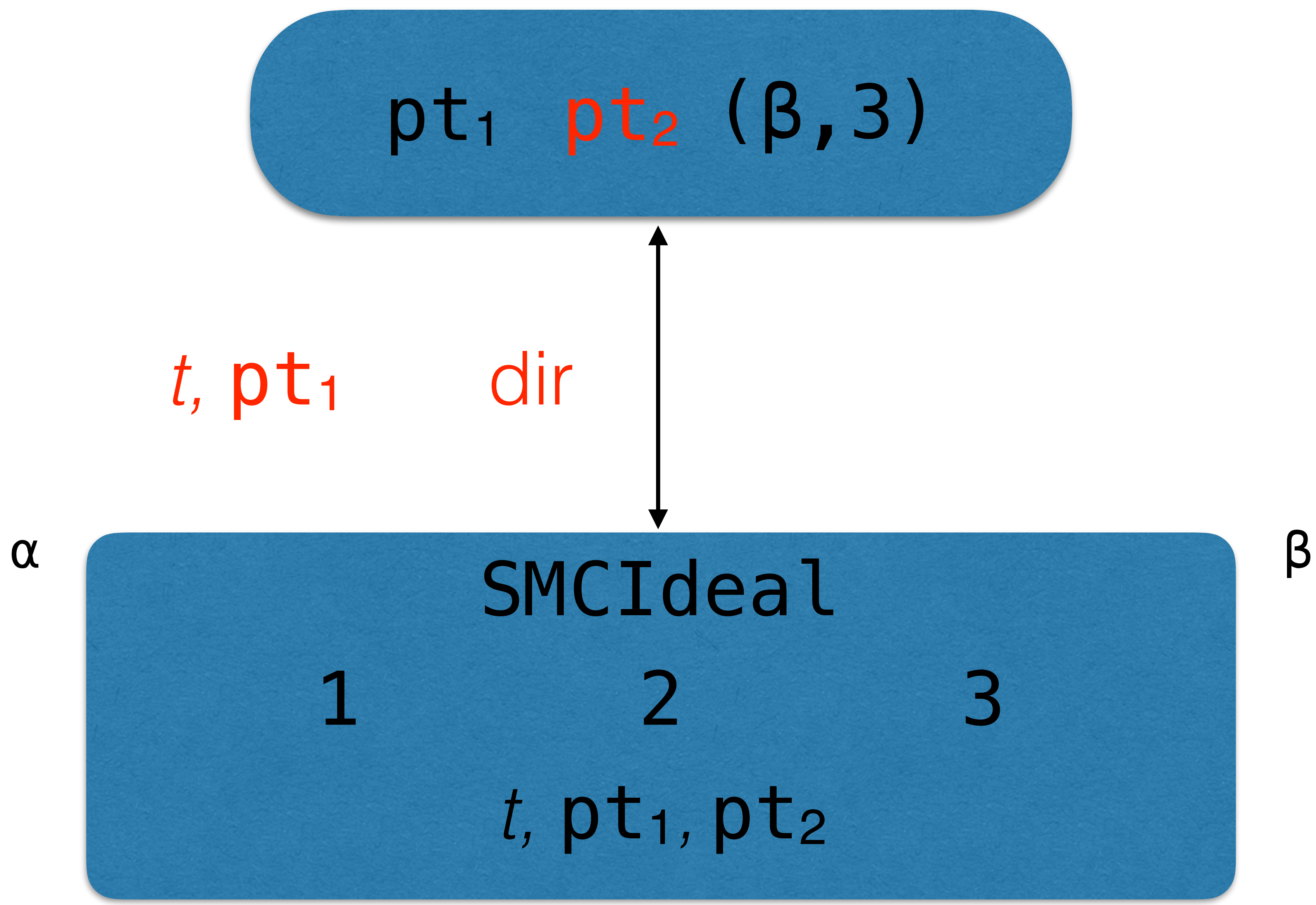
SMC Ideal Functionality



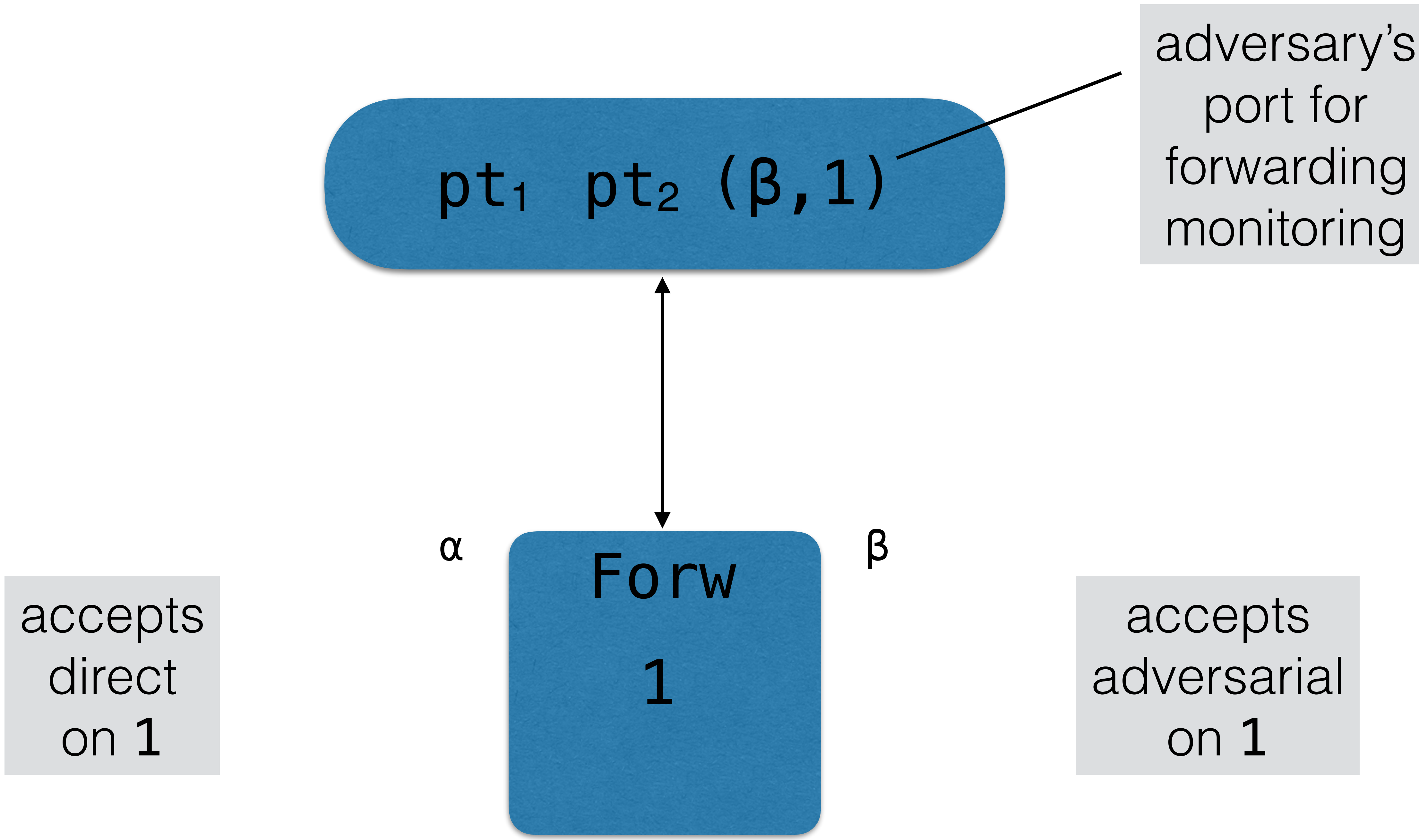
SMC Ideal Functionality



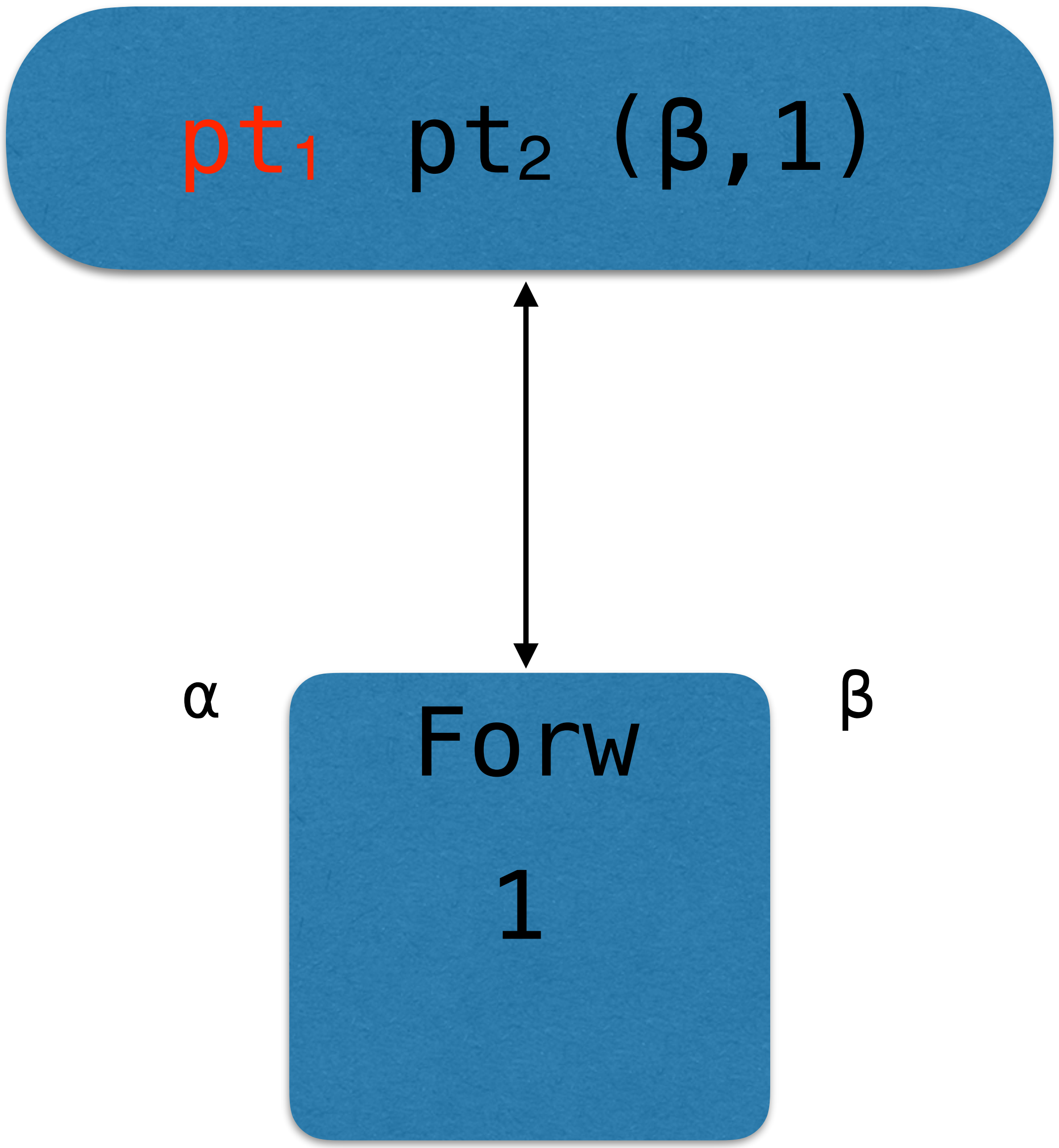
SMC Ideal Functionality



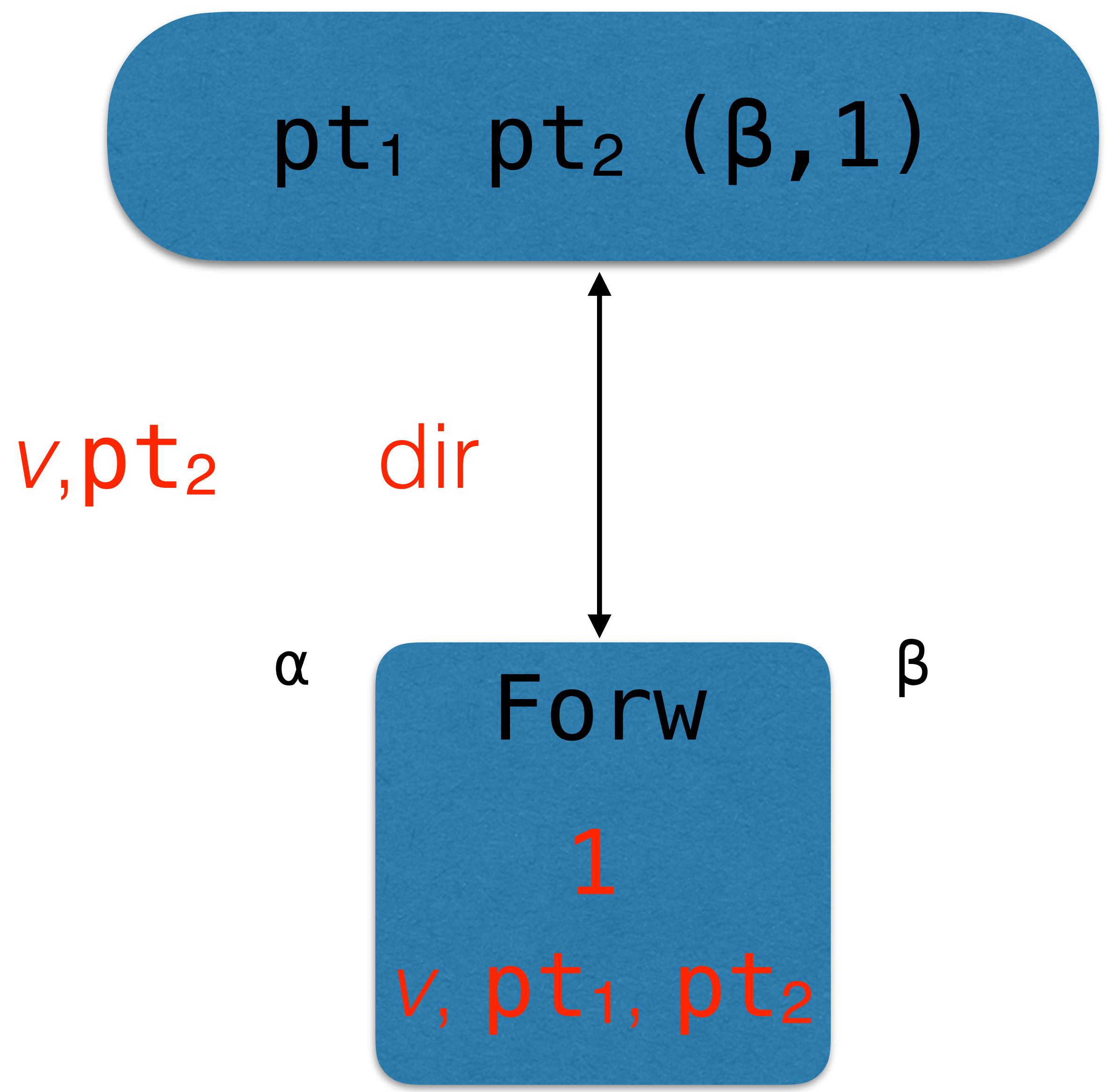
Ideal Forwarding Functionality



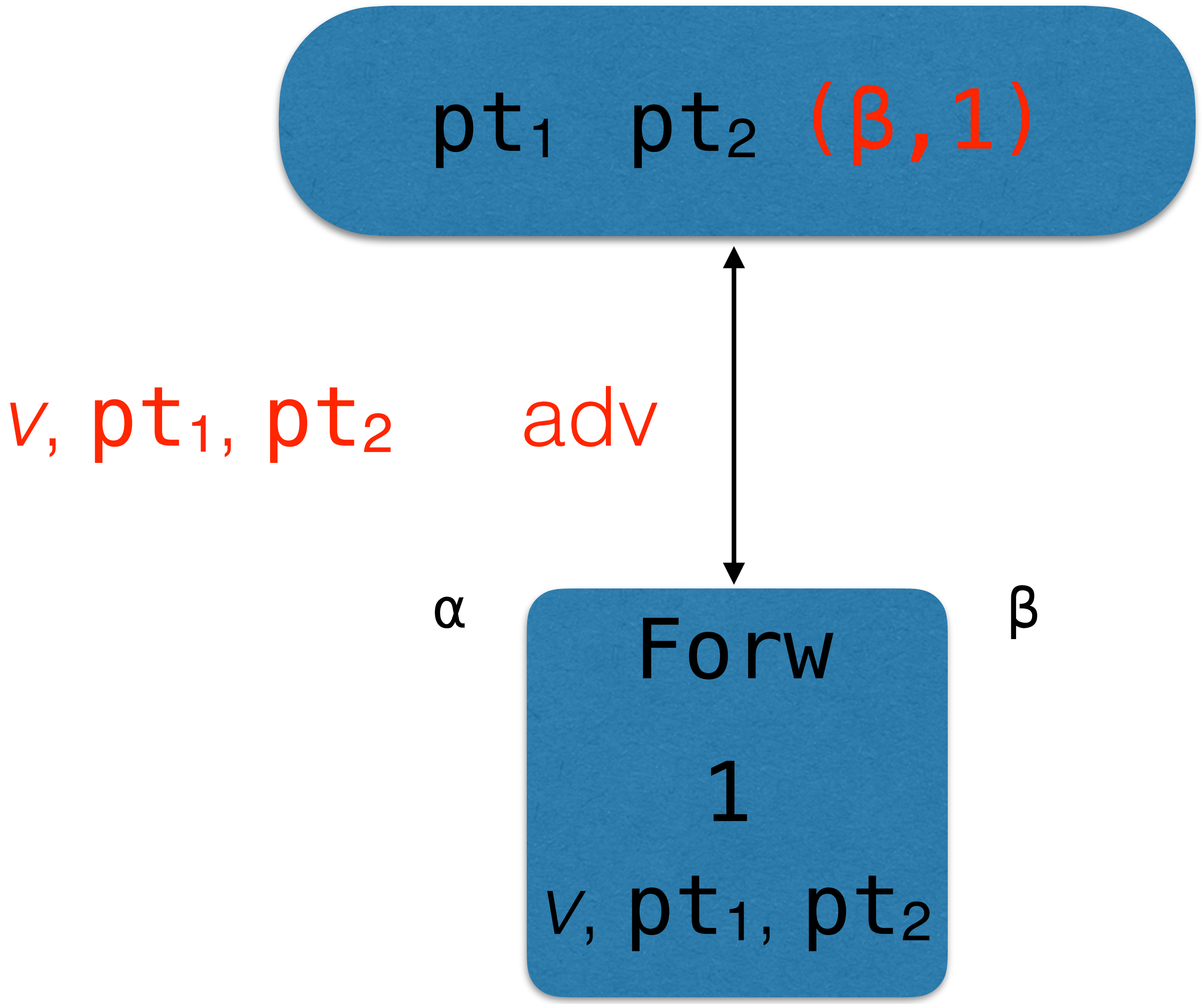
Forwarding Functionality



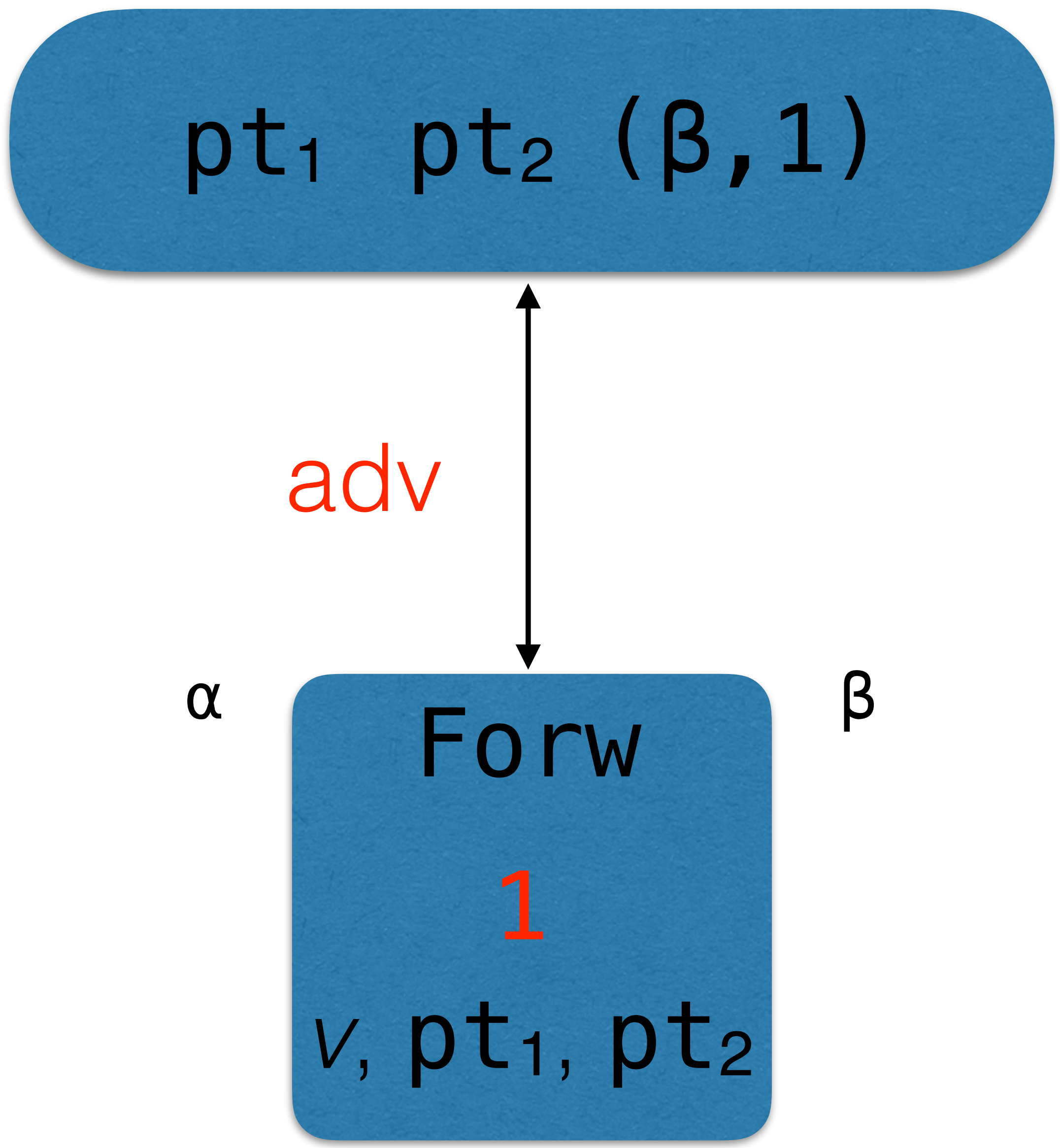
Forwarding Functionality



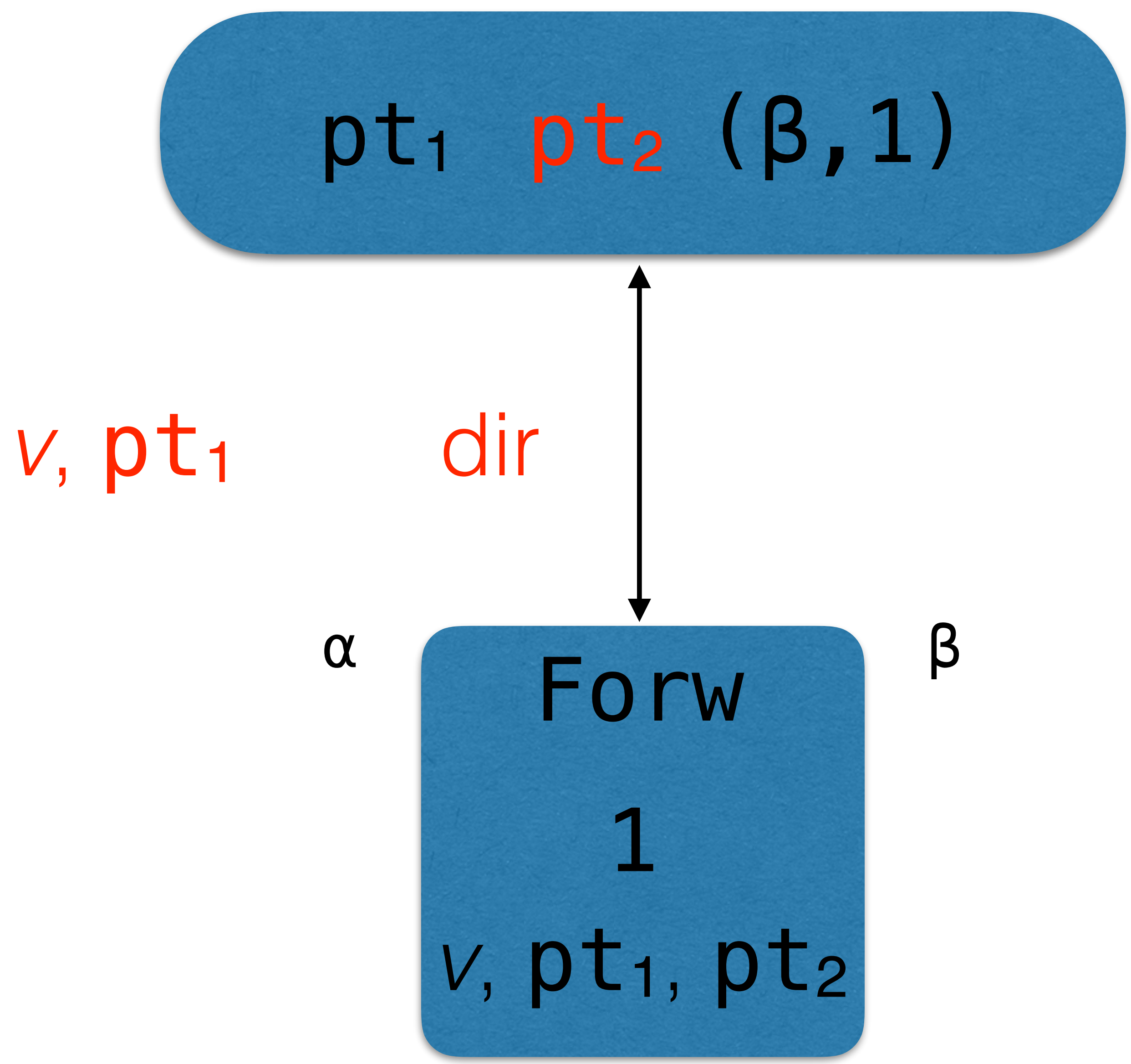
Forwarding Functionality



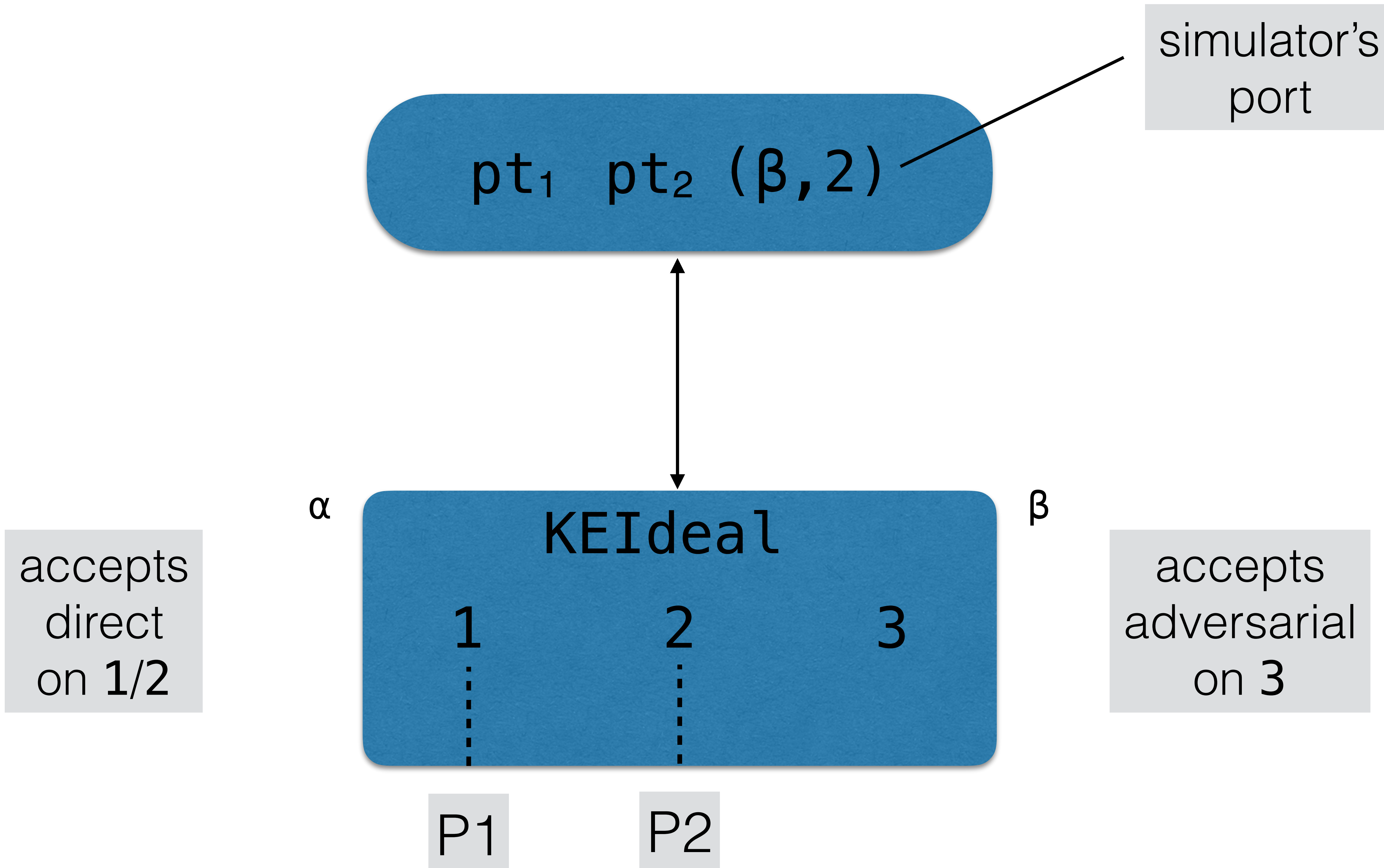
