Fault Handling and Transactions in Jolie

Ivan Lanese
Computer Science Department
University of Bologna/INRIA
Italy

Faults

- Unexpected events can happen during program execution
  - Wrong user input
  - Not enough memory
  - File not found
- A fault signals an abnormal situation that forbids the normal continuation of the computation
- Frequently related to interactions with the environment (operating system, user, ...)

Faults in distributed computing

- Faults more common in distributed systems
- Network communication is unreliable
  - Lost messages
  - Malformed or unexpected message
- Fault recovery may involve different participants
  - Need to restore distributed invariants
  - E.g., online payment: total money should be preserved

Faults in SOA

- Faults even more common in SOA
- Endpoints under control of different entities
  - Endpoints may disappear
  - Services may be changed without notice
- Fault recovery fundamental for safe service composition
- Most standards and frameworks for SOA support error recovery
  - SOAP, WS-BPEL, Jolie, ..
- Emphasis on remote fault notification

Fault handling

- Faults must be managed
- The whole application should not fail even if some unexpected event happens
- Jolie offers dedicated primitives for error handling
  - To raise faults
  - To catch and manage faults
  - To notify faults
  - To compensate activities

Transactions

- Fault handling based on the concept of transaction
- Services perform distributed computations
- We want distributed computations to be successful, or at least avoid doing damages
- Transactions (from databases) are a programming construct ensuring this
- Transactions are computations that either
  - Commit, by successfully terminating their task, or
  - Abort, having no visible effect
- Transactions implemented by
  - Undoing effects in case of abort
  - Locks to avoid other processes to see effects before commit

ACID Transactions

- Classical transactions have the following ACID properties
- Atomicity: transactions are either executed completely, or have no visible effect at all
- Consistency: transactions should not violate state invariants
- Isolation: interleaved execution of multiple transactions should give the same effect of executing them in a (not specified) sequential order
- Durability: after commit, effect of transactions should be resistant to failures

Long running transactions

- ACID transactions impossible to obtain in SOAs
  - Message sendings cannot be undone
  - Cannot take the lock on the whole system
- Long running transactions are computations that either
  - Commit, by successfully terminating their task, or
  - Abort, having their visible effect compensated
- Compensating allows the system to
  - Reach a consistent state
  - Possibly slightly different from the initial one
- We will see some basic mechanisms Jolie provides to manage errors
  - We will also do an exercise on how to implement long-running transactions with them

Fault handing mechanisms

- Faults terminate the current activity and trigger recovery activities specified by suitable handlers
- Fault handler: executed to manage a local fault
  - similar to Java catch clause
- Termination handler: executed to smoothly terminate an activity because of a failure in a parallel process
- Compensation handler: executed to (partially) undo the effect of an already completed activity in case of abort
  - Its effect may become undesired
  - If I fail to book a flight for a trip, I want to undo the booking of the corresponding hotel

Fault handling in Jolie: scopes

- Code boxed into named scopes
  - Define the boundaries of error handling activities
    scope(name){...}
  - Provide a hierarchical structure
  - The program main is actually a scope named main

Fault handling in Jolie: throwing faults

- Some faults are generated by service invocations:
  - IOException for errors in communication
  - CorrelationError for errors in correlation
  - TypeMismatch for type errors
- These are system faults
- Faults can also be thrown: throw(fname, data)
  - Data contains information on the fault
- This allows for logical faults
- Information on faults available in a variable with the name of the scope
  - Variable scopename.faultname contains fault specific information

