Exceptions

Fallimenti e risultati

A seconda delle istruzioni che vogliamo eseguire, potremmo aver bisogno di programmare operazioni che **falliscono**, ad esempio quando un valore si trova al di fuori di un intervallo, come in una divisione per 0 come denominatore o in una ricerca di un array con un indice che si trova al di fuori dei suoi limiti.

Oltre a poter **esprimere** queste situazioni nei programmi, dovremmo (dobbiamo) anche trovare il modo di **strutturarle**. L'idea è che le operazioni che dipendono dai risultati di quelle che falliscono siano consapevoli di questa possibilità — e i controlli statici del linguaggio possono aiutarci a gestirle.

Abbiamo già visto una soluzione a questo problema con i tipi monadici `Result`, che incapsulano e strutturano il passaggio/accesso dei risultati e degli errori.

Tuttavia, se da un lato i `Result` aiutano a rendere espliciti tutti i possibili errori, dall'altro possono diventare macchinosi da comporre quando si annidano le operazioni (ad esempio, quando si compongono una o più operazioni che non gestiscono direttamente i fallimenti, ma si limitano a "inoltrarli" al loro chiamante). Inoltre, i risultati costringono lo sviluppatore a codice più verboso (e difficile da consultare) per "aprire" il loro contenuto e poter agire su di esso.

Alternative ai tipi di Result

Naturalmente, esistono alternative (spesso meno raffinate/eleganti) dei risultati.

Una di queste consiste nel definire alcuni **valori eccezionali** che l'operazione utilizza per segnalare il fallimento al chiamante; ad esempio, potremmo avere un'implementazione grezza della divisione che utilizza 0 per indicare un fallimento. Naturalmente, questo metodo è piuttosto limitato, poiché, ad esempio, confondiamo le chiamate legittime e quelle che falliscono (ad esempio, 0/1 e 1/0).

Un altro modo è l'**inversione del controllo** tra chiamato e chiamante, dove una funzione che potrebbe fallire chiede al chiamante di passare, oltre ai suoi input ordinari, una funzione che chiama in caso di fallimento. Sebbene questo possa funzionare in linea di principio (solo nei linguaggi che supportano le funzioni come argomenti delle operazioni), rende più complesso il ragionamento sul flusso dei programmi e più oscuro l'uso delle operazioni (potremmo aver bisogno di sapere cosa fa l'operazione con la nostra funzione che gestione l'errore, per scriverla correttamente).

Eccezioni

Le **eccezioni** sono un'alternativa ai Result (e sono molto più popolari di questi).

L'idea alla base delle eccezioni è che una condizione eccezionale (un raro evento di fallimento) causa il **trasferimento** diretto **del controllo** a un **gestore di eccezioni** definito in qualche punto dello stack di chiamate del programma. Questo può trovarsi nell'operazione che chiama quella che fallisce, in quella che si trova sopra la prima o anche completamente assente, nel qual caso (se fatto di proposito) significa che non c'è nulla che il programmatore possa fare per recuperare l'eccezione e l'unica soluzione è interrompere il programma.

Ad esempio, per l'operatore di divisione, possiamo chiamare un’eccezione (irrecuperabile/unrecoverable) il fatto che il sistema esaurisca la memoria, mentre fallire con un denominatore a 0 è il suo comportamento caratterizzante.

Gestione delle eccezioni

C o Rust non forniscono un supporto diretto alla gestione delle eccezioni e gli utenti ricorrono a una delle alternative citate: ad esempio, Rust si basa principalmente su operatori dedicati ai tipi Result che riducono al minimo la lunghezza della base di codice.

Altri linguaggi, come Java, forniscono costrutti di gestione delle eccezioni che legano lessicalmente i **gestori di eccezioni** a blocchi di codice, che li sostituiscono nel caso in cui "catturino" un'eccezione.