Forwarding Functionality



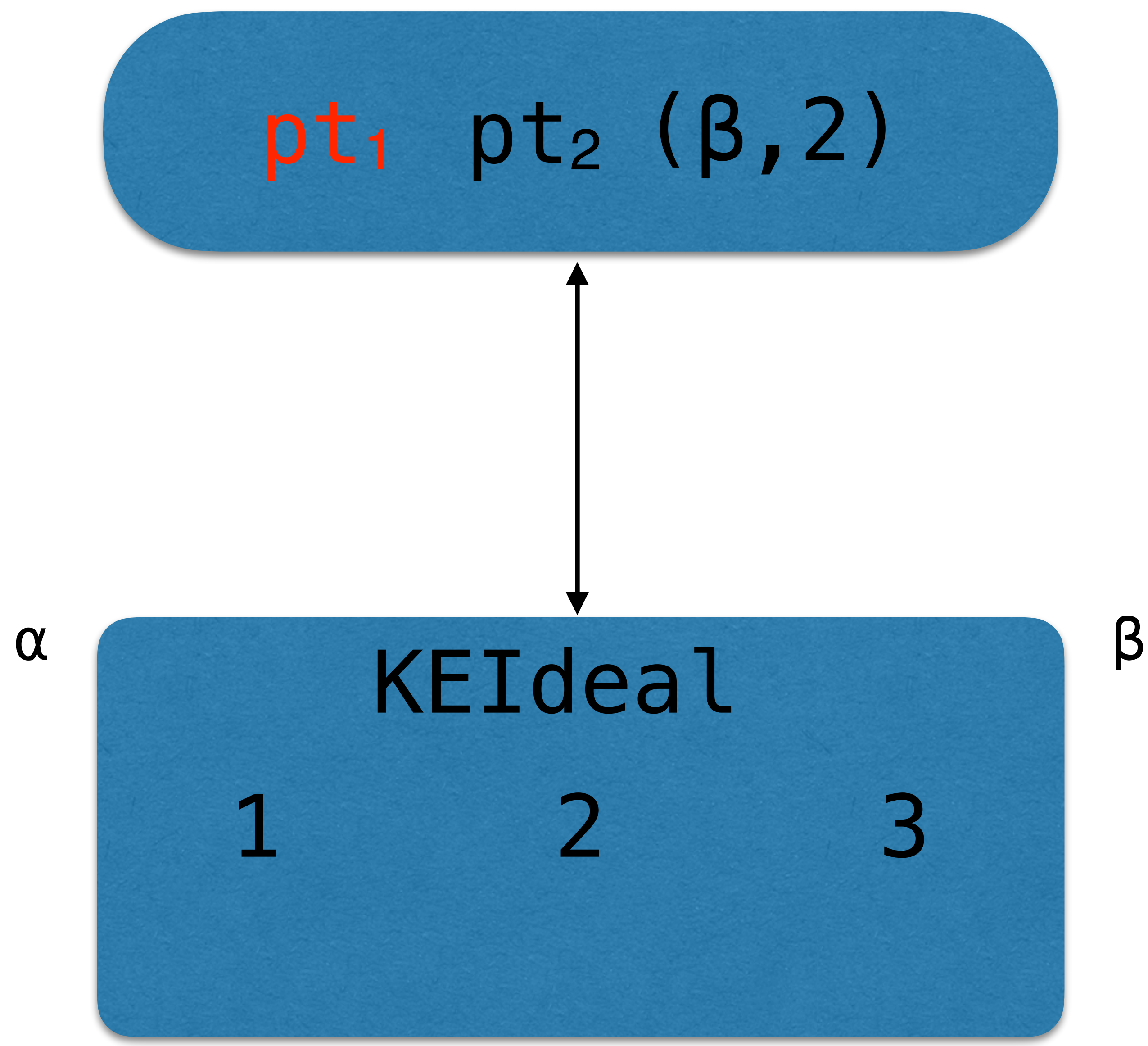
Forwarding Functionality



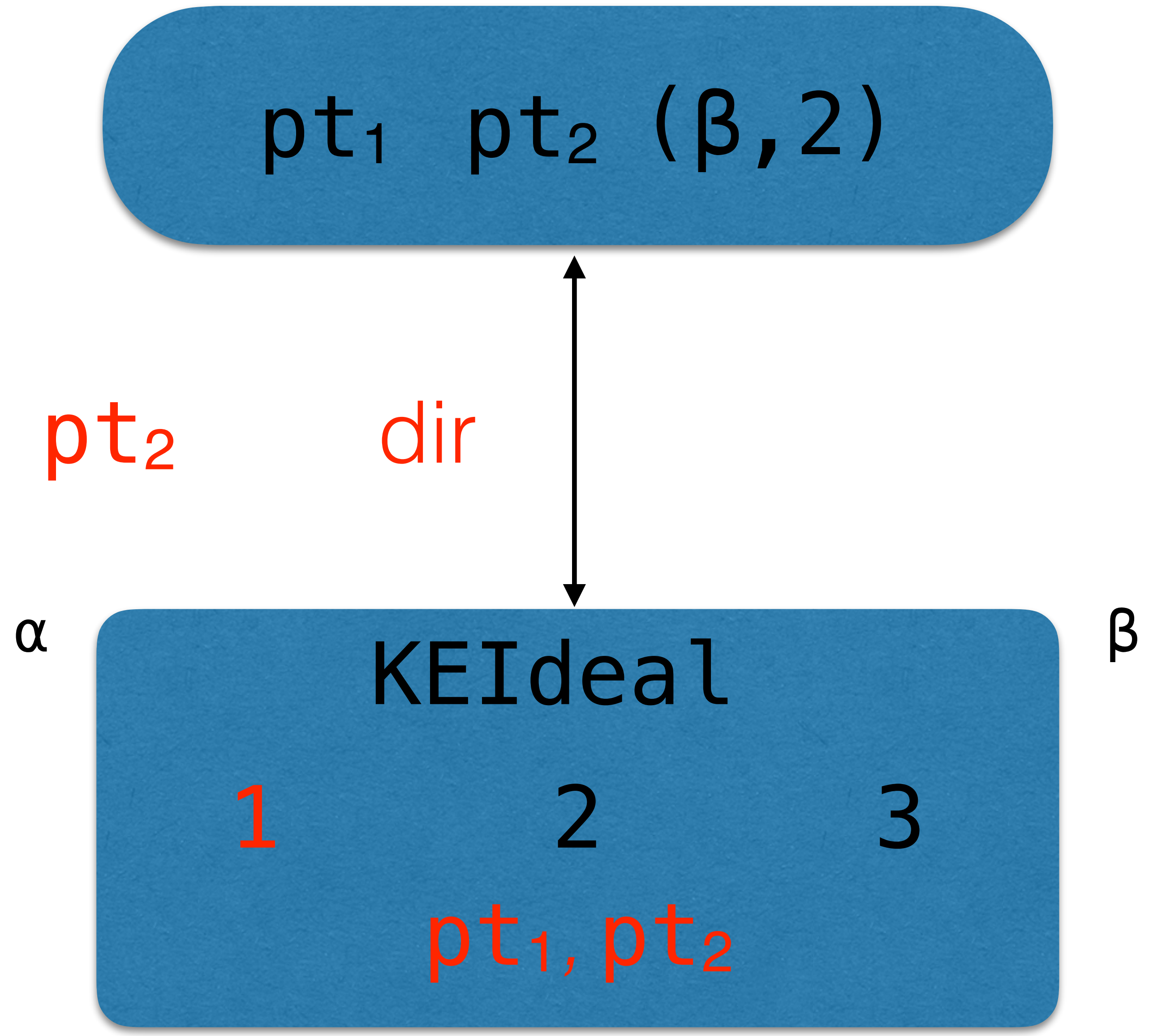
Key Exchange Ideal Functionality



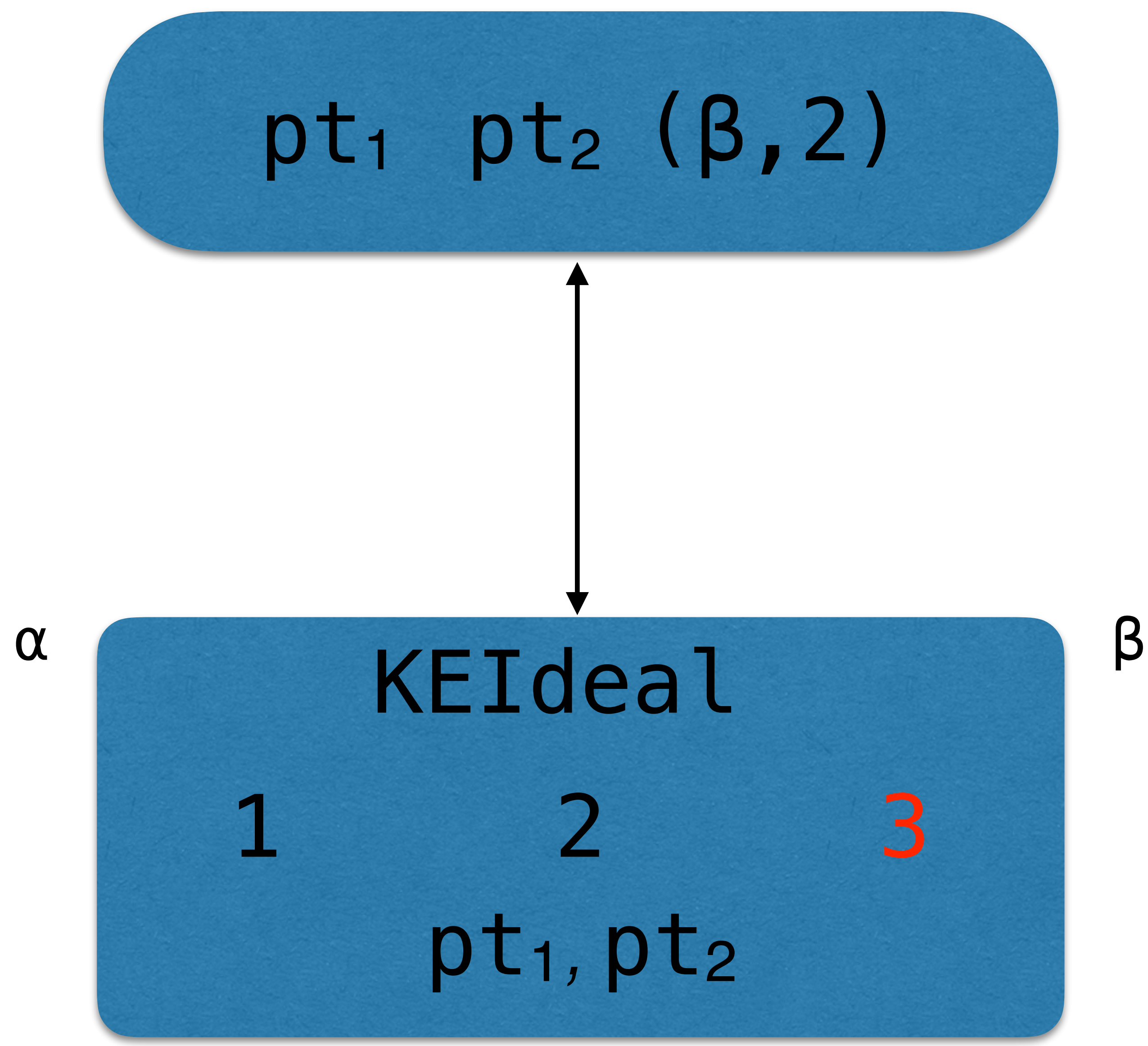
Key Exchange Ideal Functionality



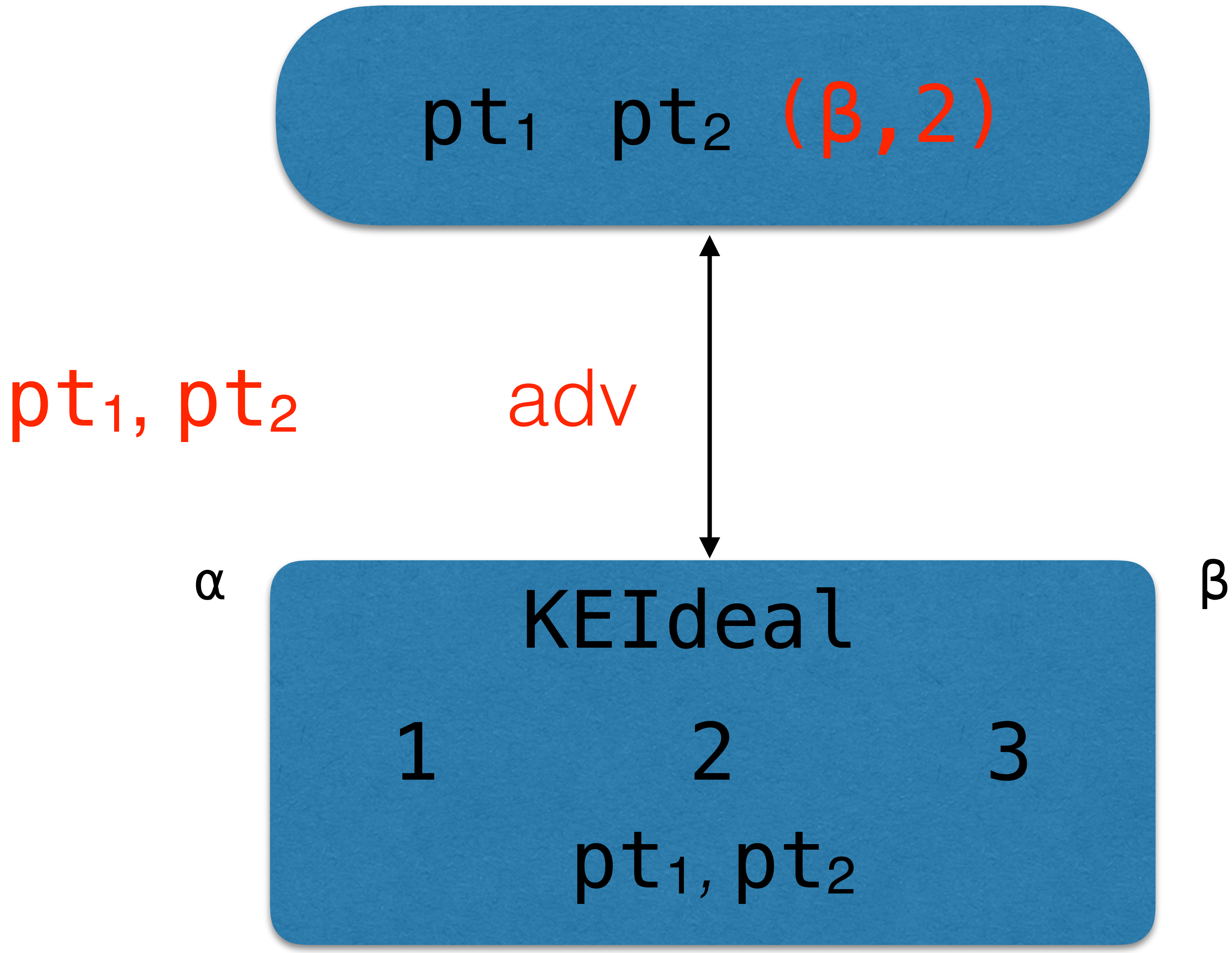
Key Exchange Ideal Functionality



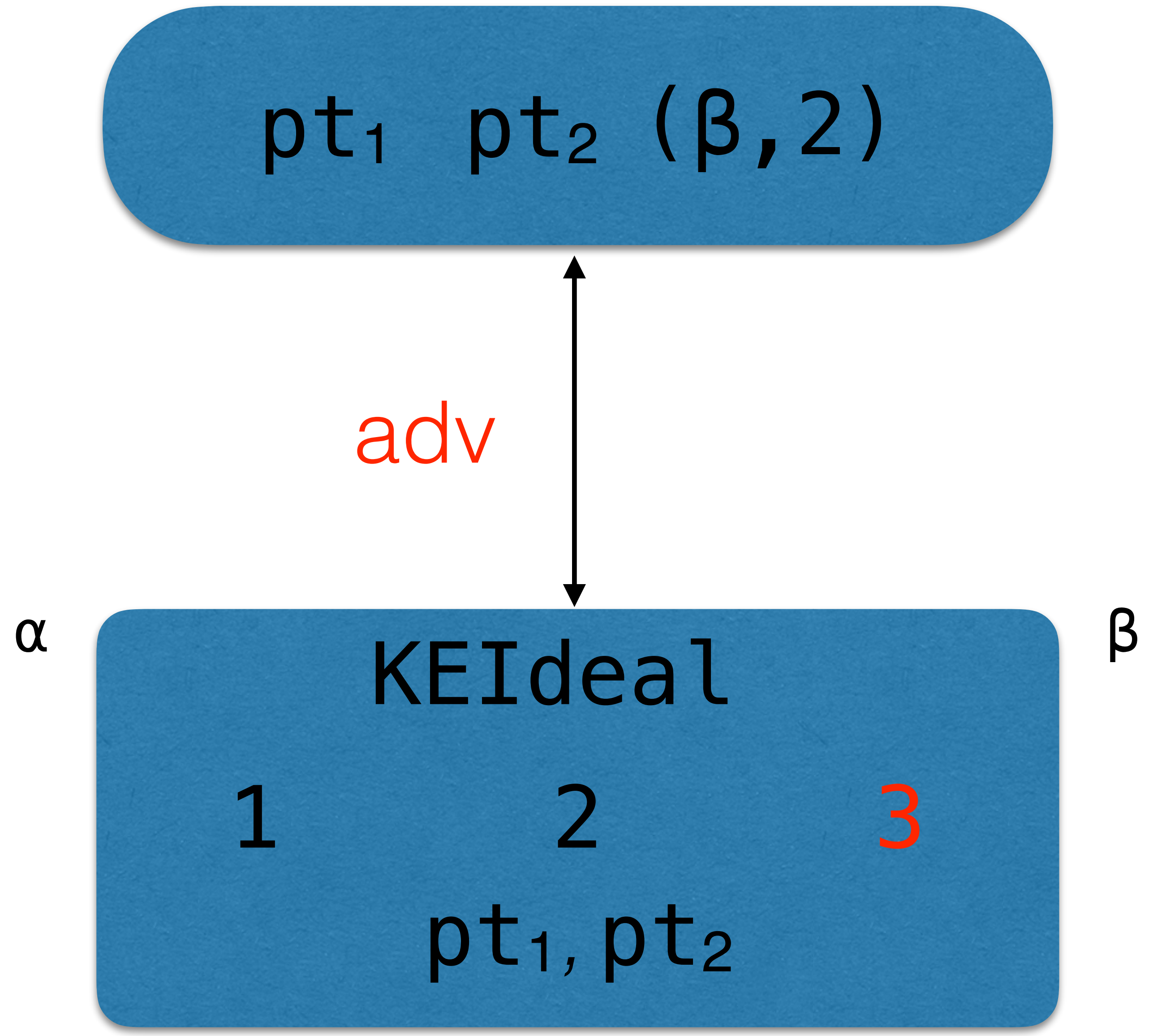
Key Exchange Ideal Functionality



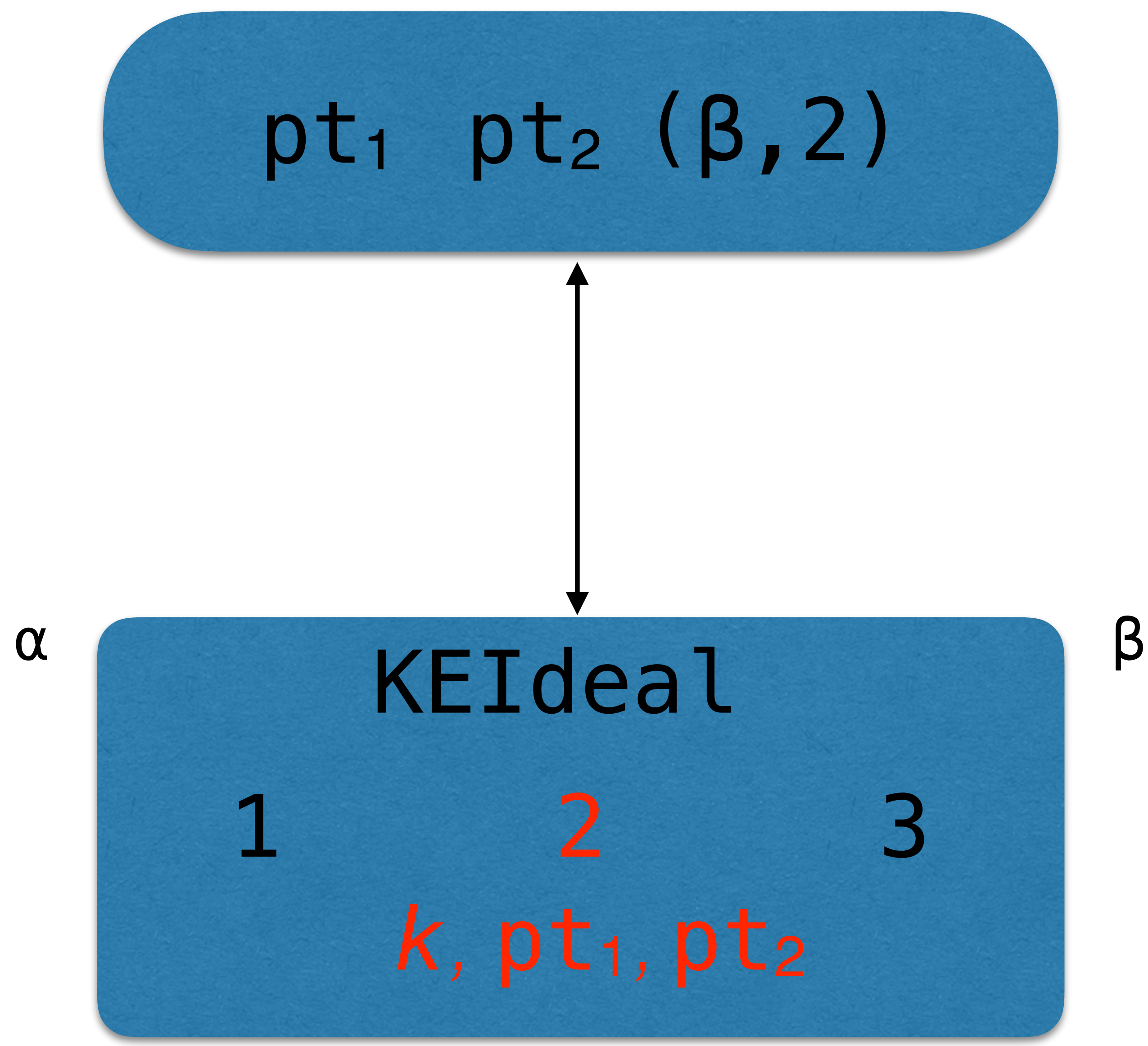
Key Exchange Ideal Functionality