Fault handling in Jolie: fault handlers

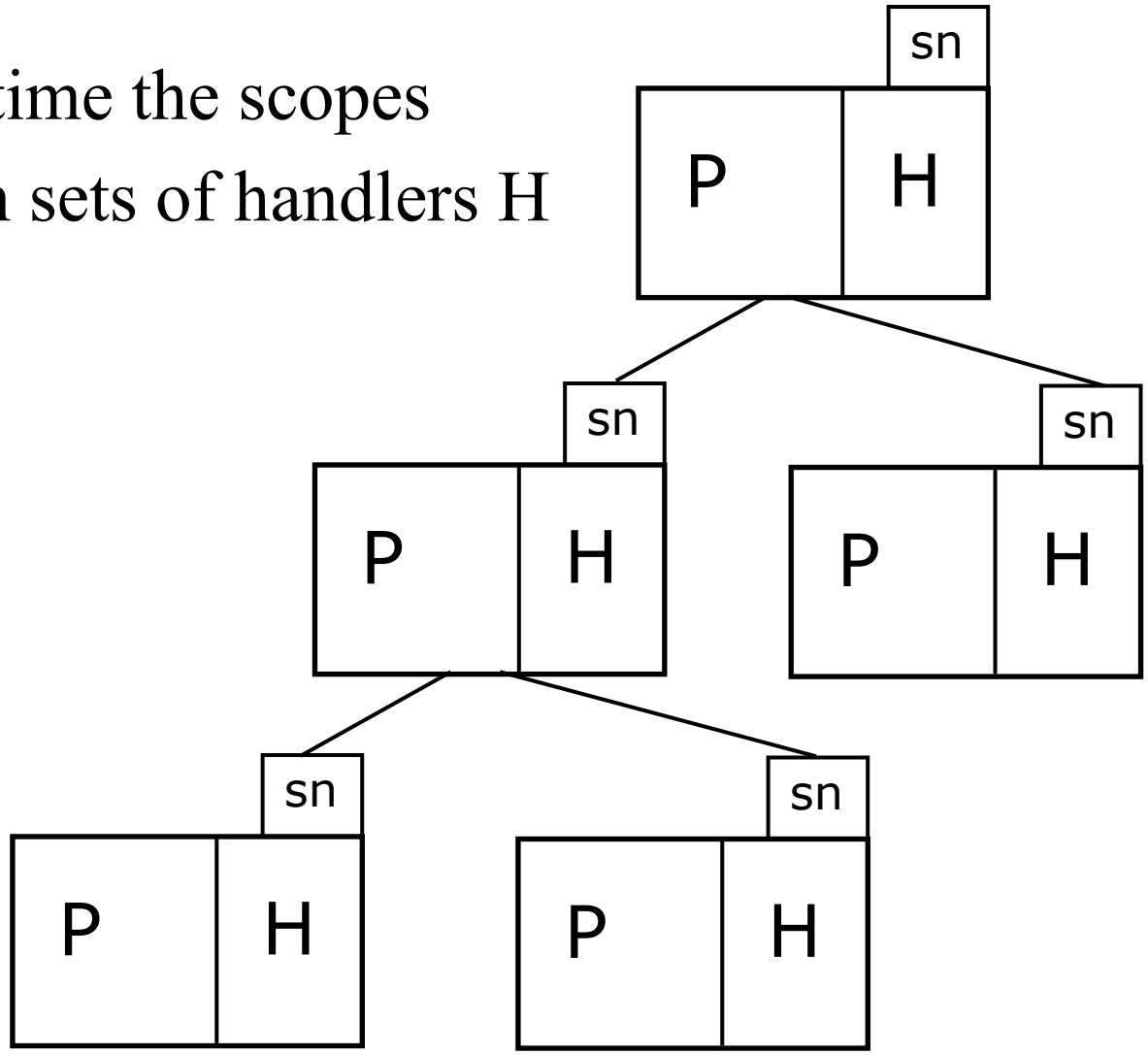
- Handlers to manage faults can be installed:
  install (fname1 => ..., fname2 => ..., fname3 => ...)
- A default fault handler can be installed
  install (default =>...)
- When a fault occurs, the corresponding handler is executed