In Java, le eccezioni permeano completamente il linguaggio, e.g., 1/0 solleva

```
java.lang.ArithmeticException: / by zero
```

Gestione delle eccezioni

Per gestire le eccezioni, Java (e linguaggi simili) fornisce il costrutto **`try-catch`**, ad es,

```
try {
 System.out.println( “Let’s try to divide by zero” );
 double x = 1 / 0;
} catch ( ArithmeticException exception ){
 System.err.println( “You shall not divide by zero!” );
}
```

Dove il blocco **try** racchiude il codice che potrebbe sollevare eccezioni e il blocco **catch** racchiude il comportamento che dobbiamo eseguire se catturiamo qualche eccezione (vedremo, più o meno specifica).

In particolare, il codice nel blocco **try** non viene eseguito "atomicamente". Infatti, il codice nel blocco catch sostituisce il codice nel blocco try solo a partire dall'istruzione che solleva (lancia/**throw**, in gergo Java) l'eccezione. Nell'esempio, l'output (sullo standard I/O) del nostro programma stampa la prima stringa, `"Let's try ..."`, seguita dalla seconda, `"You shall not ..."`.

Gestione delle eccezioni

Per gestire le eccezioni, Java (e linguaggi simili) fornisce il costrutto **try-catch**, ad es,

```
try {
 System.out.println( “Let’s try to divide by zero” );
 double x = 1 / 0;
} catch ( ArithmeticException exception ){
 System.err.println( “You shall not divide by zero!” );
}
```

Il blocco **catch** accetta un argomento: l'eccezione che si aspetta di intercettare.

In generale, per strutturare la gestione delle eccezioni, i blocchi catch **specificano le eccezioni che intercettano tramite nomi**, che spesso i linguaggi **consolidano coi simboli del sistema di tipi**.

In Java, questo mix migliora la flessibilità del costrutto **try-catch** attraverso il sottotipaggio: tutti i valori sollevabili (**throw**) e catturabili (**catch**) sono sottotipi del tipo speciale **Throwable**, che prende il nome dall'istruzione di sollevamento delle eccezioni di Java, **throw**. Nel nostro esempio, il blocco catch dichiara di aspettarsi di intercettare eccezioni del (sottotipo) ArithmeticException.

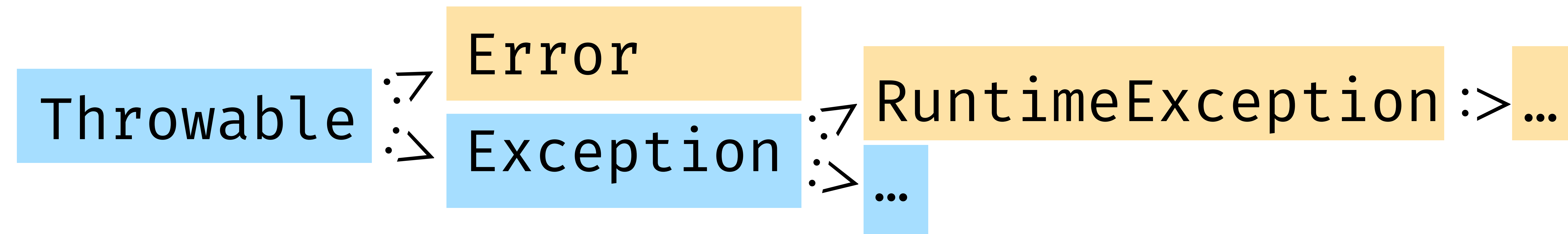
Gestione delle eccezioni e sottotipaggio

```
class MyException extends Throwable { … }

try {
 throw new MyException();
} catch ( MyException exception ){
 System.err.println( “Caught an exception!” );
}
```

In Java, gli sviluppatori raramente definiscono sottotipi diretti del tipo `Throwable`, ma utilizzano sottotipi dei due sottotipi di `Throwable` `Error` ed `Exception`.

Gestione delle eccezioni e sottotipaggio: Eccezioni Implicite vs Esplicite

La distinzione tra `Error` ed `Exception` è che il sistema di tipi di Java obbliga lo sviluppatore a gestire esplicitamente qualsiasi sottotipo di `Throwable`, tranne gli `Error` e un sottotipo speciale di `Exception`, chiamato RuntimeException.