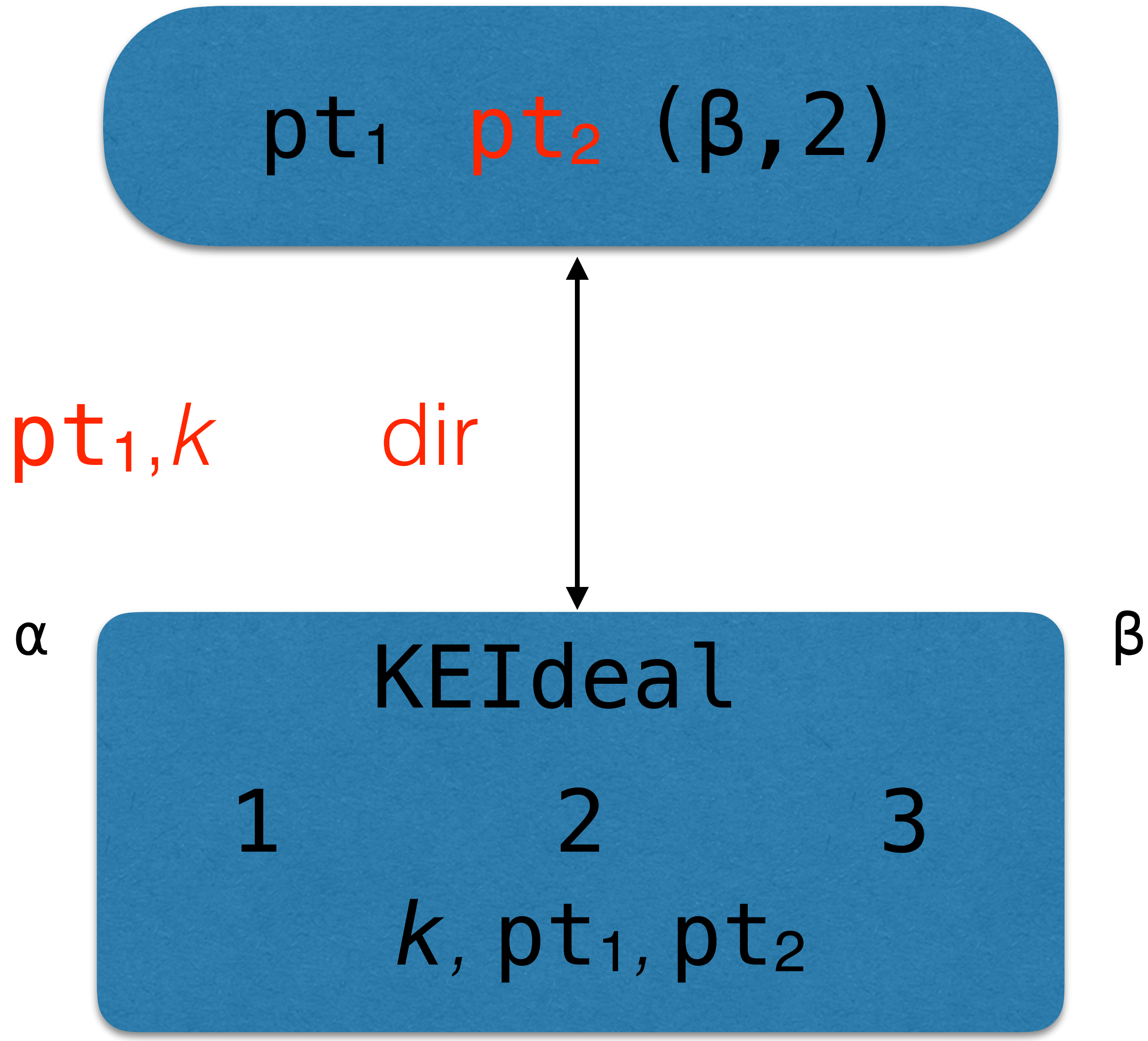
Key Exchange Ideal Functionality



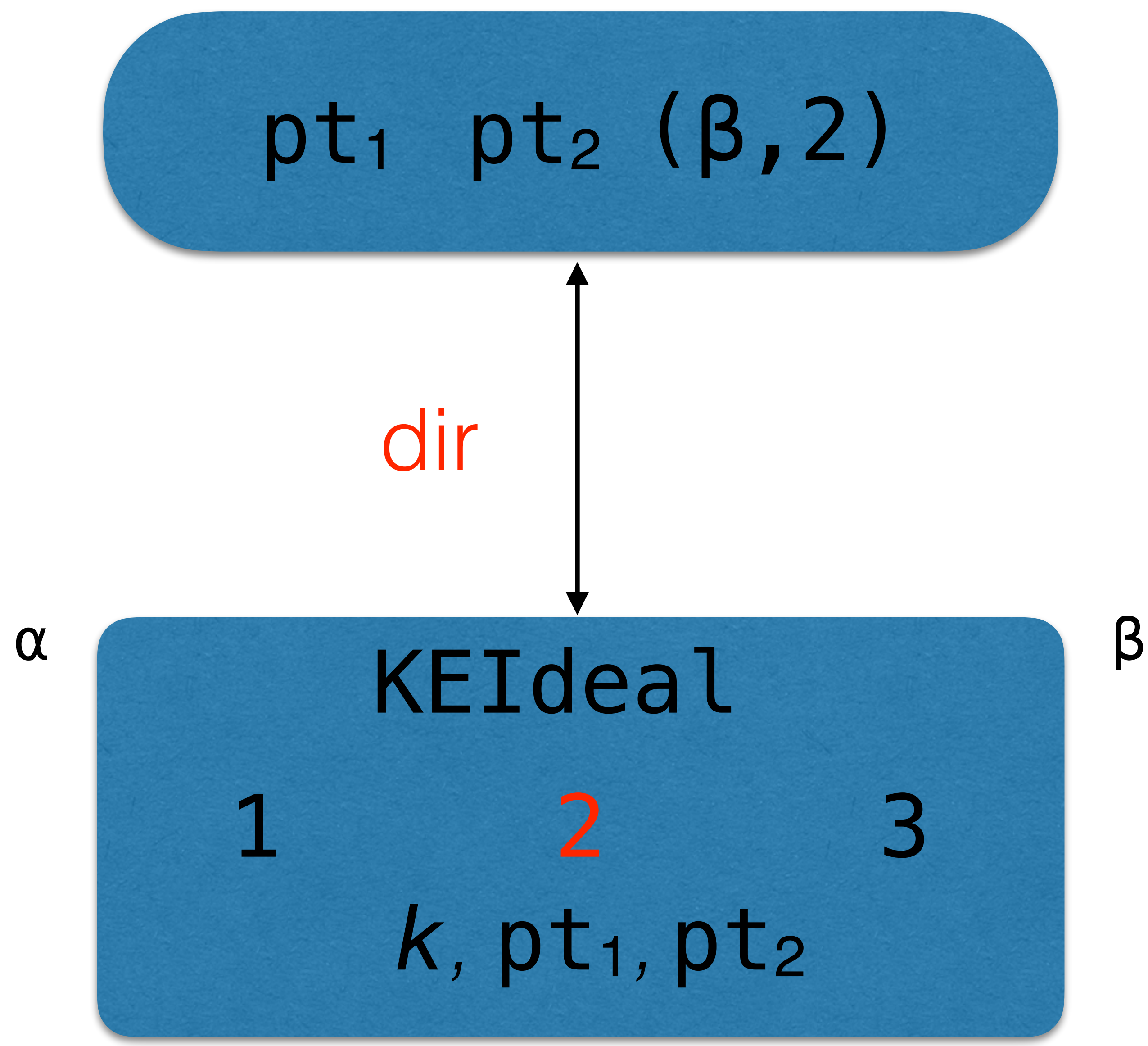
Key Exchange Ideal Functionality



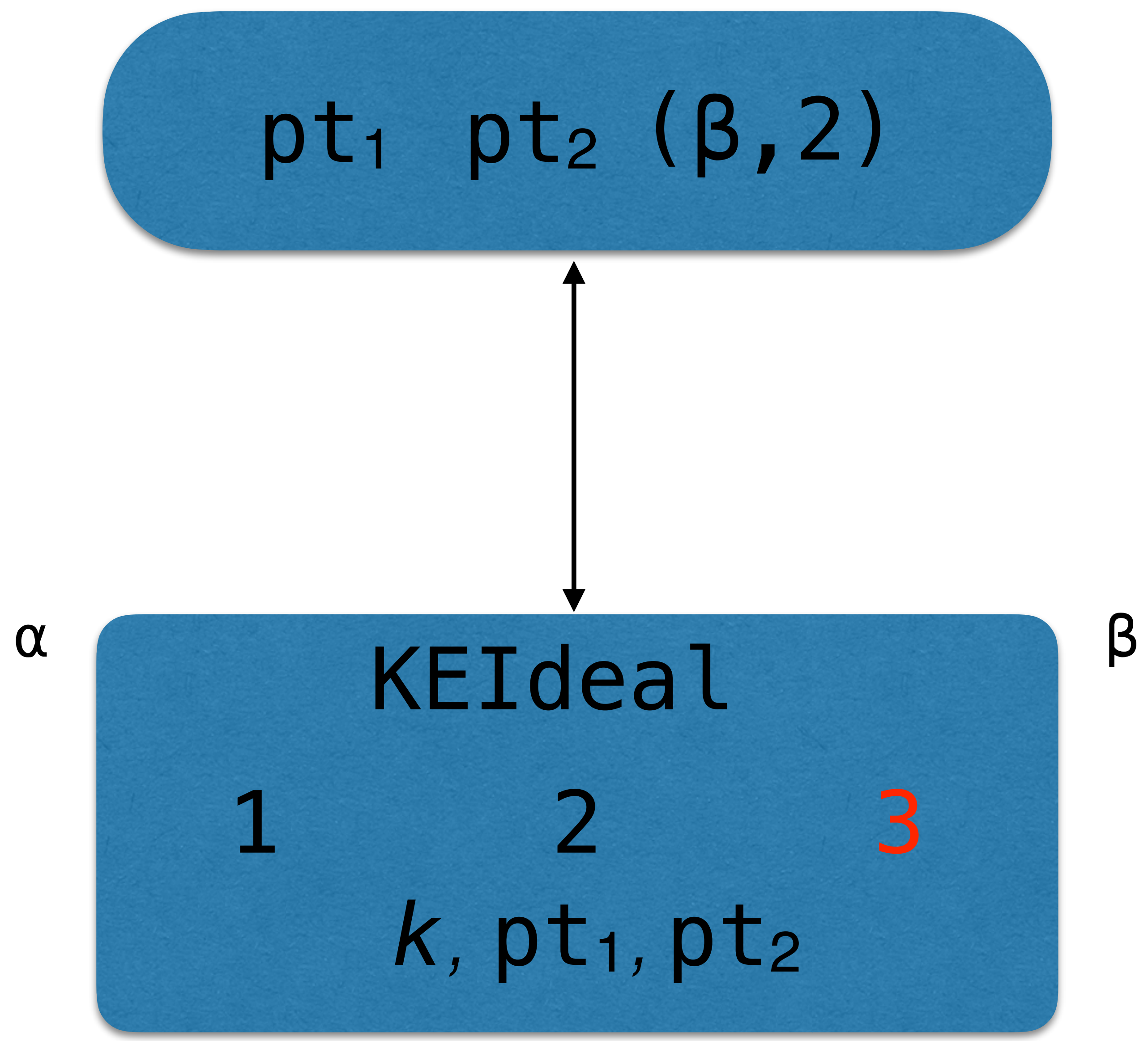
Key Exchange Ideal Functionality



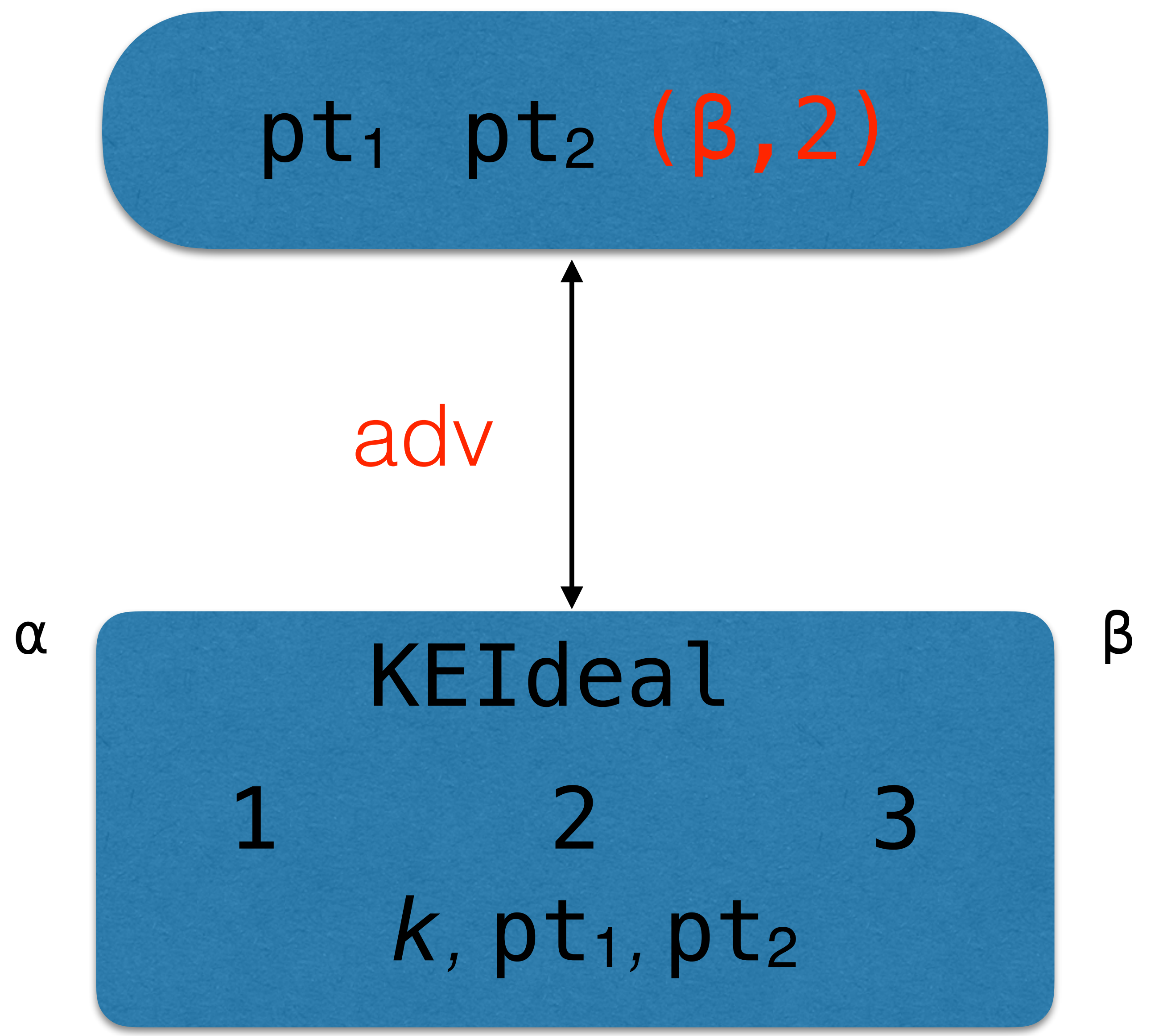
Key Exchange Ideal Functionality



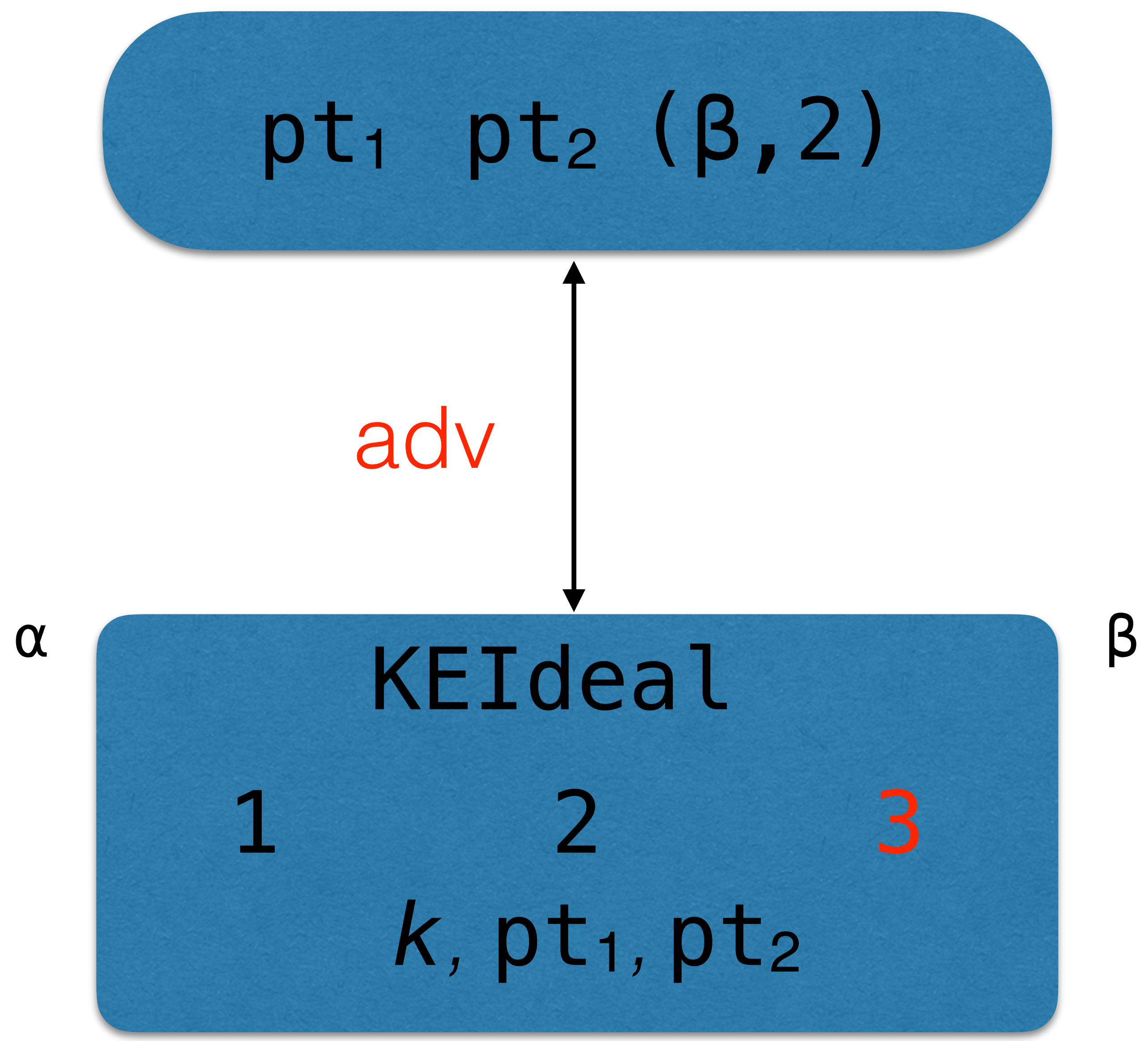
Key Exchange Ideal Functionality



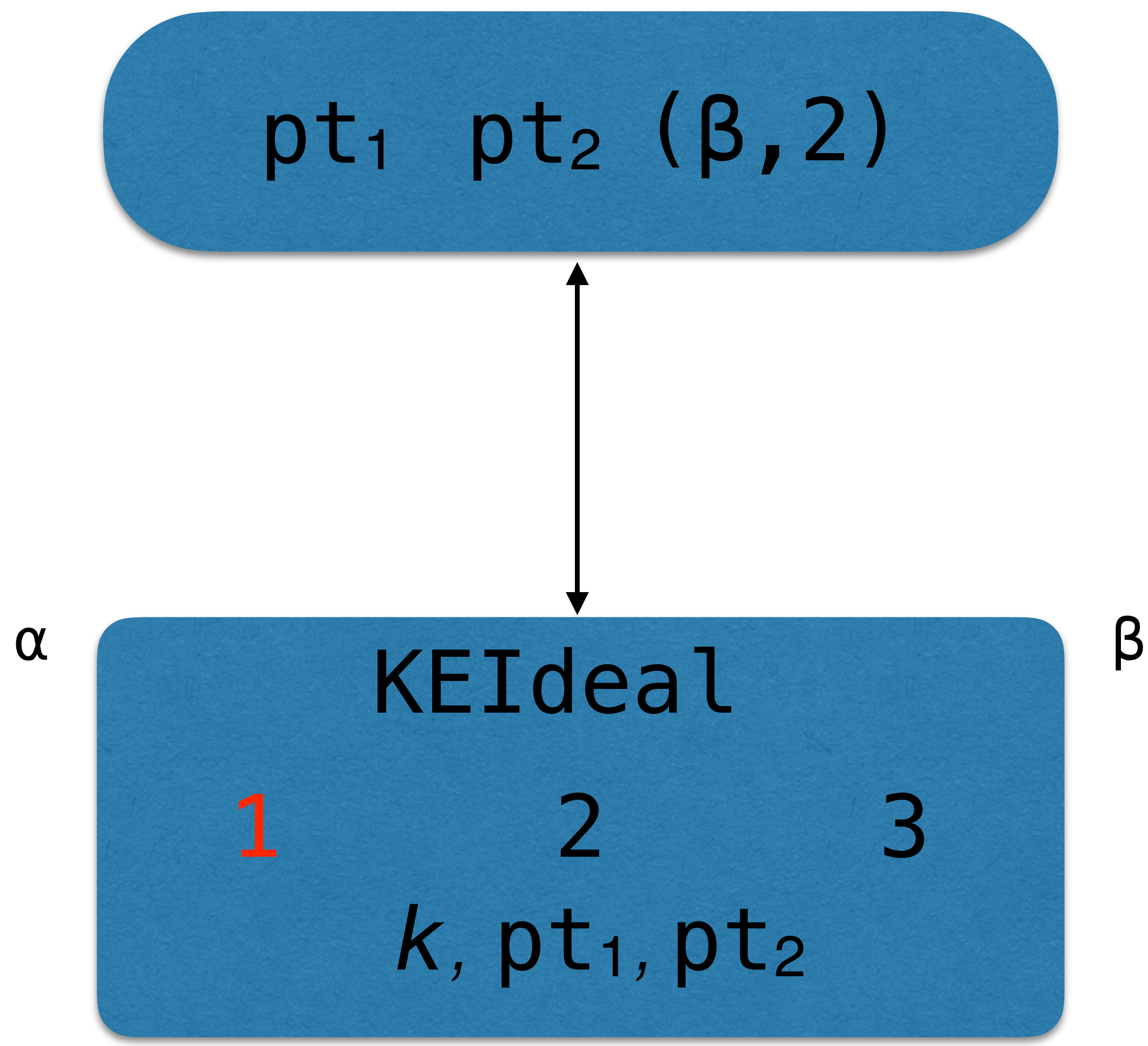
Key Exchange Ideal Functionality



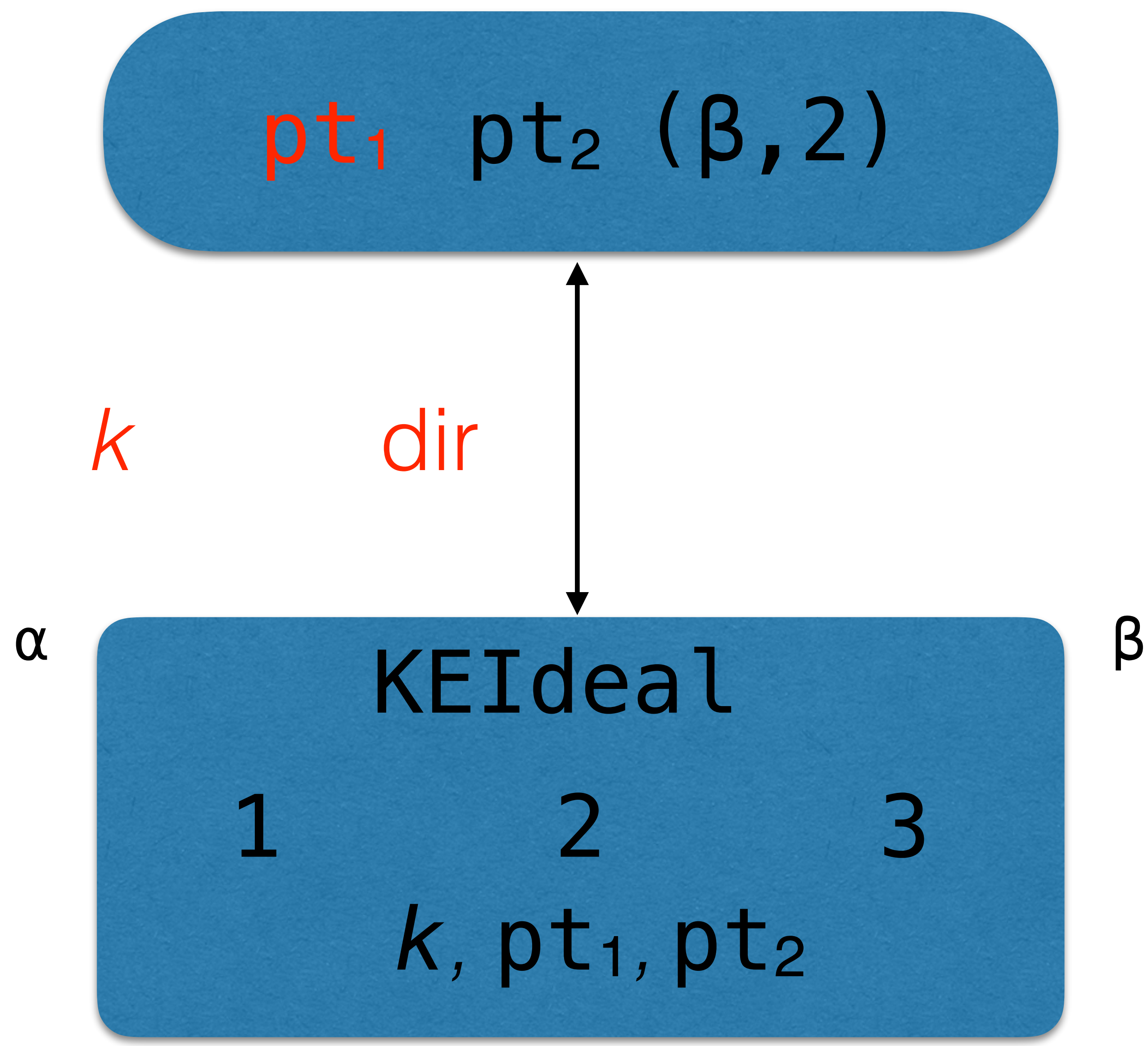