Termination handling in Jolie

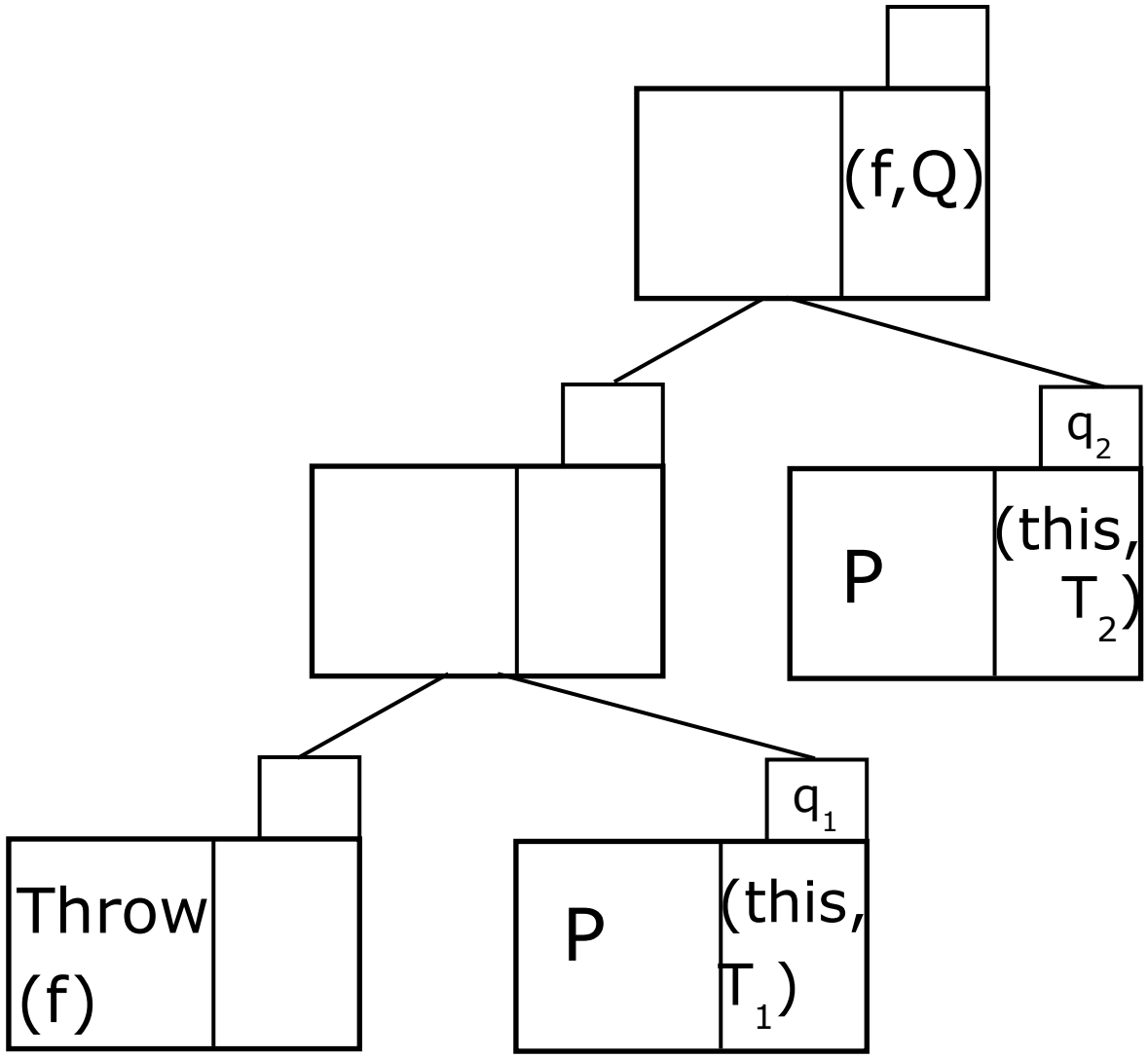
- A fault terminates parallel scopes
- We may want to execute some code before terminating to ensure smooth termination
  - Free resources
  - Undo partial activities
- Termination code is specified using termination handlers
  - install(this => ...)