Gli **`Error`** e le **`RuntimeExceptions`** rappresentano rispettivamente **fallimenti irrecuperabili a livello di runtime e di applicazione** che dovrebbero interrompere l'esecuzione del programma, quindi il programmatore dovrebbe gestirli solo se sa (che esistono e) come recuperarli.

Quindi, in Java le eccezioni non sono così ... eccezionali (come da definizione di "*rari casi di errore*") e sono il costrutto principale che il linguaggio fornisce per gestire gli errori.

Eccezioni Esplicite

Concentrandosi sulle Exception, Java obbliga gli sviluppatori a gestirle esplicitamente in due modi possibili: o dichiarando che un'operazione lancia un'eccezione o gestendo l'eccezione lanciata all'interno dell'operazione.

```
void o() throws MyException{
 throw new MyException();
}
```

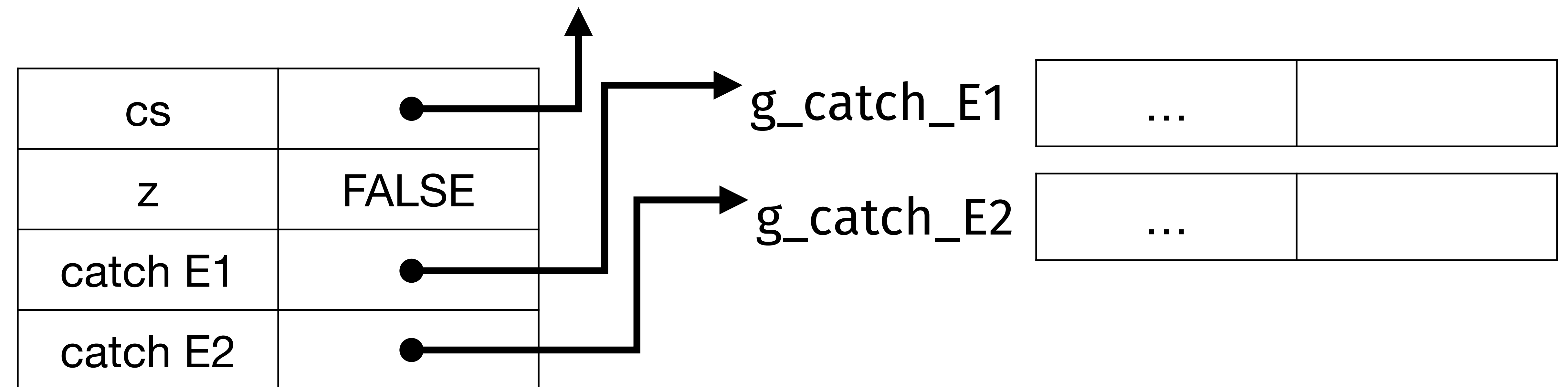
```
void o(){
 try {
  throw new MyException();
 } catch ( MyException e ){
  …
 }
}
```

Implementare il try-catch

```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
g();
```