Key Exchange Ideal Functionality



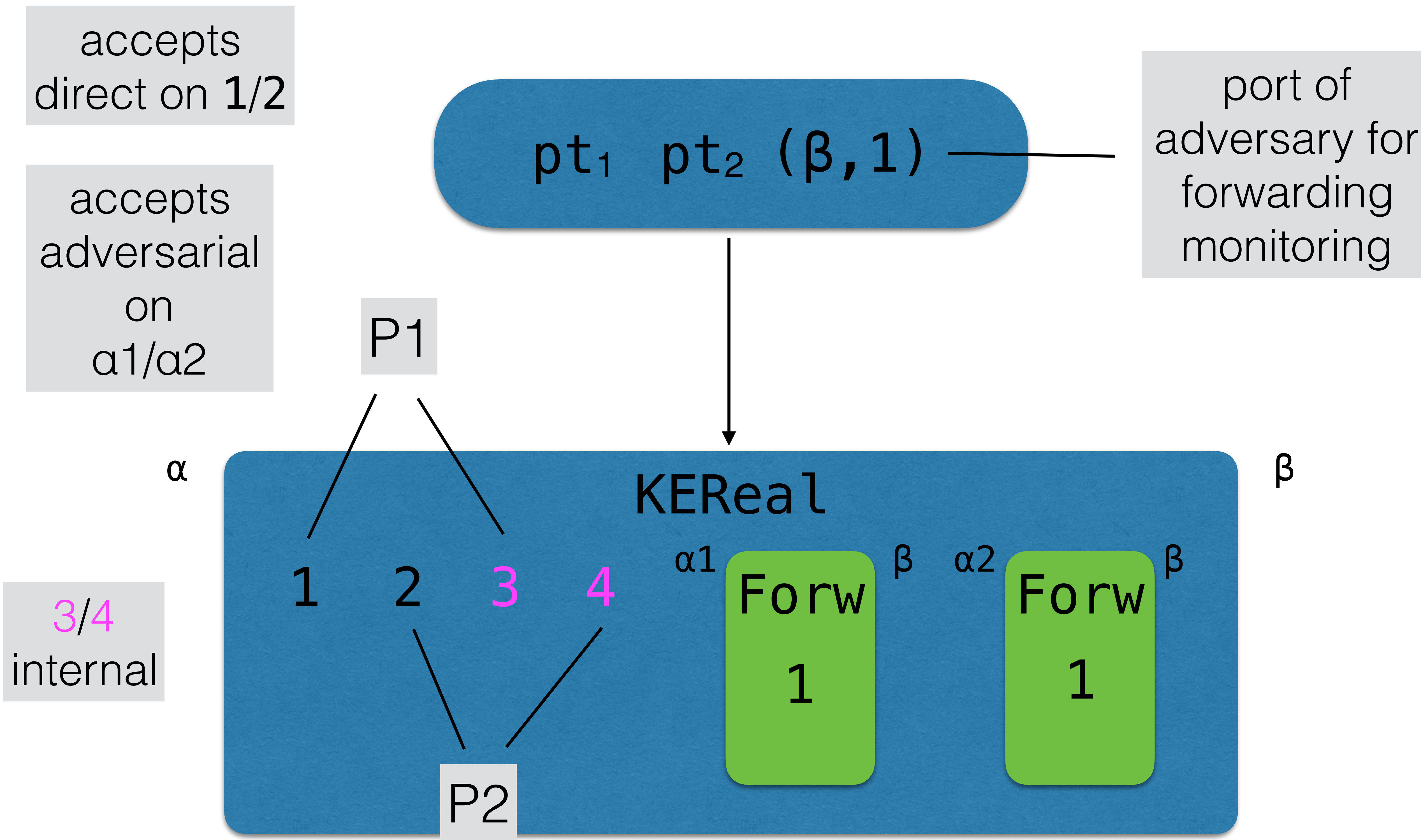
Key Exchange Ideal Functionality



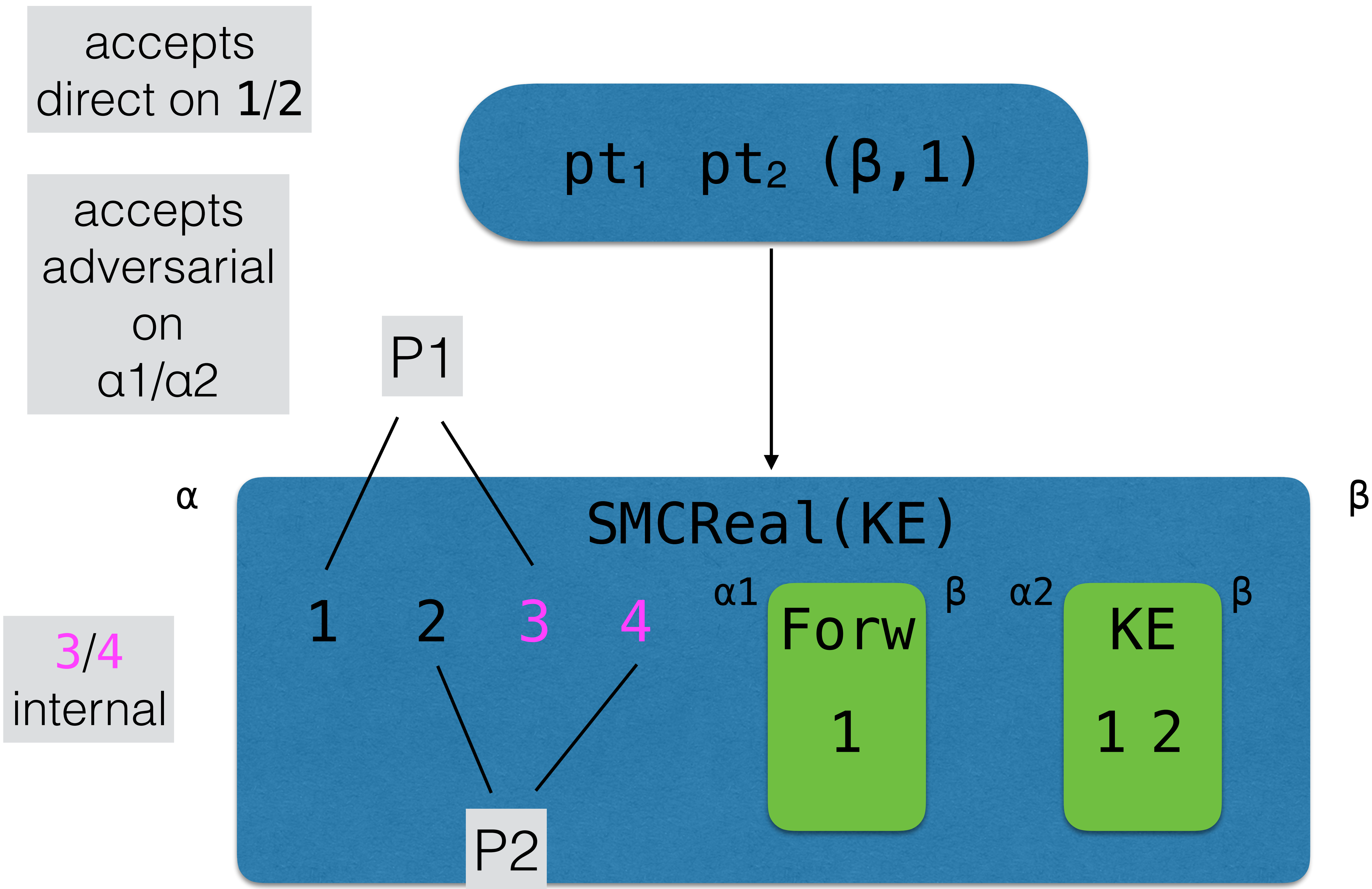
Key Exchange Ideal Functionality



Key Exchange Real Functionality

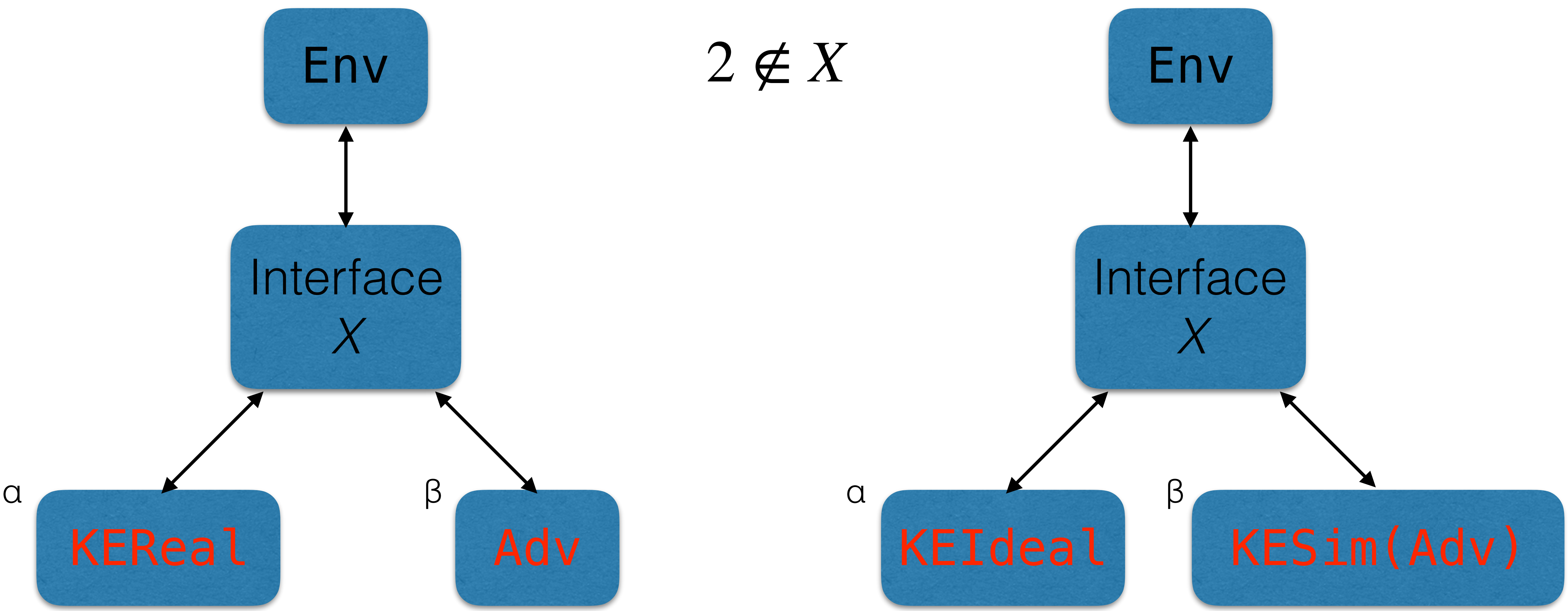


SMC Real Functionality

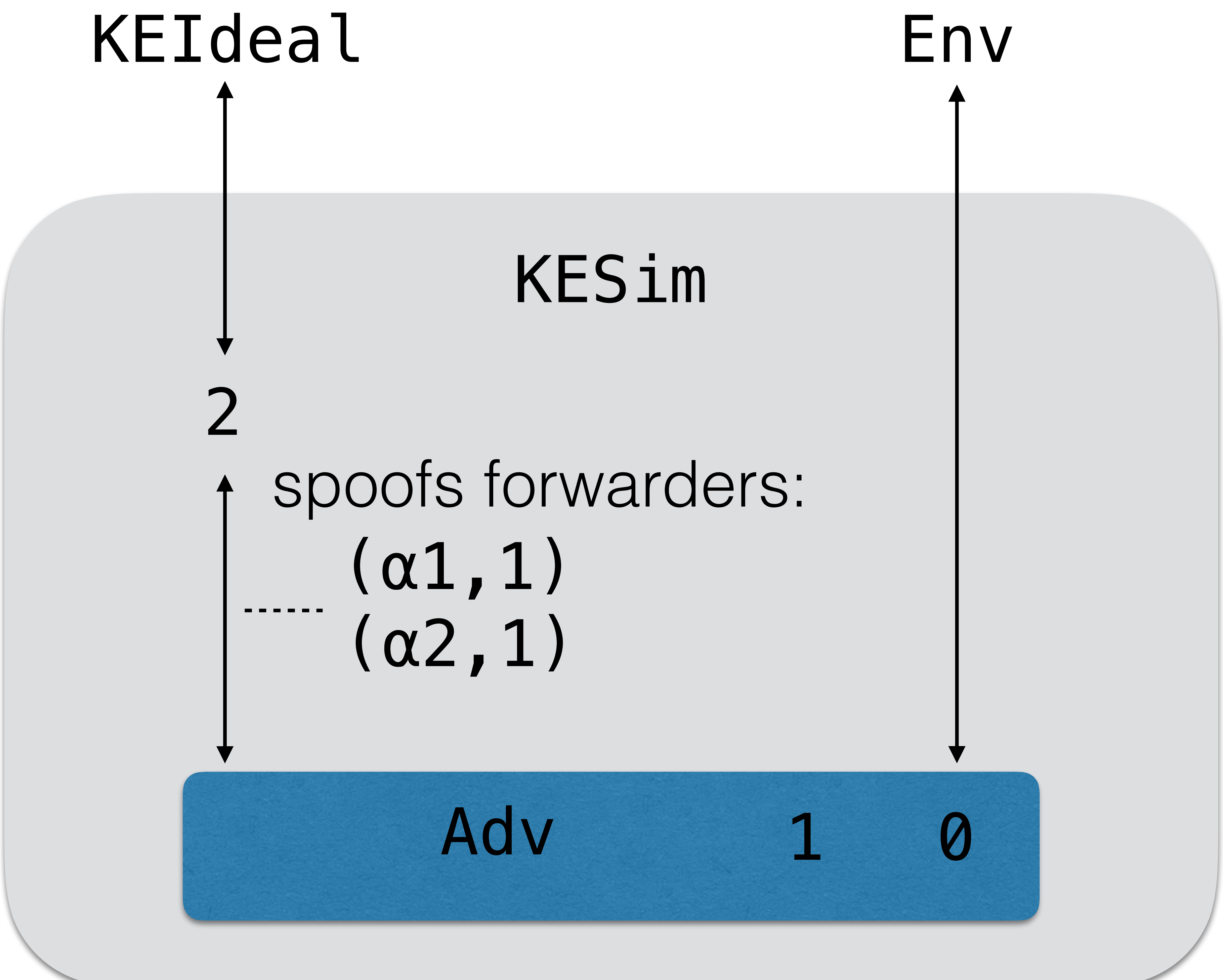


Key Exchange Security

To prove the security of key exchange, we must formulate a simulator, **`KESim`**, and connect the real and ideal games via a sequence of intermediate games:



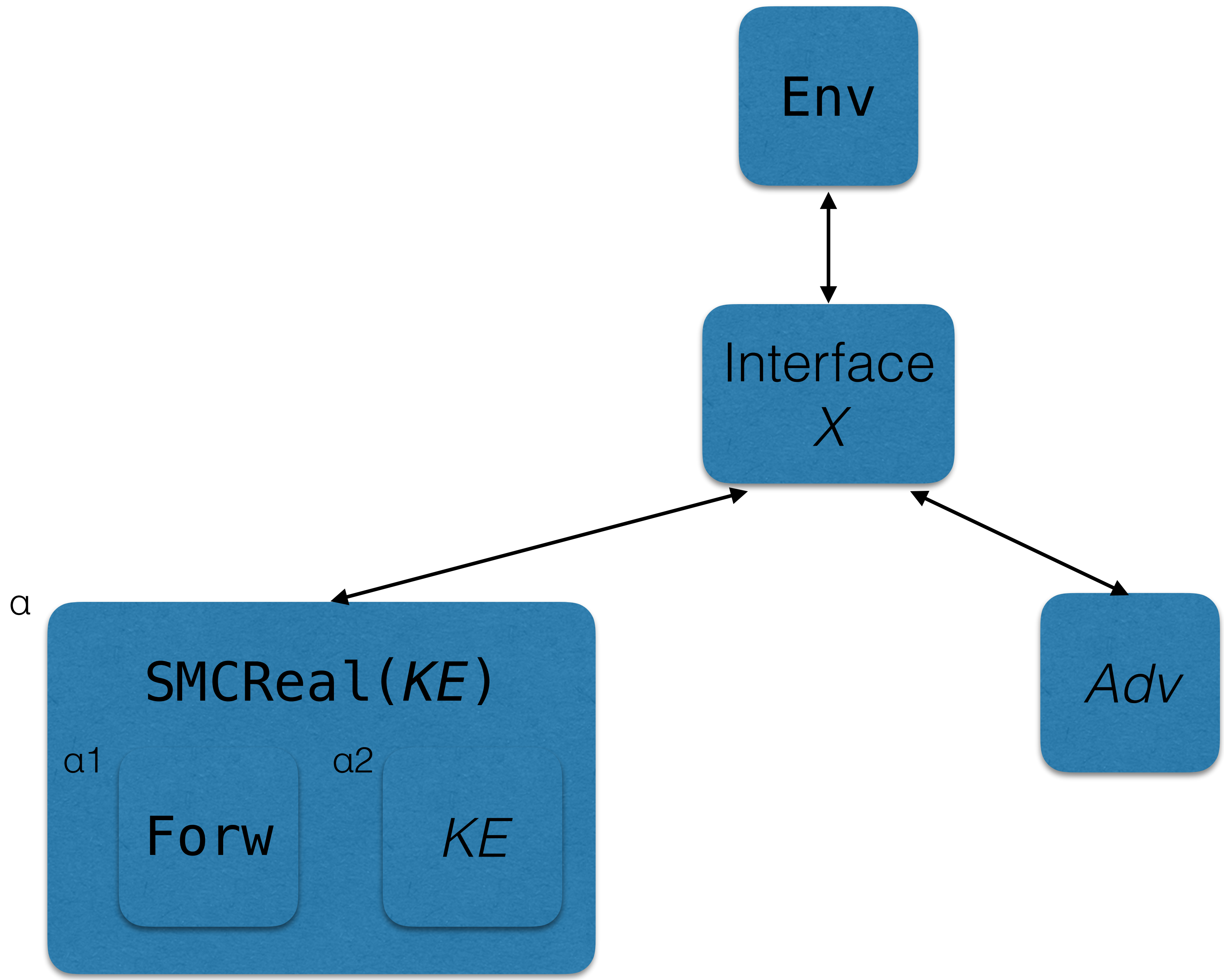
Key Exchange Simulator



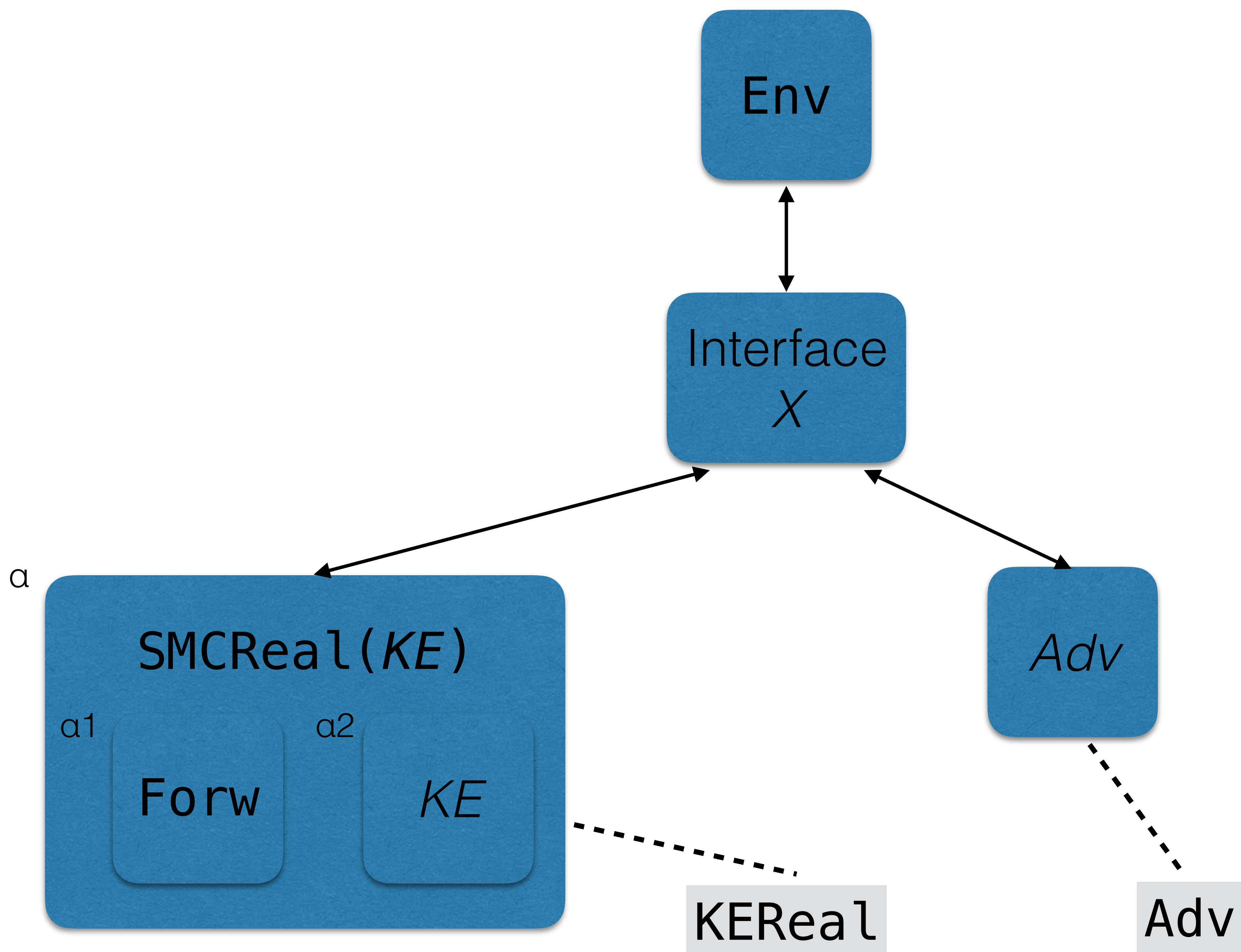
Key Exchange Sequence of Games

- Use EasyCrypt's eager/lazy sampling to move choices of random exponents to beginning of game
- Reduce to Decisional Diffie-Hellman (DDH) assumption
  - Constructed DDH adversary parameterized by `Env` and `Adv`
- Now the agreed upon key is `g` ^ q_3, for a random q_3
- Use eager/lazy sampling to delay generation of exponents
- Connect this hybrid game with ideal game