The scope hierarchy

At runtime the scopes contain sets of handlers H

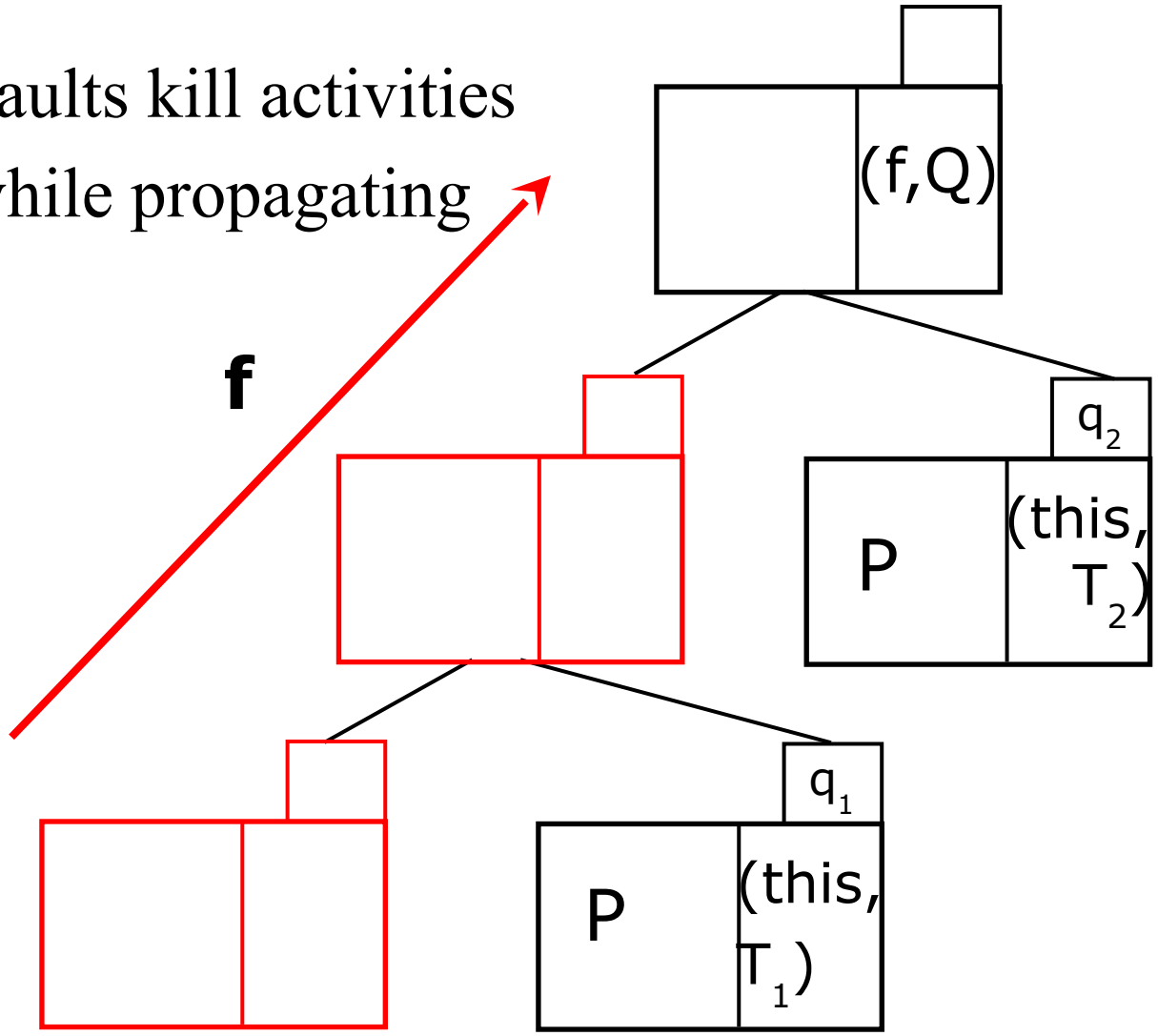


Throwing a fault

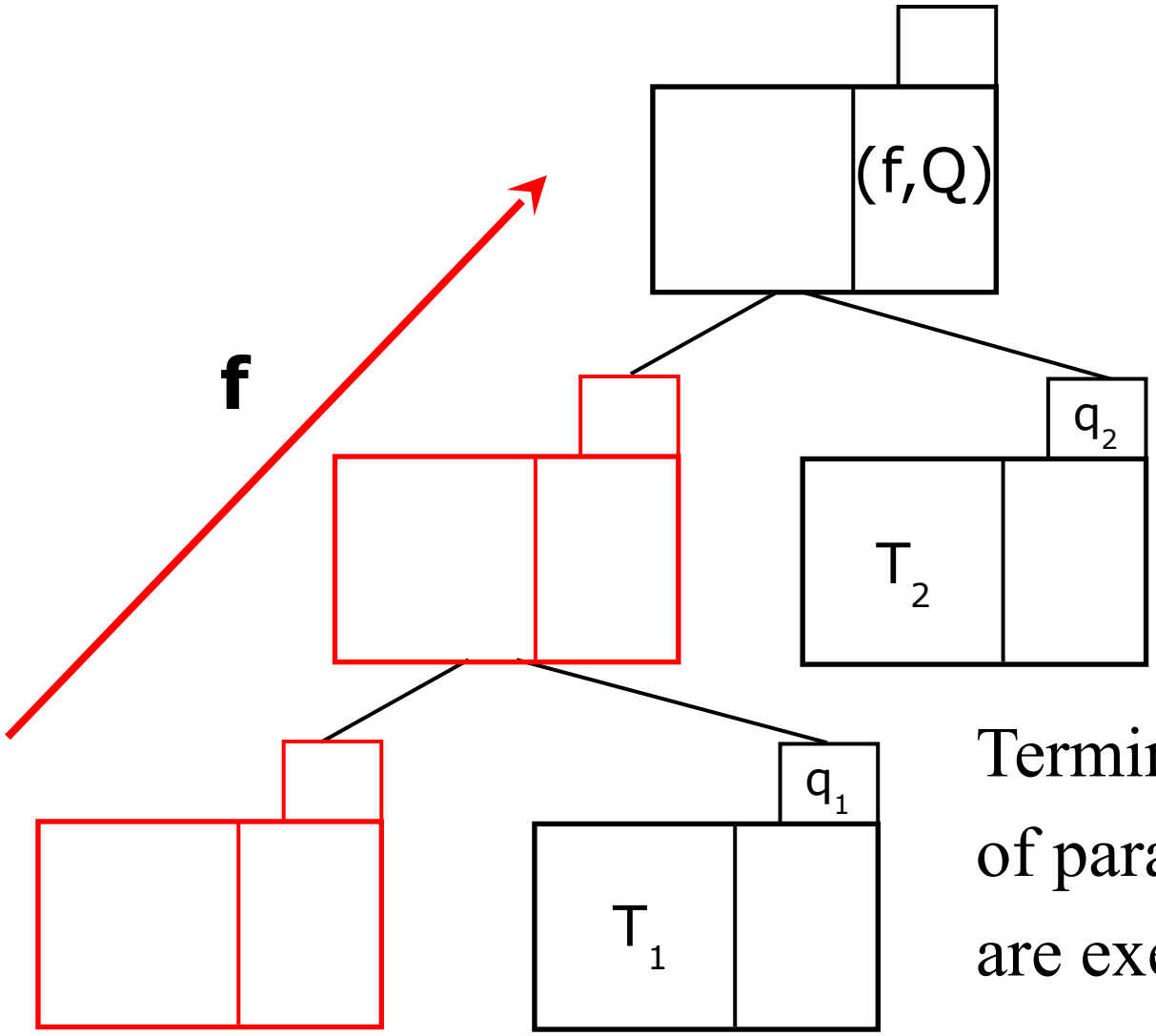


Throwing a fault

Faults kill activities while propagating

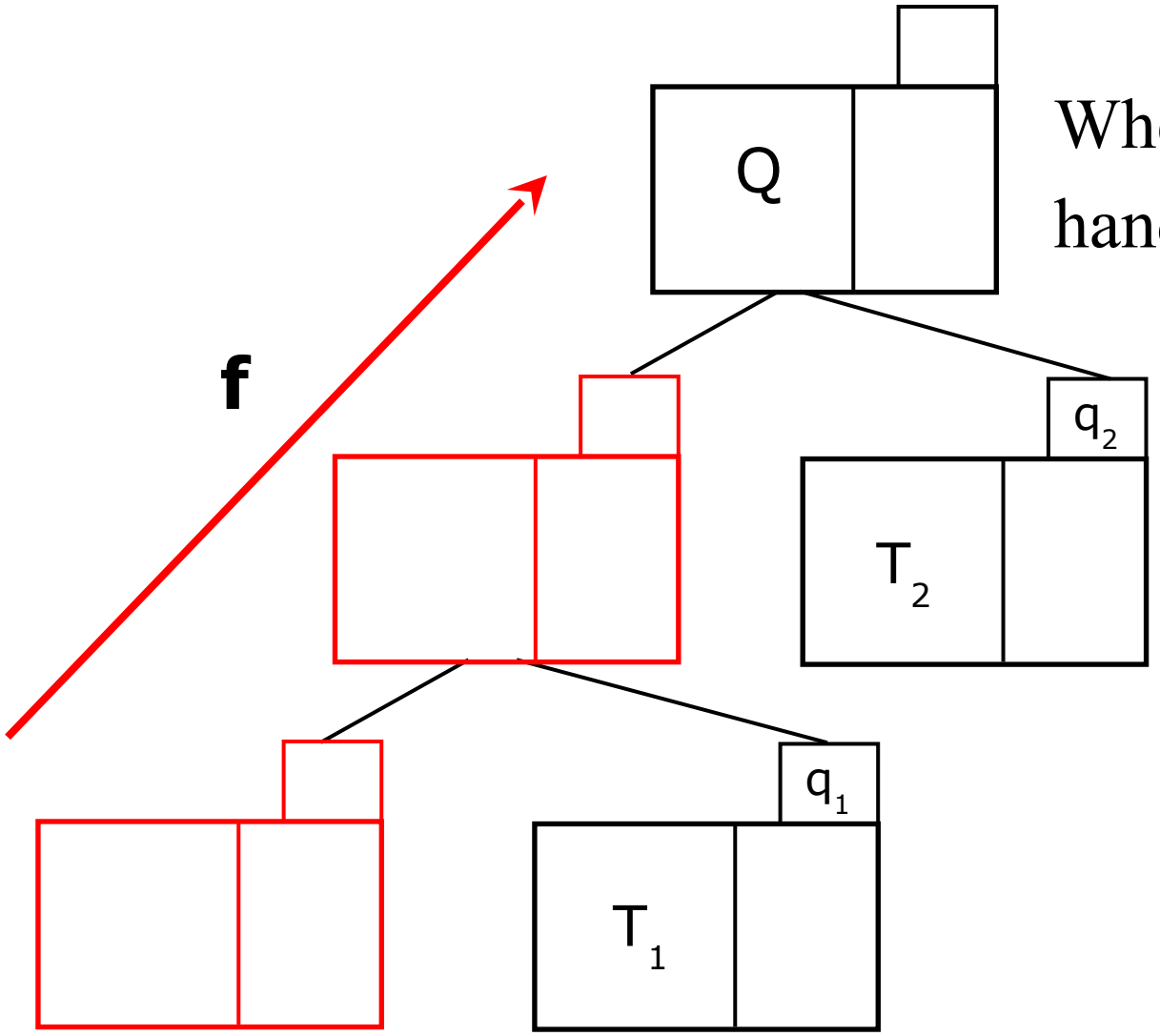


Throwing a fault



Termination handlers of parallel activities are executed

Throwing a fault



When found, the fault handler is executed

Interplay between fault recovery activities

- Concurrent faults may happen
- Jolie specifies what happens with concurrent fault recovery activities
- Recovery activities cannot be killed by other faults
  - Error recovery activities are always completed
- But termination overrides fault handling
  - Global errors more important than local ones
- After having been killed, a scope smoothly terminates
  - Ongoing communications are terminated
  - No more faults can be thrown

Compensation handlers

- Allow to undo the effect of an already completed activity
  - The fault handler of a purchase activity could ask to annul a previously done payment
- When a scope terminates, its termination handler becomes its compensation handler
- Compensation handlers have to be explicitly invoked
  - Primitive comp(scopeName)
  - Available only inside handlers
  - Only child activities can be compensated

Dynamic fault handling

- In all the languages we are aware of handlers are statically defined while programming
  - Java throw … catch …
  - BPEL handlers
- Not always easy to write the desired handler

Example

- scope (q){
  throw(f) |
  while (i <100) {
  if i%2==0 then {P} else {Q};
  i=i+1
  }
  }
- We want to compensate each completed execution of P and Q in the reverse order of execution
  - This is called backward recovery
- We need auxiliary variables to track the executions of P and Q
  - Complex and error-prone