exec. point

1

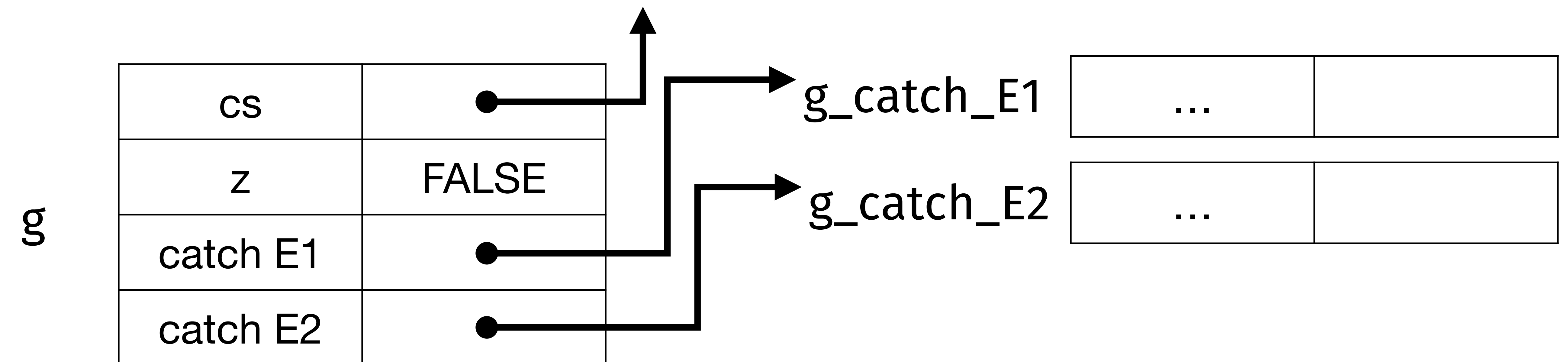


Implementare il try-catch

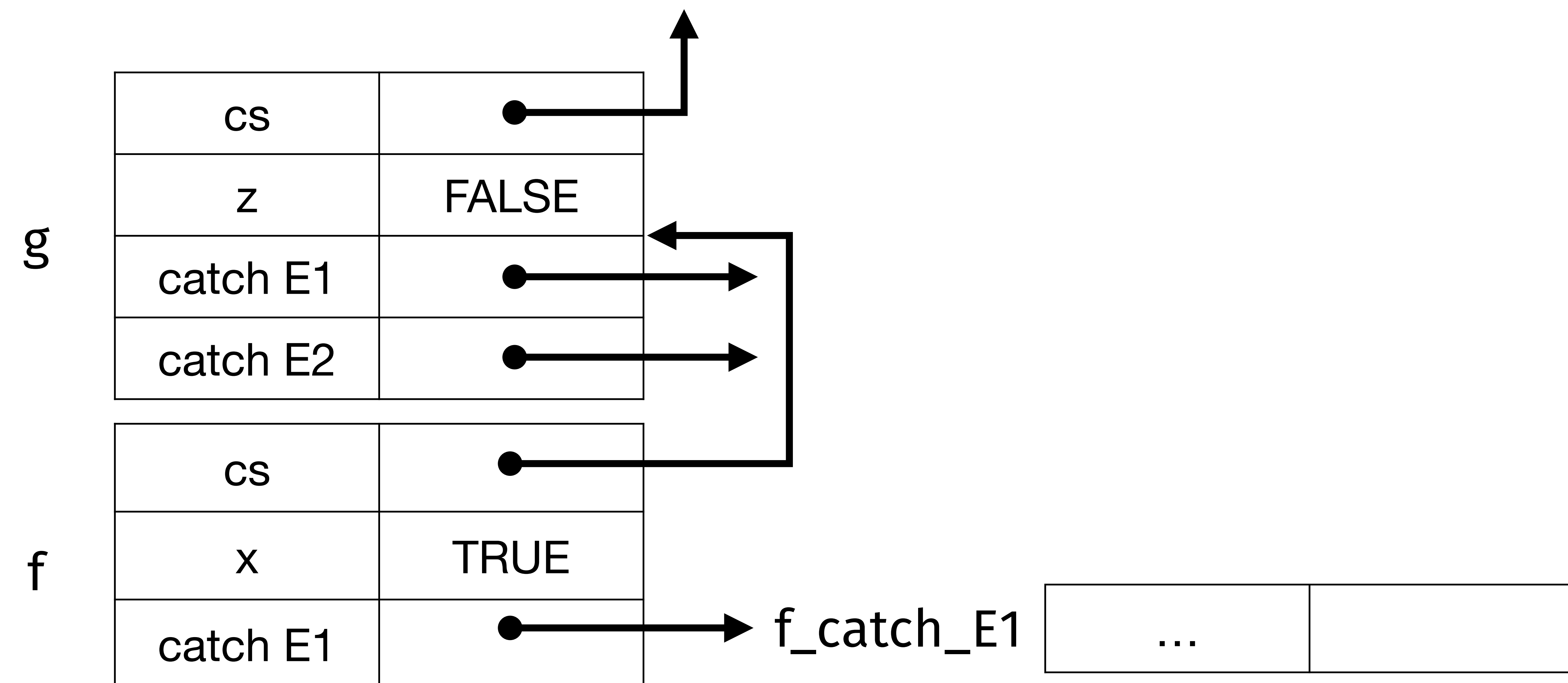
```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
g();
```