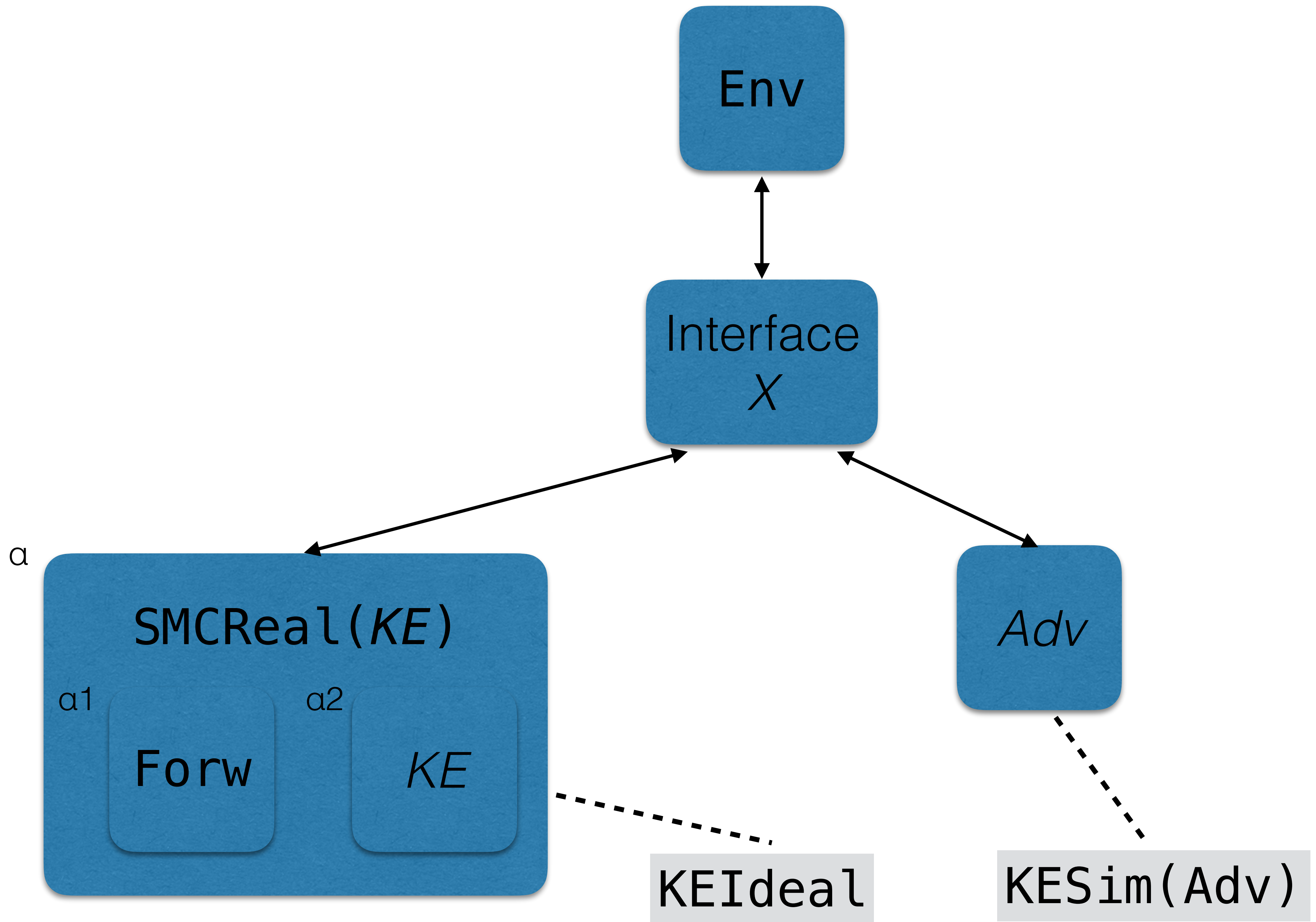
Instance of Composition Theorem



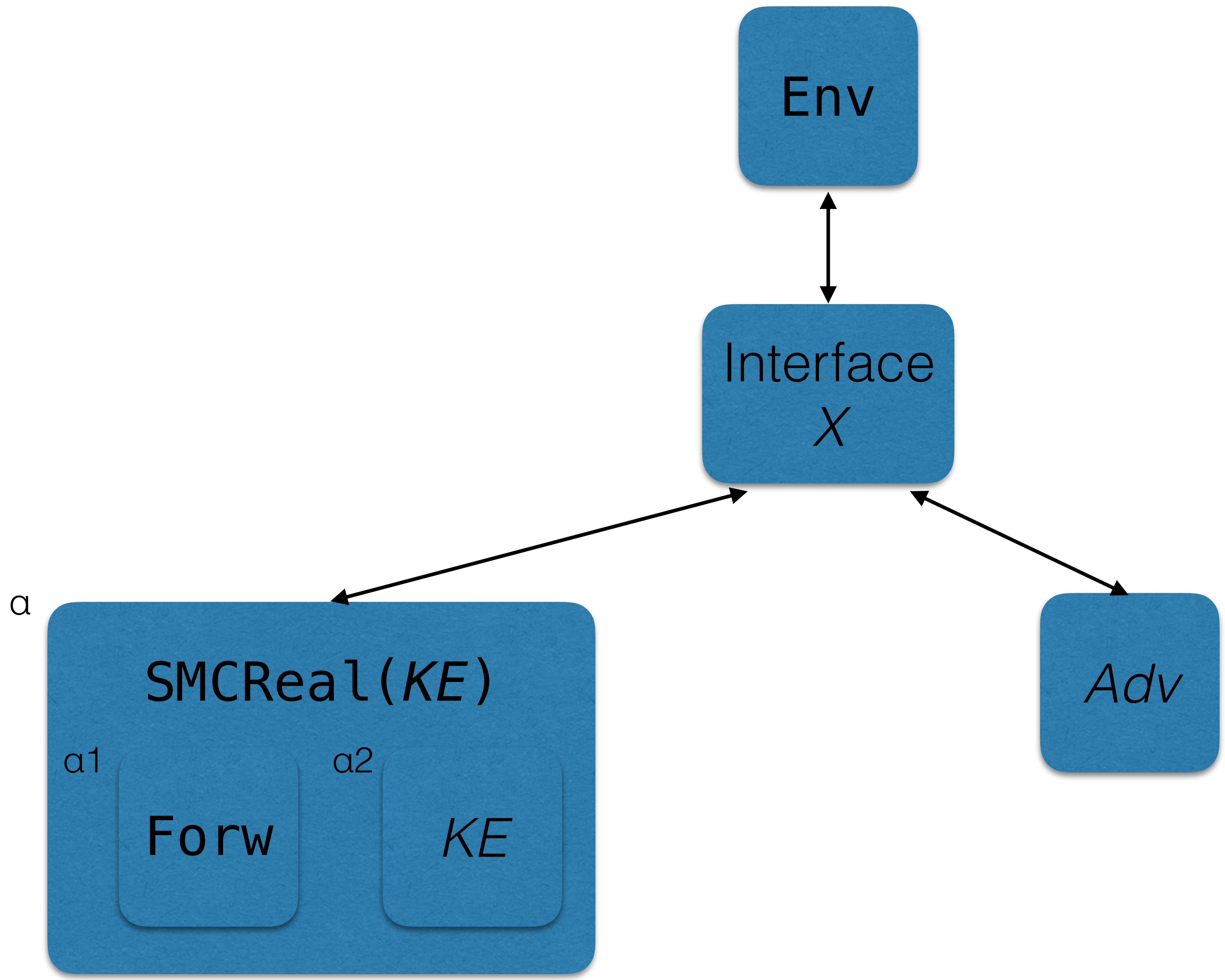
Instance of Composition Theorem



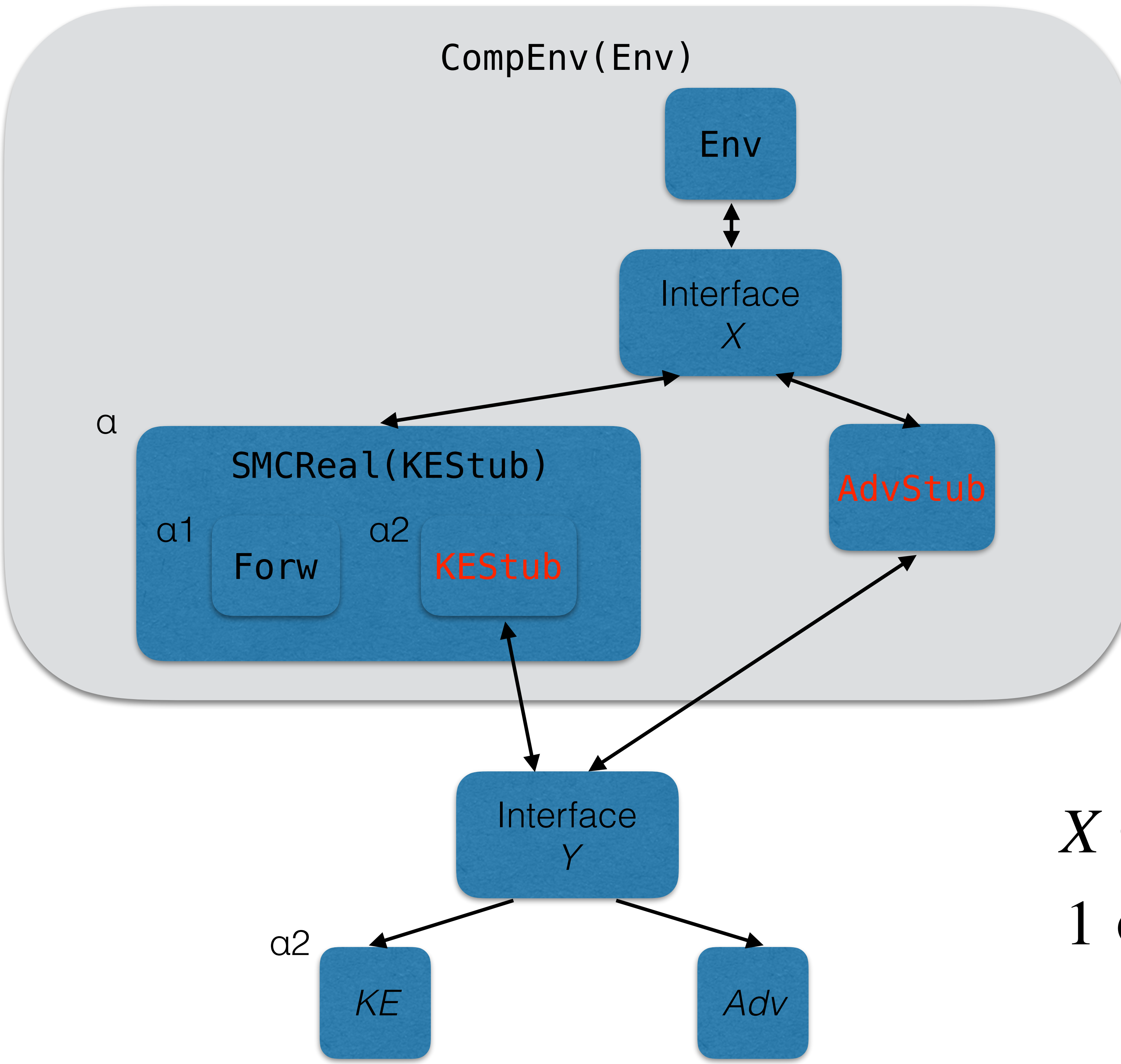
Instance Composition Theorem



Bridging Lemma for Composition Theorem



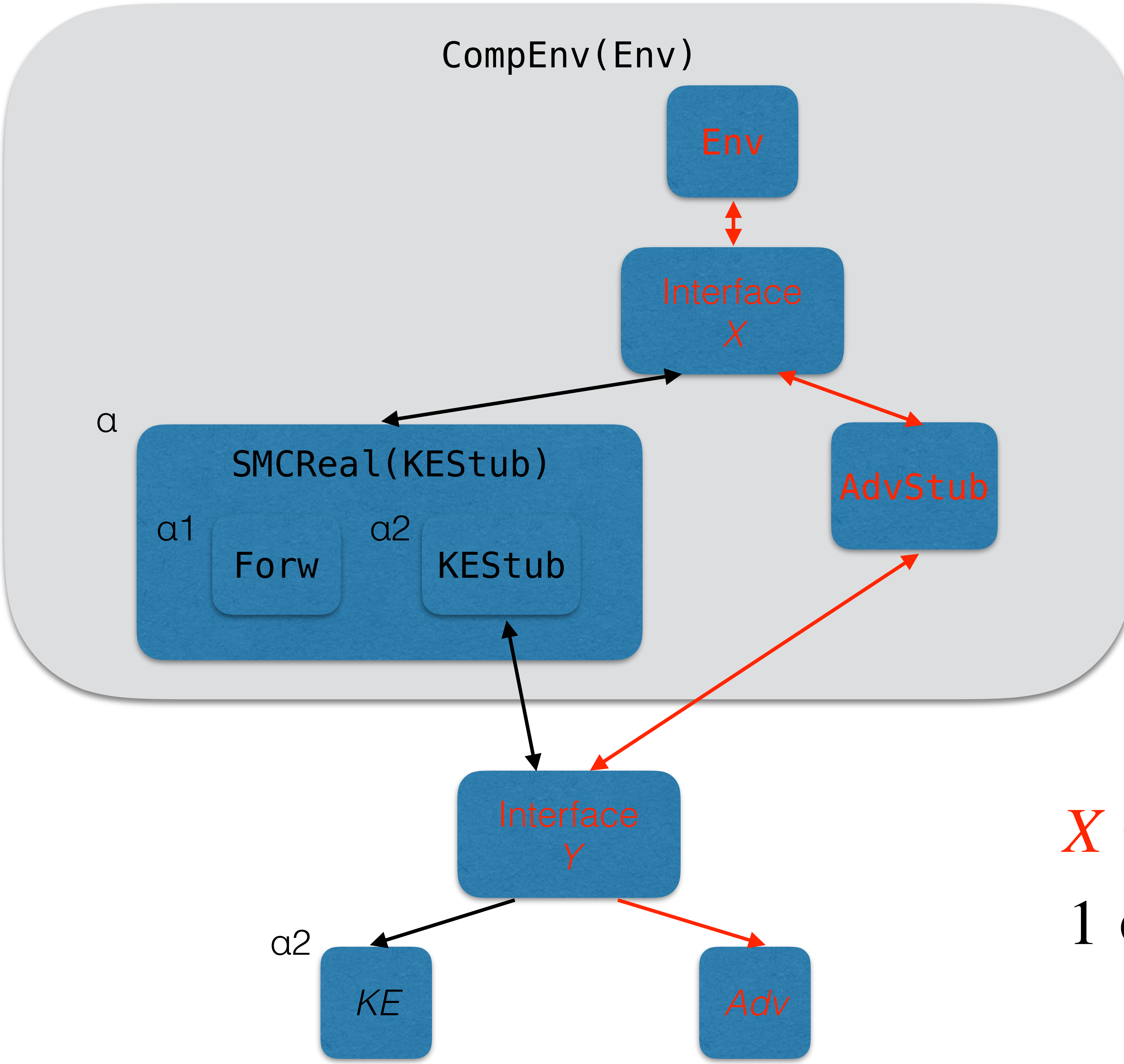
Bridging Lemma for Composition Theorem



$X \subseteq Y$

$1 \in Y$

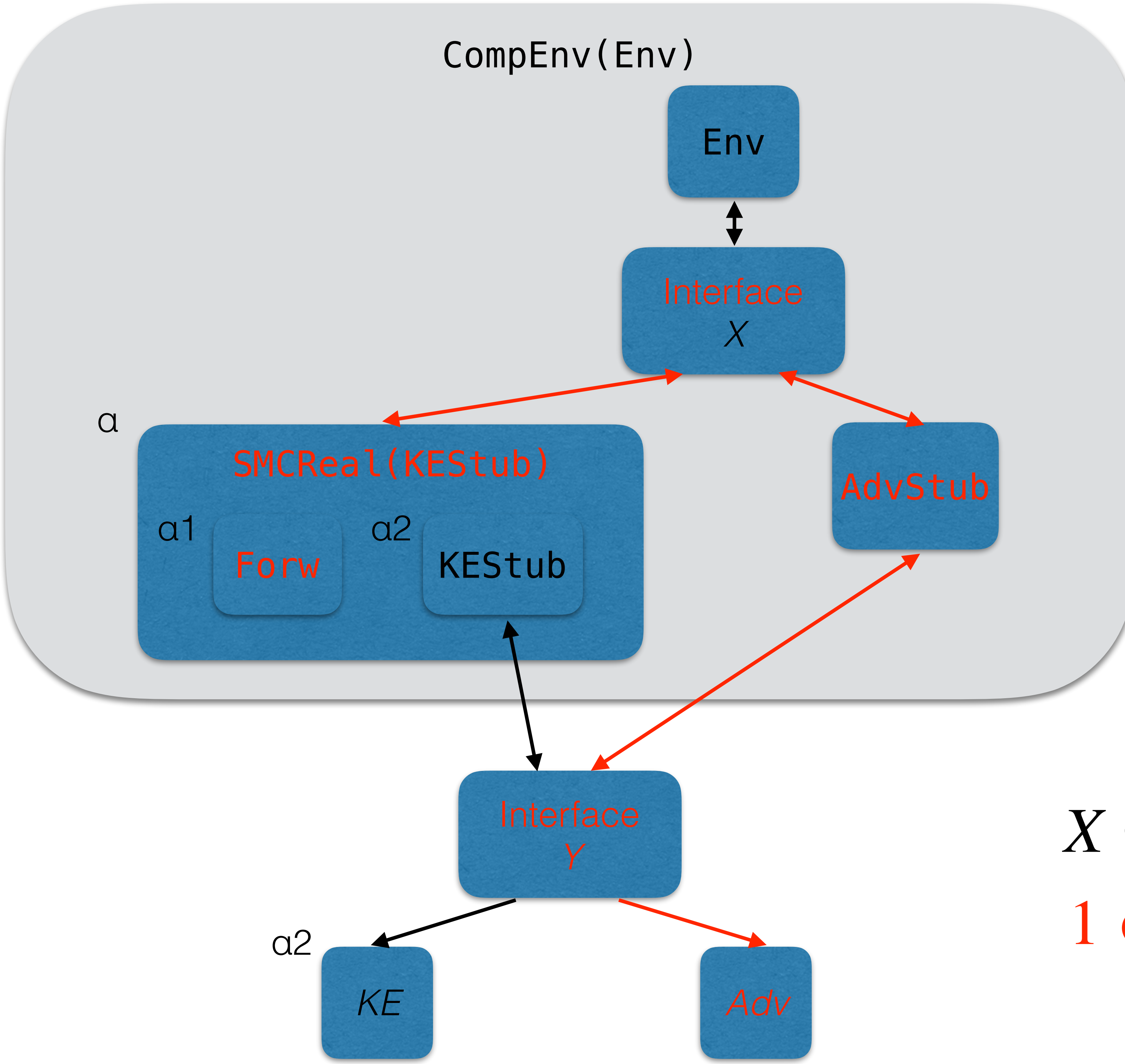
Bridging Lemma for Composition Theorem



$X \subseteq Y$

$1 \in Y$

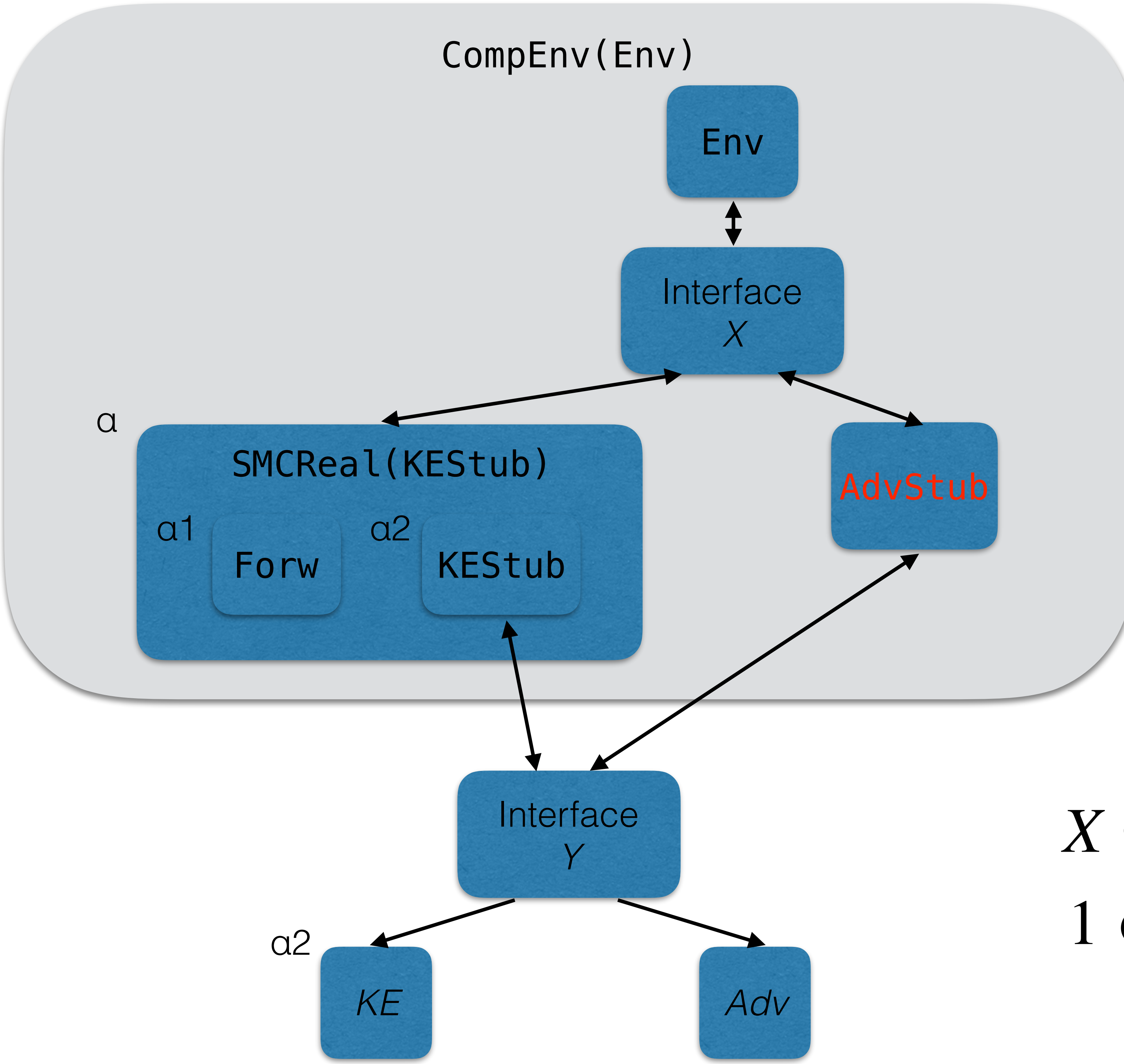
Bridging Lemma for Composition Theorem



$X \subseteq Y$

$1 \in Y$

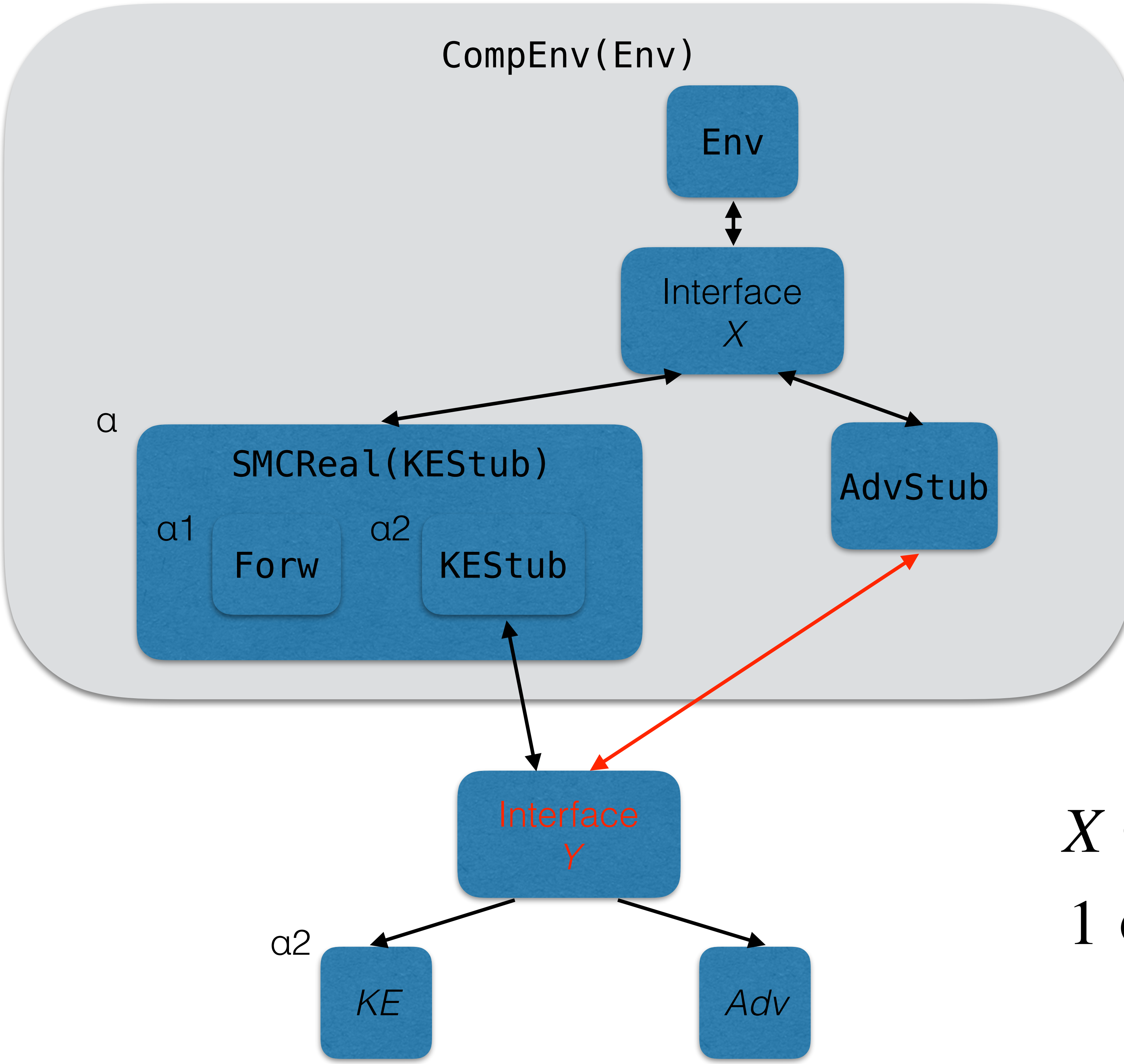
Bridging Lemma for Composition Theorem



$X \subseteq Y$

$1 \in Y$

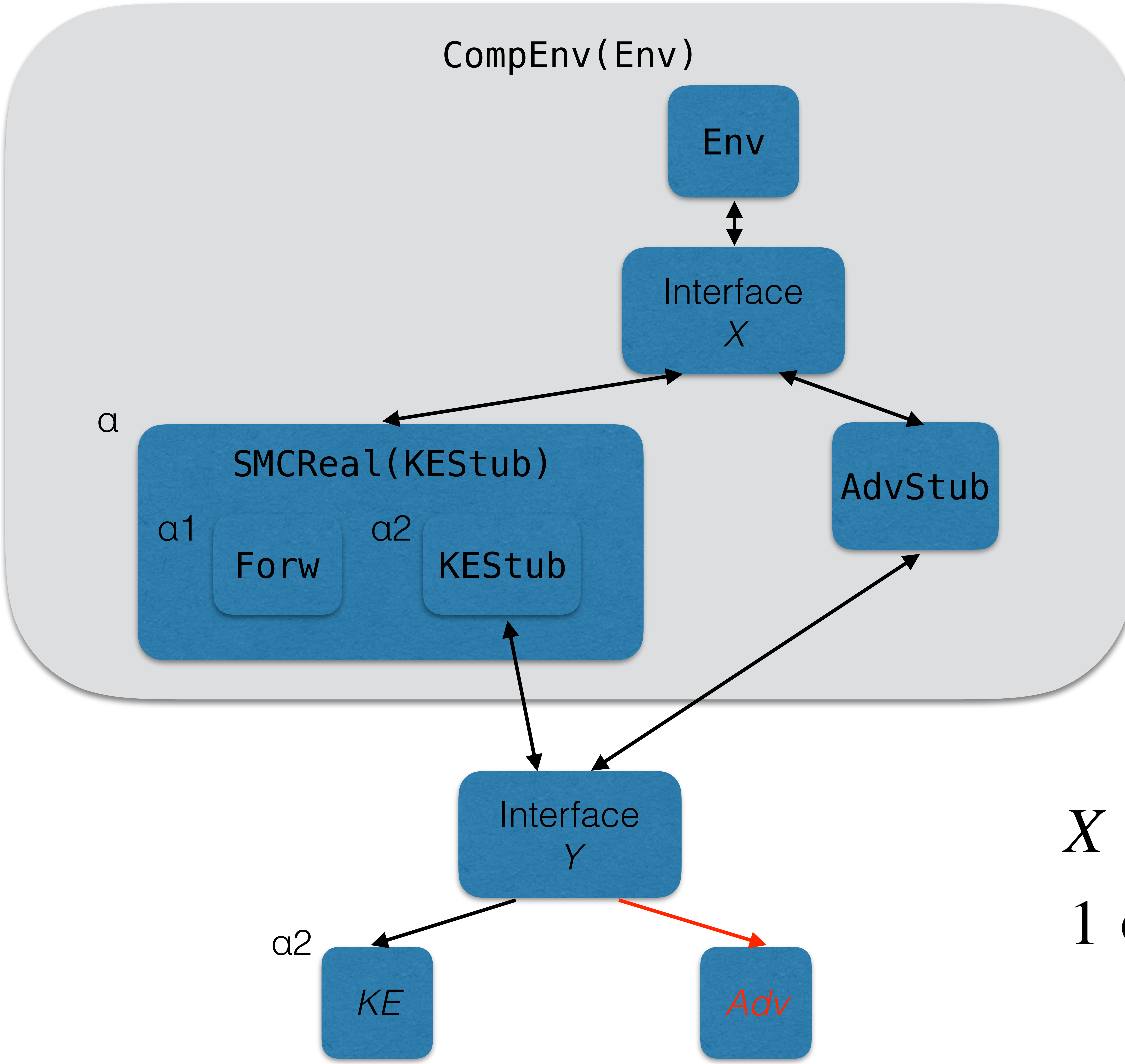
Bridging Lemma for Composition Theorem



$X \subseteq Y$

$1 \in Y$

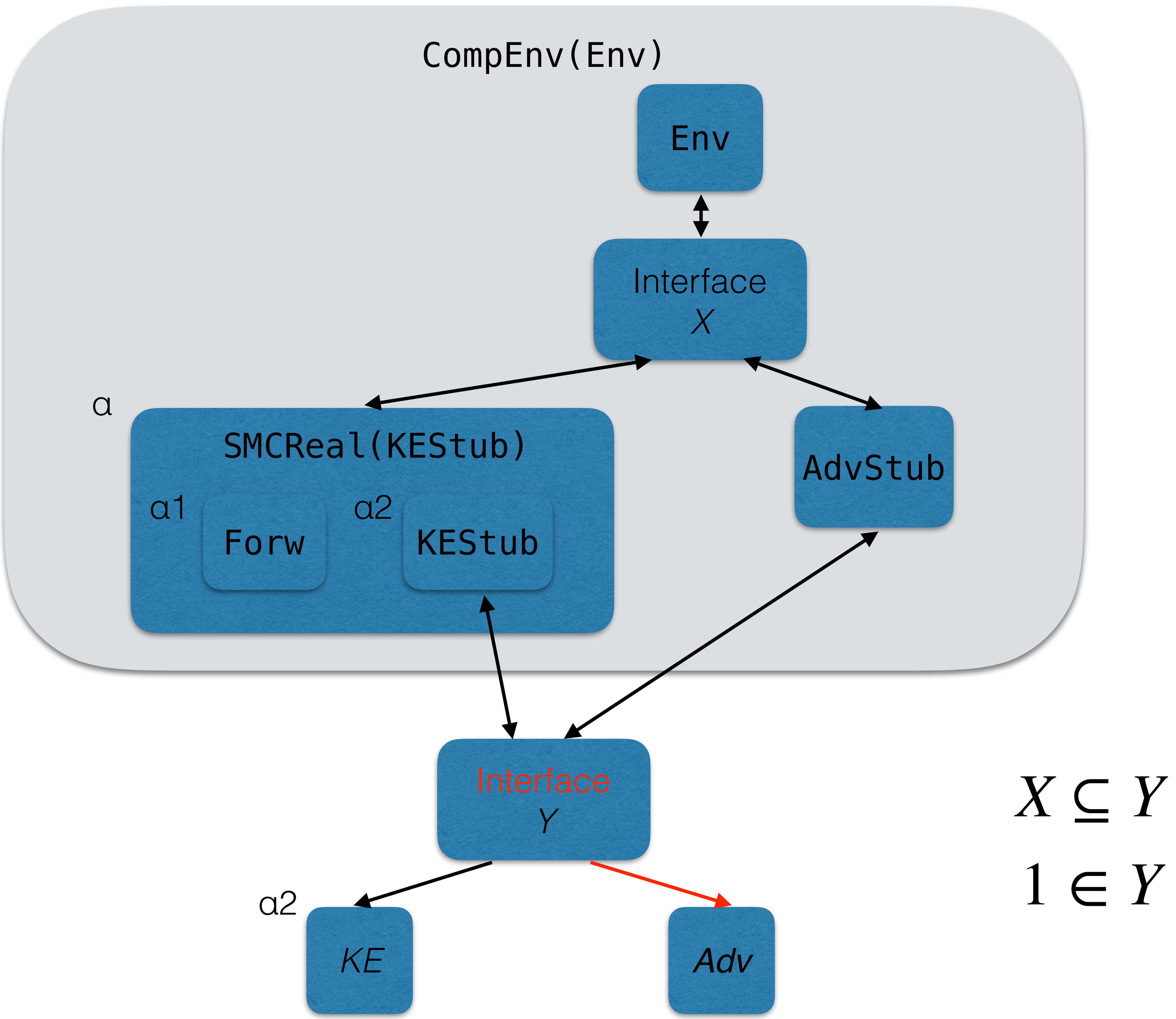
Bridging Lemma for Composition Theorem



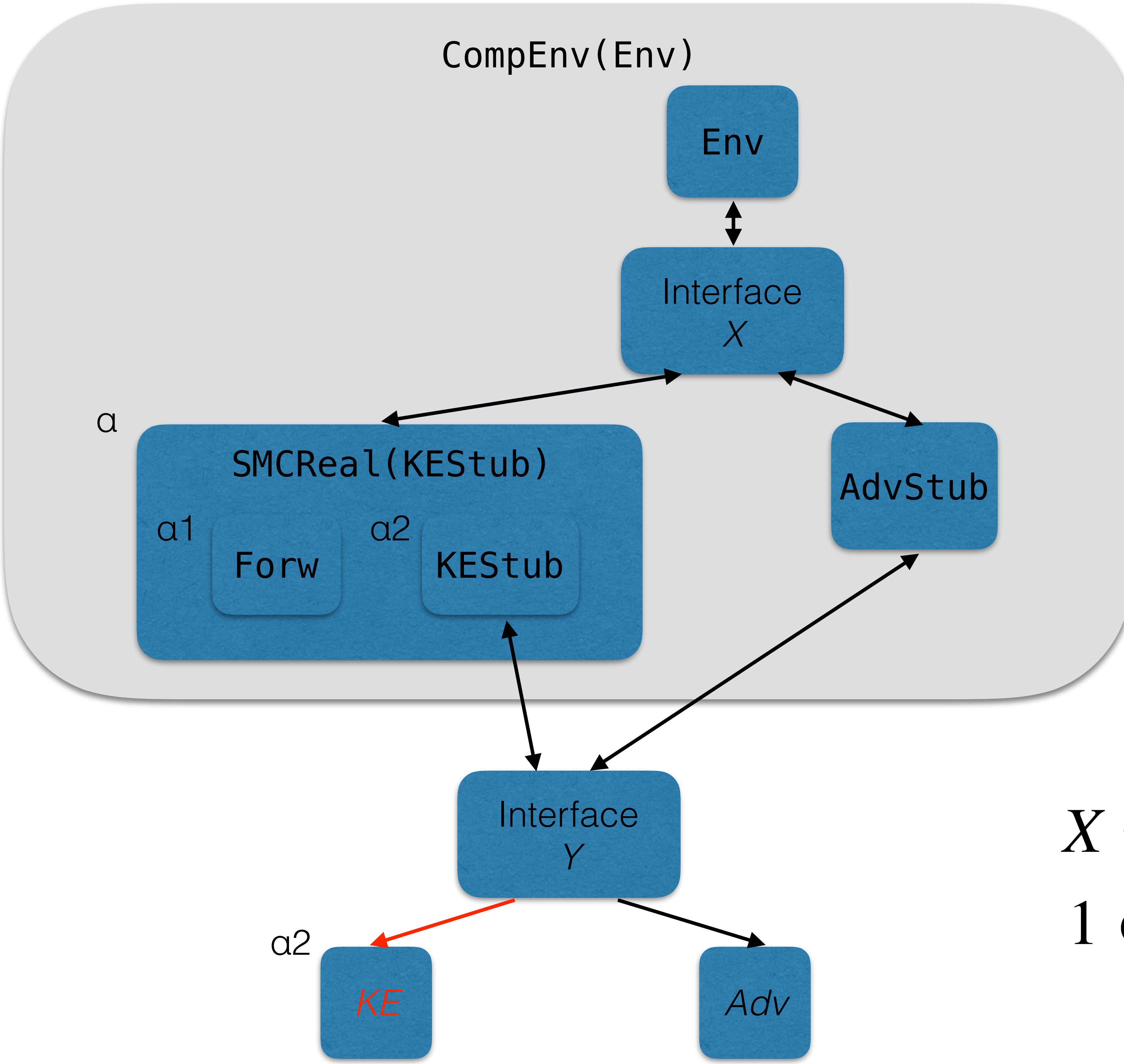
$X \subseteq Y$

$1 \in Y$

Bridging Lemma for Composition Theorem



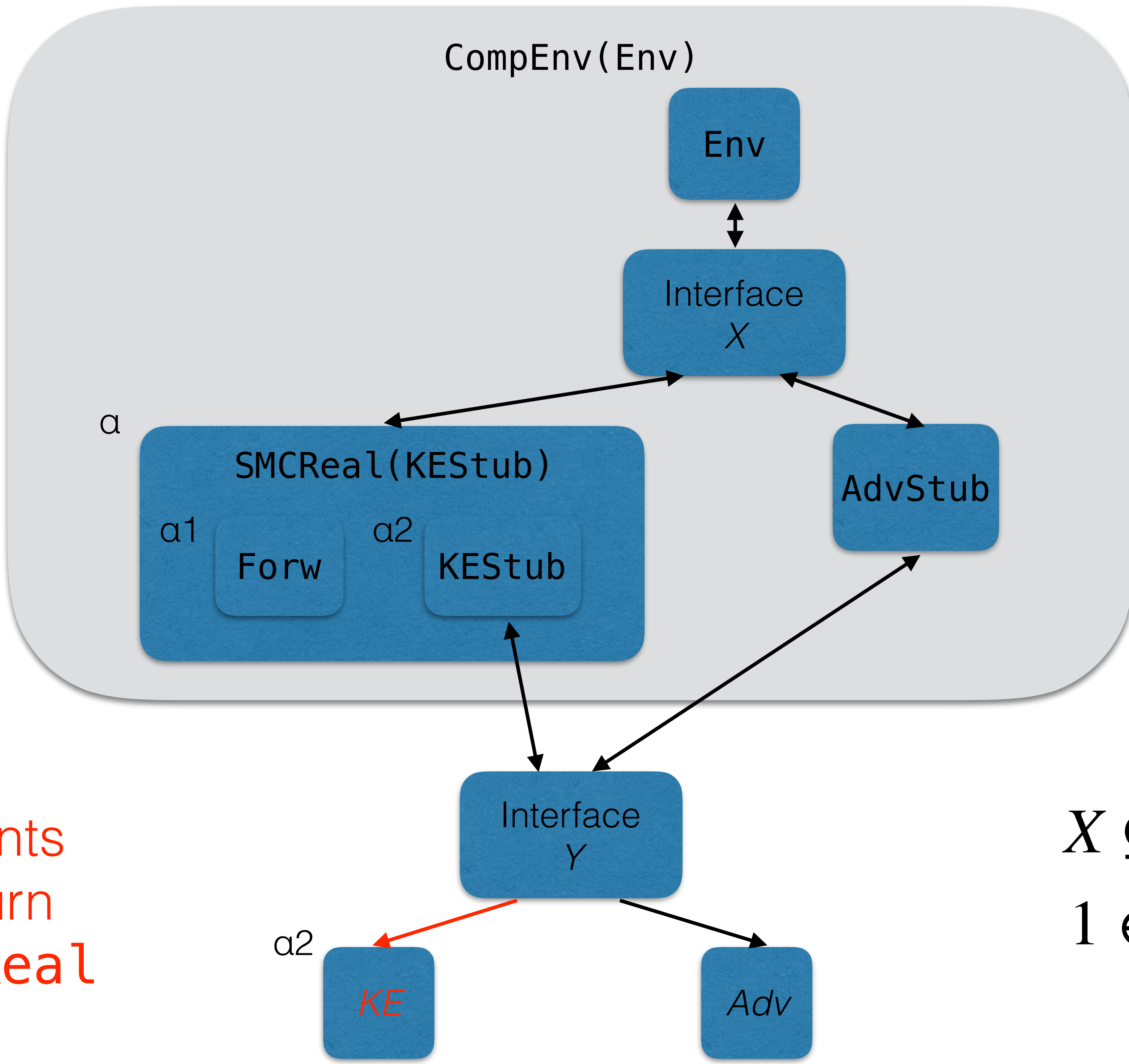
Bridging Lemma for Composition Theorem



$X \subseteq Y$

$1 \in Y$

Bridging Lemma for Composition Theorem

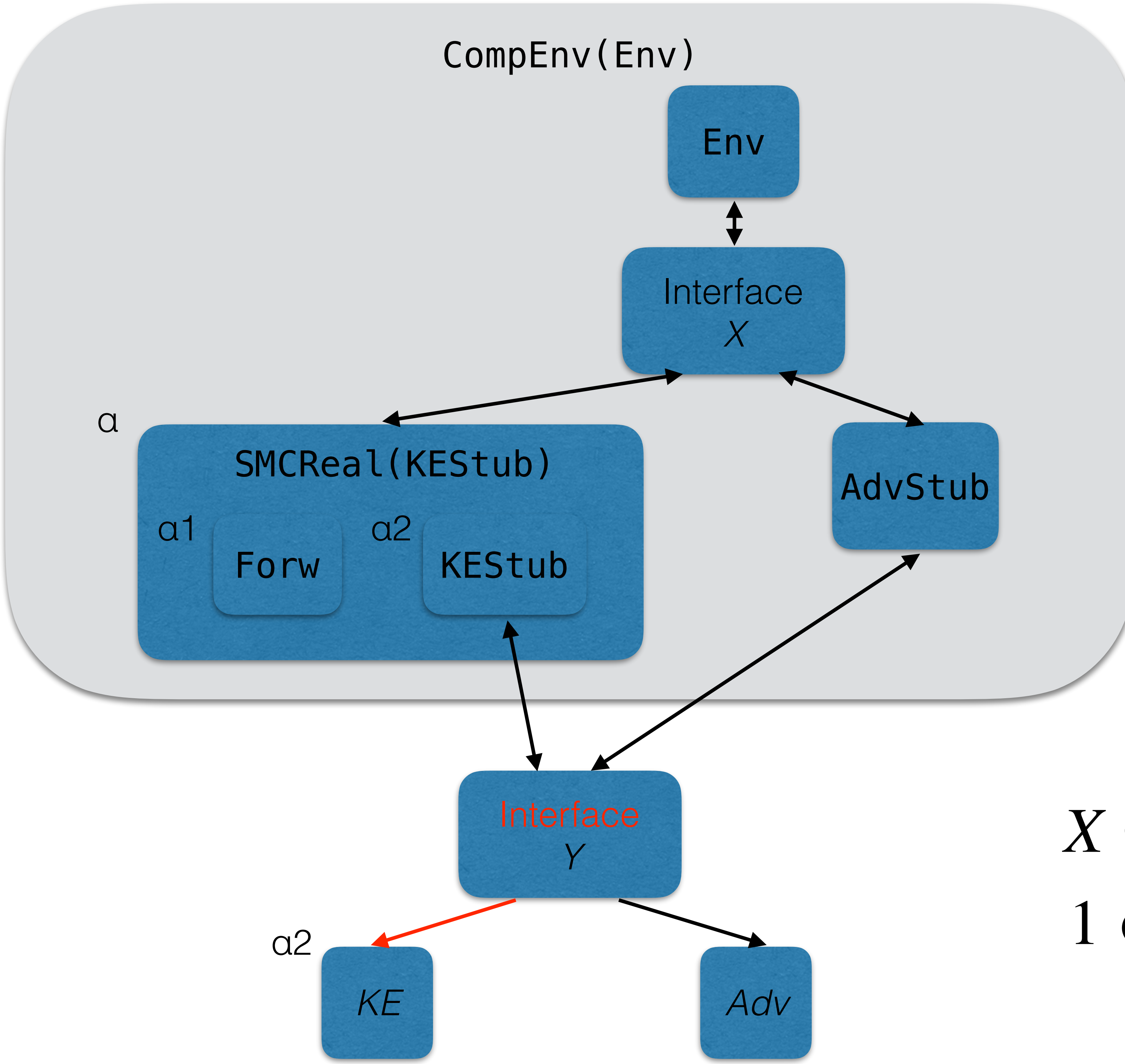


KE wants
to return
to **SMCReal**

$X \subseteq Y$

$1 \in Y$

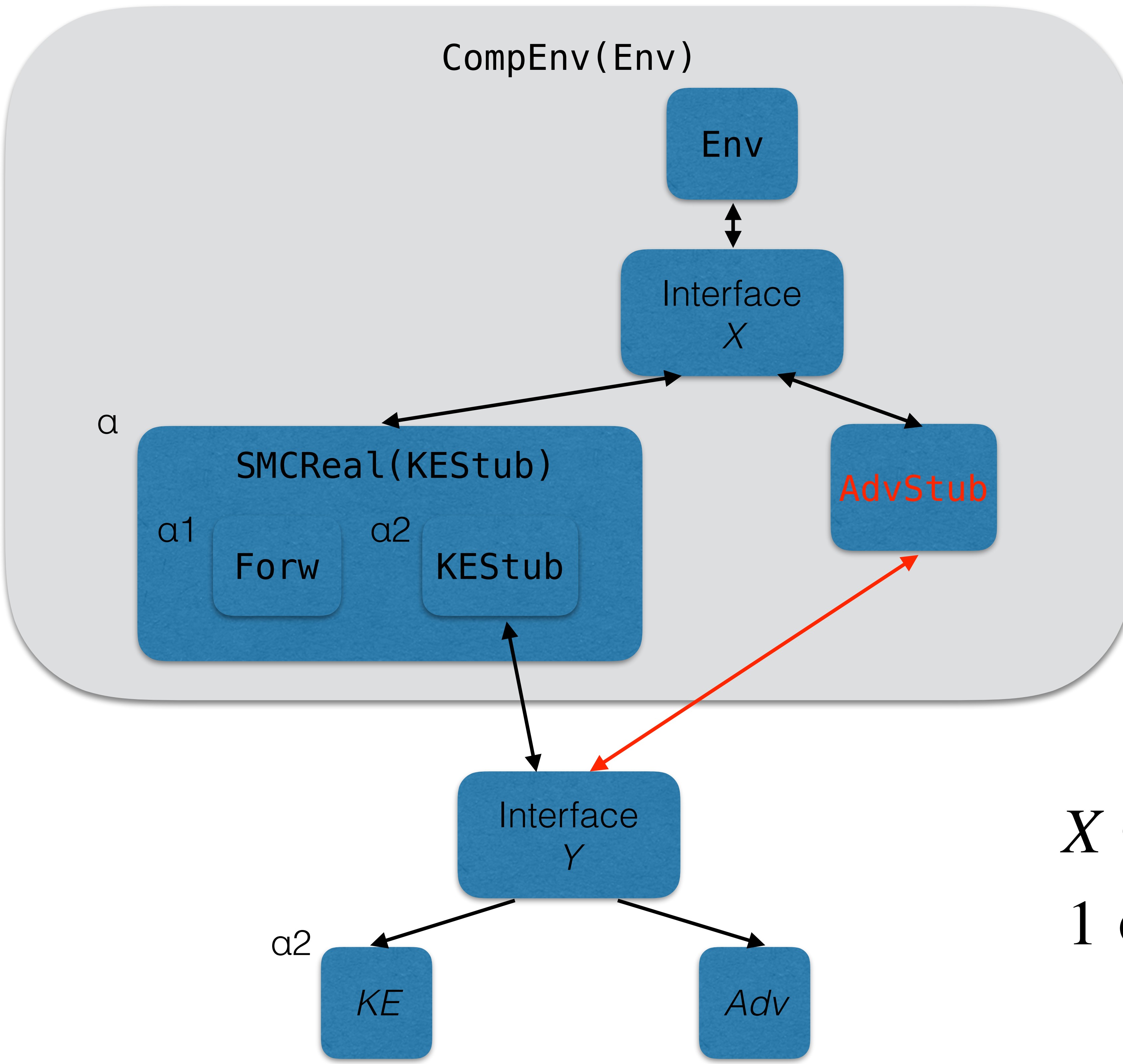
Bridging Lemma for Composition Theorem



$$X \subseteq Y$$

$$1 \in Y$$

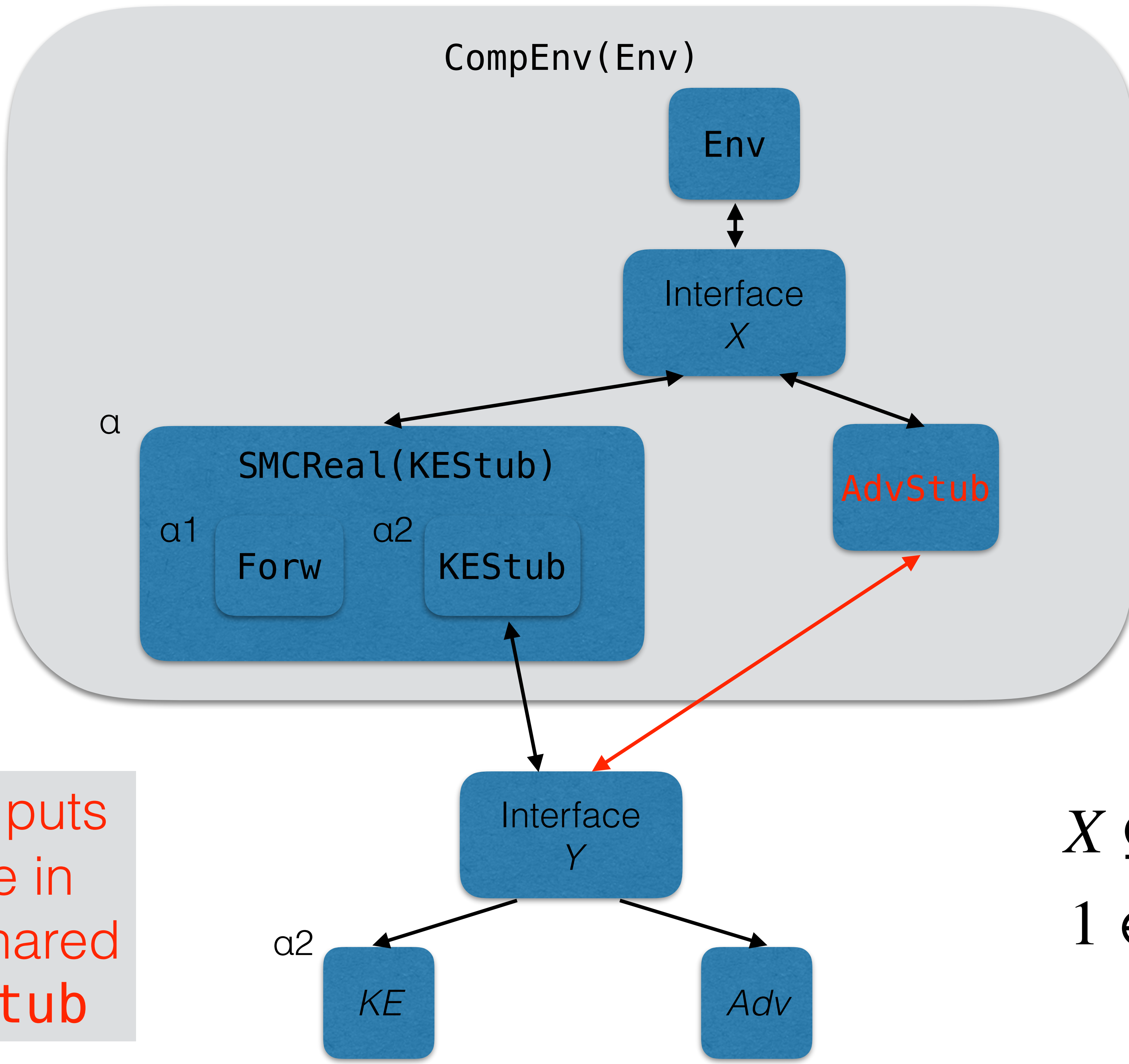
Bridging Lemma for Composition Theorem



$X \subseteq Y$

$1 \in Y$

Bridging Lemma for Composition Theorem

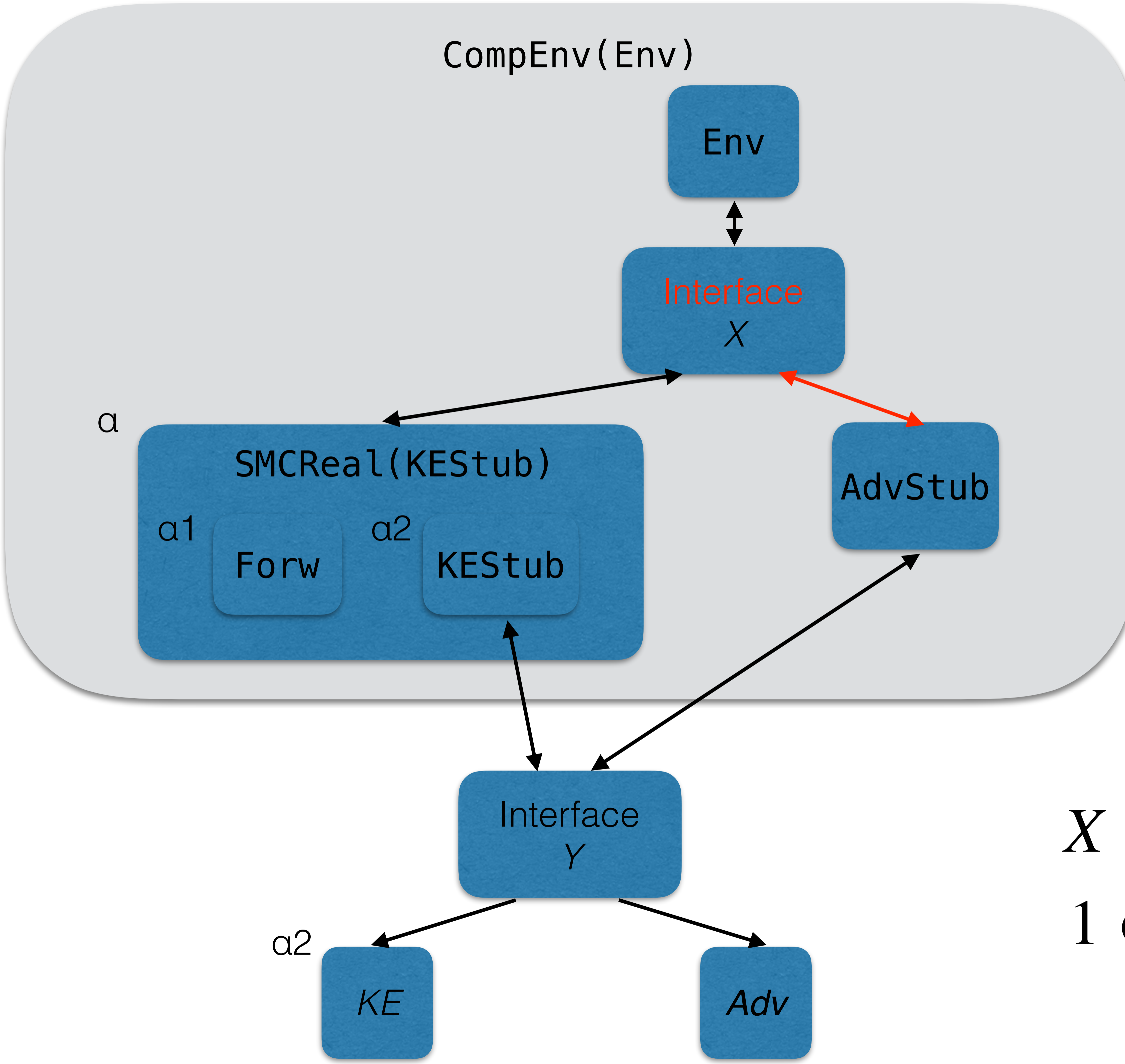


AdvStub puts message in mailbox shared with KEStub

$X \subseteq Y$

$1 \in Y$

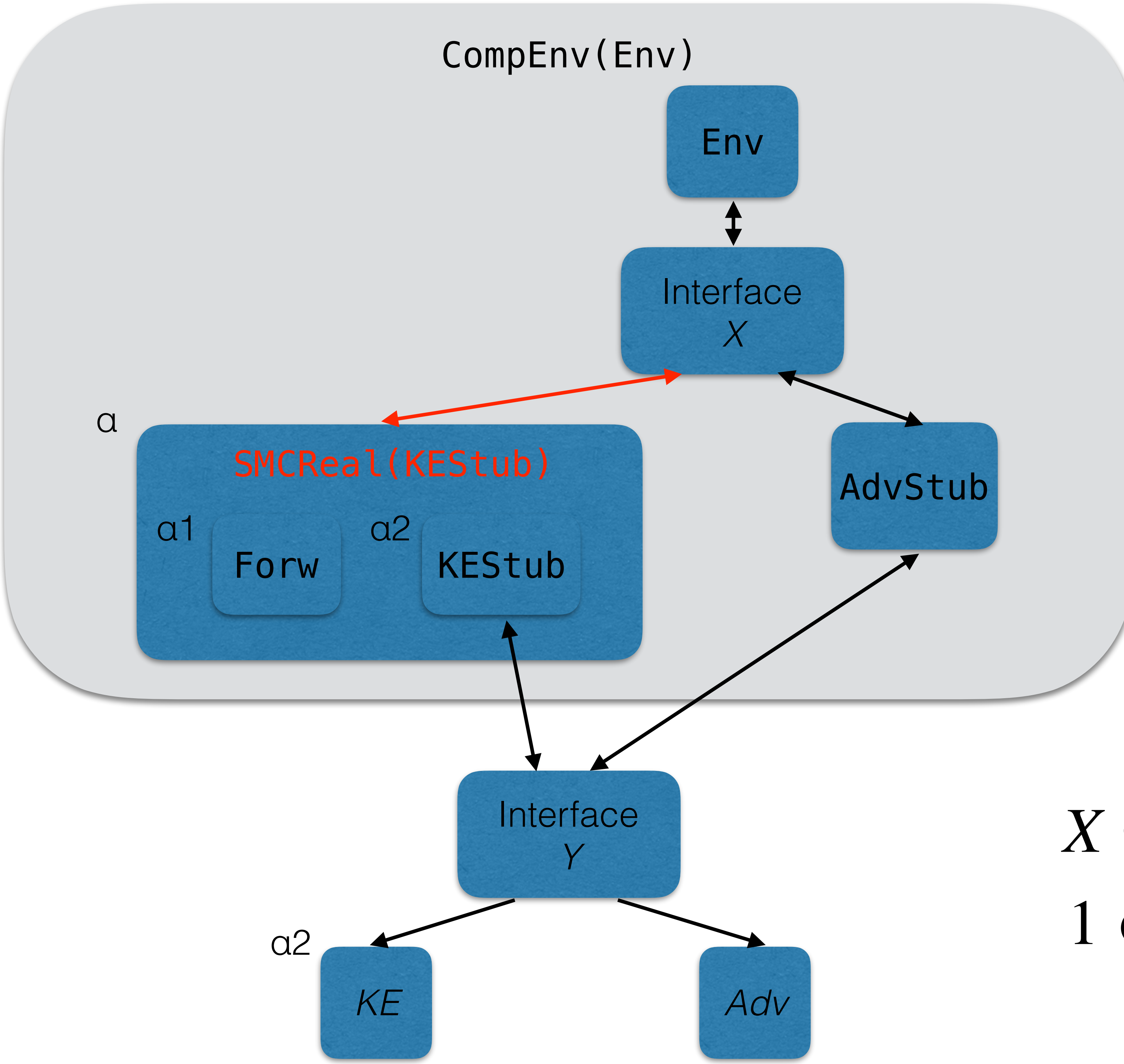
Bridging Lemma for Composition Theorem



$X \subseteq Y$

$1 \in Y$

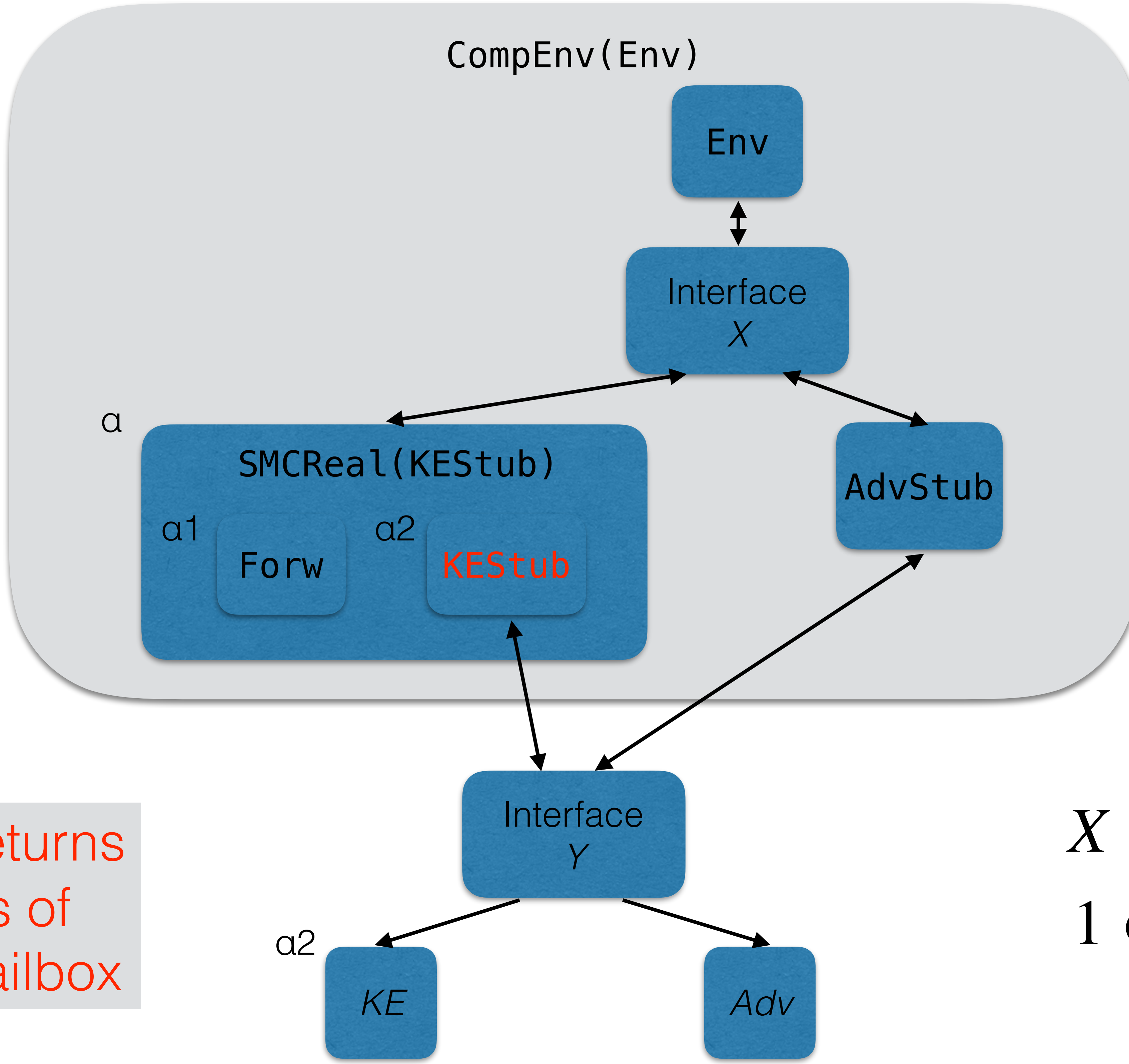