Atomicity problem

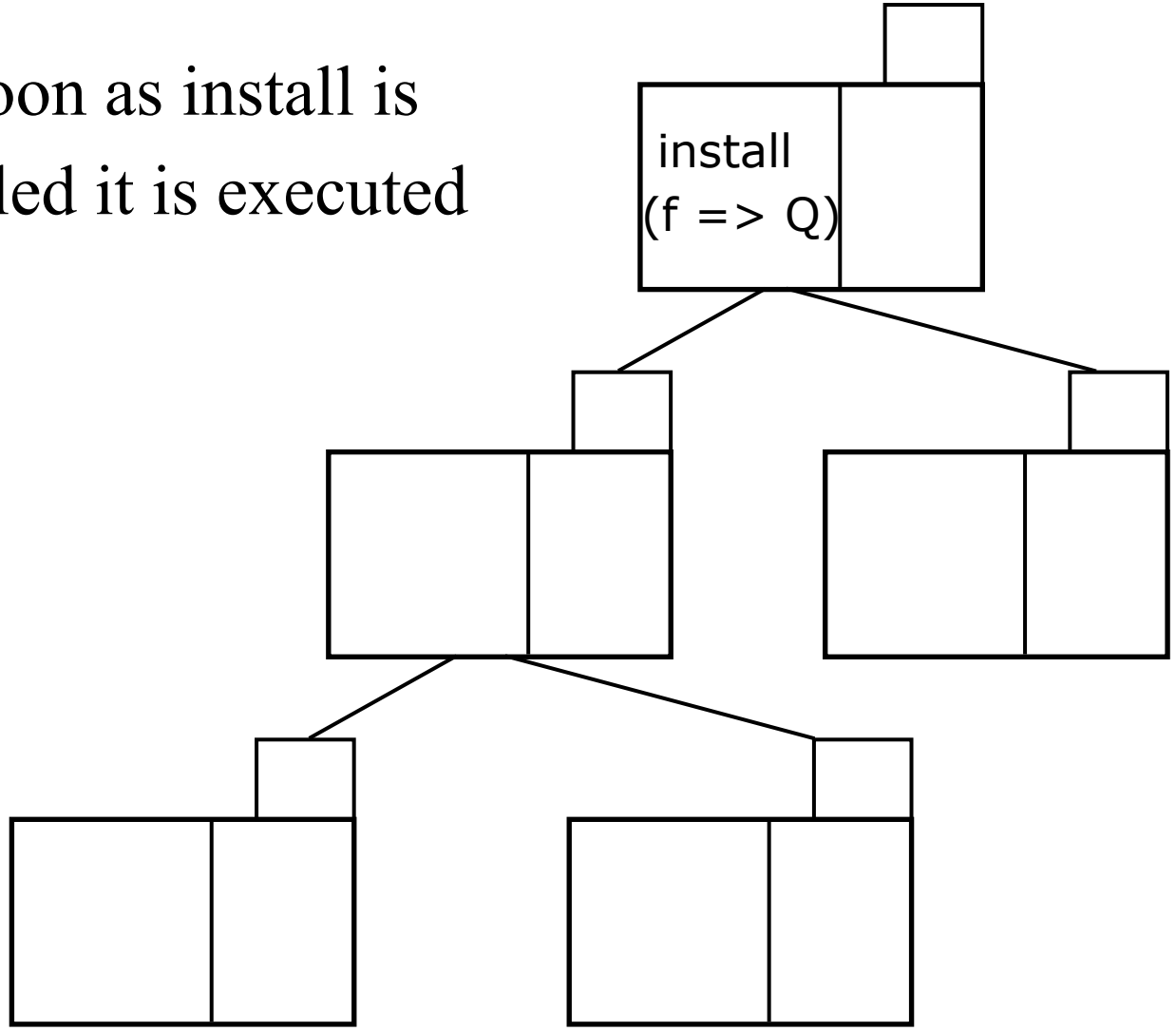
- When one should update the auxiliary variables?
- If I update the variables after the activity has completed
  - It may happen that P has been executed but the auxiliary variables have not been updated yet
  - If a fault occurs then the last execution of P is not compensated
- If I update the variables before the activity has completed
  - It may happen that P has not been completed but the variables log its execution
  - If a fault occurs then one additional execution of P is compensated