Bridging Lemma for Composition Theorem



$$X \subseteq Y$$

$$1 \in Y$$

Bridging Lemma for Composition Theorem



KEStub returns contents of shared mailbox

$X \subseteq Y$

$1 \in Y$

One-time Pad Step

- Next, we must define the SMC simulator `SMCSim`, and connect
  - `SMCReal(KEIdeal)`/`Adv’`
  - `SMCIdeal`/`SMCSim(Adv’)`

  where the input guard must exclude port index `3`
- This is done using EasyCrypt’s random sampling tactic
  - uses an isomorphism on the uniform distribution on exponents involving the plain text to be communicated
- We then apply the above when `Adv’` = `KESim(Adv)`

Overall Security Theorem

- Combining the instance of the composition theorem with the one-time pad step yields the connection between
  - `SMCReal(KEReal)`/`Adv`
  - `SMCIdeal`/`SMCSim(KESim(Adv))`

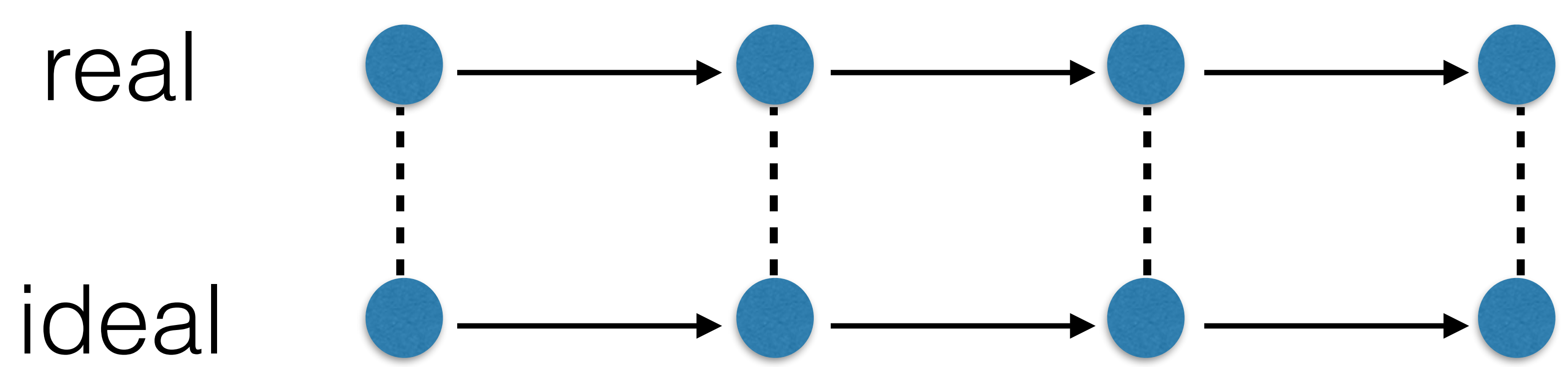
  where:
    - the input guard excludes `2` (from `KESim`) *and* `3` (from `SMCSim`)
    - the security upper bound is the DDH one, where the DDH adversary is applied to the composed environment

Lessons Learned

- SMC case study is complete, and validates our architecture and approach
- But it was too much work to scale-up to more realistic systems without some improvements to EasyCrypt and supporting tools

Relational Invariants/Symbolic Evaluation

- Proofs use *relational invariants* allowing the related evolutions of real and ideal games to be tracked
- Since the real and ideal worlds are structurally dissimilar, this means doing a lot of *symbolic evaluation*, essentially running code via tactics
- We have proposed and are implementing a way of automating this



Realization of UC Composition Theorem

- In our case study, we proved an instance of the UC Composition Theorem, via the definition of the composed environment and bridging lemmas