The Jolie solution

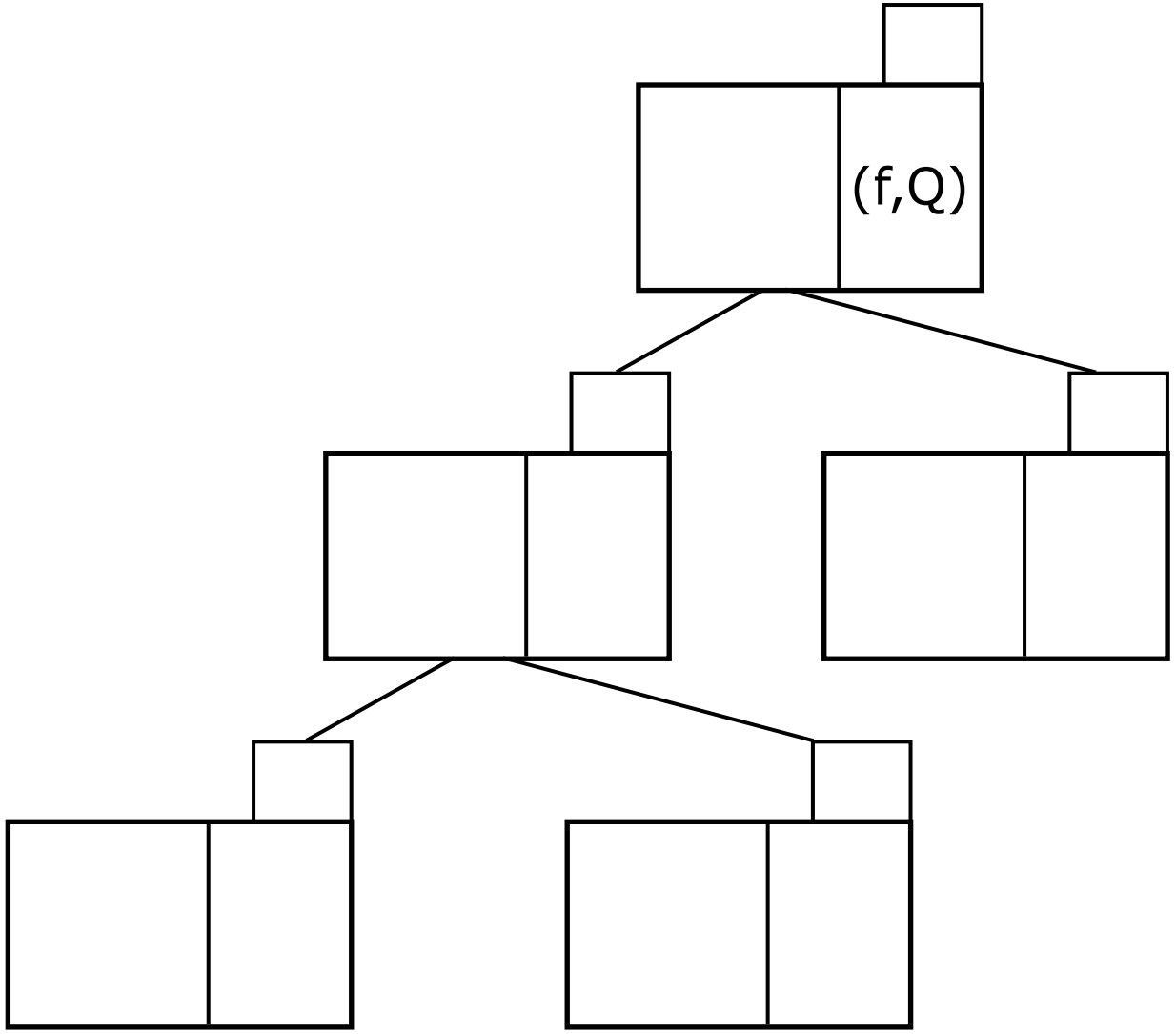
- scope (q){
  throw(f) |
  while (i <100) {
  if i%2==0 then
  {P; install(f => P';cH) }
  else {Q; install(f => Q';cH)}
  i=i+1;}}
- P' compensates P, Q' compensates Q
- The handlers are dynamically updated
- cH (current handler) is replaced by the previous handler
- install has higher priority than fault execution
  - No atomicity problem

Installing an handler

As soon as install is enabled it is executed



Installing an handler



Installation-time variable evaluation

- Sometimes one needs to remember the value of a variable when a handler has been installed
  - In the P, Q example one may want to write the number of the iteration being recovered
- The ^ operator freezes the value of the variable
  - If var is a variable an occurrence of ^var inside an handler is replaced by the value of var at handler installation time

Fault handling and request-response

- Request-response is a long lasting interaction
- Faults on one side influence the other side
- Two possibilities:
  - Faults on server side during the interaction
  - Faults on client side while waiting for the answer

Faults on server side

- A client asks a payment to the bank, the bank fails
- In BPEL the client receives a generic “missing-reply” exception
- In Jolie
  - The exact fault occurred at the bank is notified to the client
  - The notification raises the same fault on the client side
  - Suitable actions can be taken to manage the remote fault
- Faults propagated to the caller have to be declared in the interface
  RequestResponse: op throws faultName (type)
  - Type is the type of the data associated to the fault

Faults on client side

- A client asks a payment to the bank, then fails before the answer arrives
- In BPEL the return message is discarded
- In Jolie
  - The return message is waited for
  - The handlers can be updated according to whether or not a non-faulty message is received
  - The remote activity can be compensated if necessary

Exercise (simple)

- Extend the calculator client and server by adding an integer divide operation
- Pay attention to manage
  - DivisionByZero as a logical fault
  - IOException (system generated fault)

- Comment: why not managing DivisionByZero as simple message?
  - Separation of concerns

Exercise (more interesting)

- Extend the “buy with helper” example with the possibility for the seller to fail after he gets the money from both seller and helper: how could you ensure a compensation is performed?
- What if the seller can fail at any moment?