- We are now generalizing this work, producing a generic version of these definitions/proofs
- To obtain needed instances of the composition theorem, we'll then instantiate the generic definitions/proofs, and automatically generate some additional bridging definitions and proofs

Update from Paper

Dummy Adversary Lemma

- The same relational state may hold in two situations when the adversary is called:
  - when the adversary was called after the state was *first established*; or
  - when the adversary was invoked by the environment at stage when the state *already held*
- See the paper for how we currently unify these two cases in our proofs
- But we are working toward an improvement in which the user can think they are working in the so-called dummy adversary model — i.e., with an adversary that acts as instructed by the environment

Expressing Functionalities

- Defining real and ideal functionalities and simulators involves low-level message-routing code
- This boilerplate can be automatically generated, given domain specific language (DSL) for expressing functionalities and simulators
- DSL will be allow crypto theorists to more easily write and understand functionalities and simulators
- DSL type-checking will catch errors like badly formed messages, e.g., ones with bad source addresses
- Short term: translate DSL into existing EasyCrypt
- Longer term: integrate it into EasyCrypt

Conclusions

- The successful completion of our case study shows the validity of our UC in EasyCrypt architecture and approach
- But extensions and improvements to EasyCrypt and supporting tools will be needed for the approach to scale-up to realistic systems
- The EasyCrypt code for our case study, and a link to the extended (ePrint) version of our paper are available on GitHub:

`github.com/easyuc/EasyUC`