exec. point

1



2

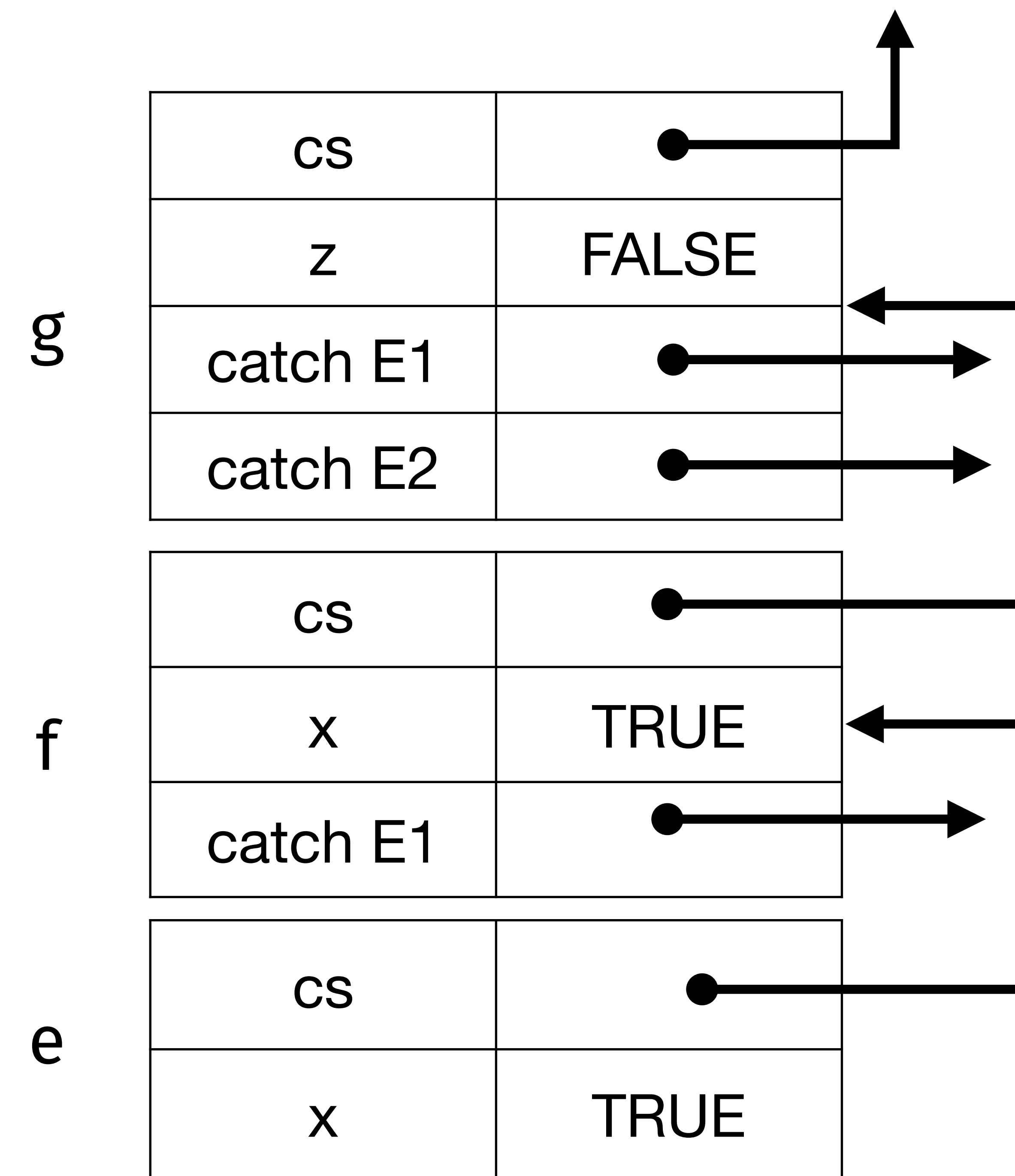


Implementare il try-catch

exec. point

```
void e() throws E
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
g();
```

3

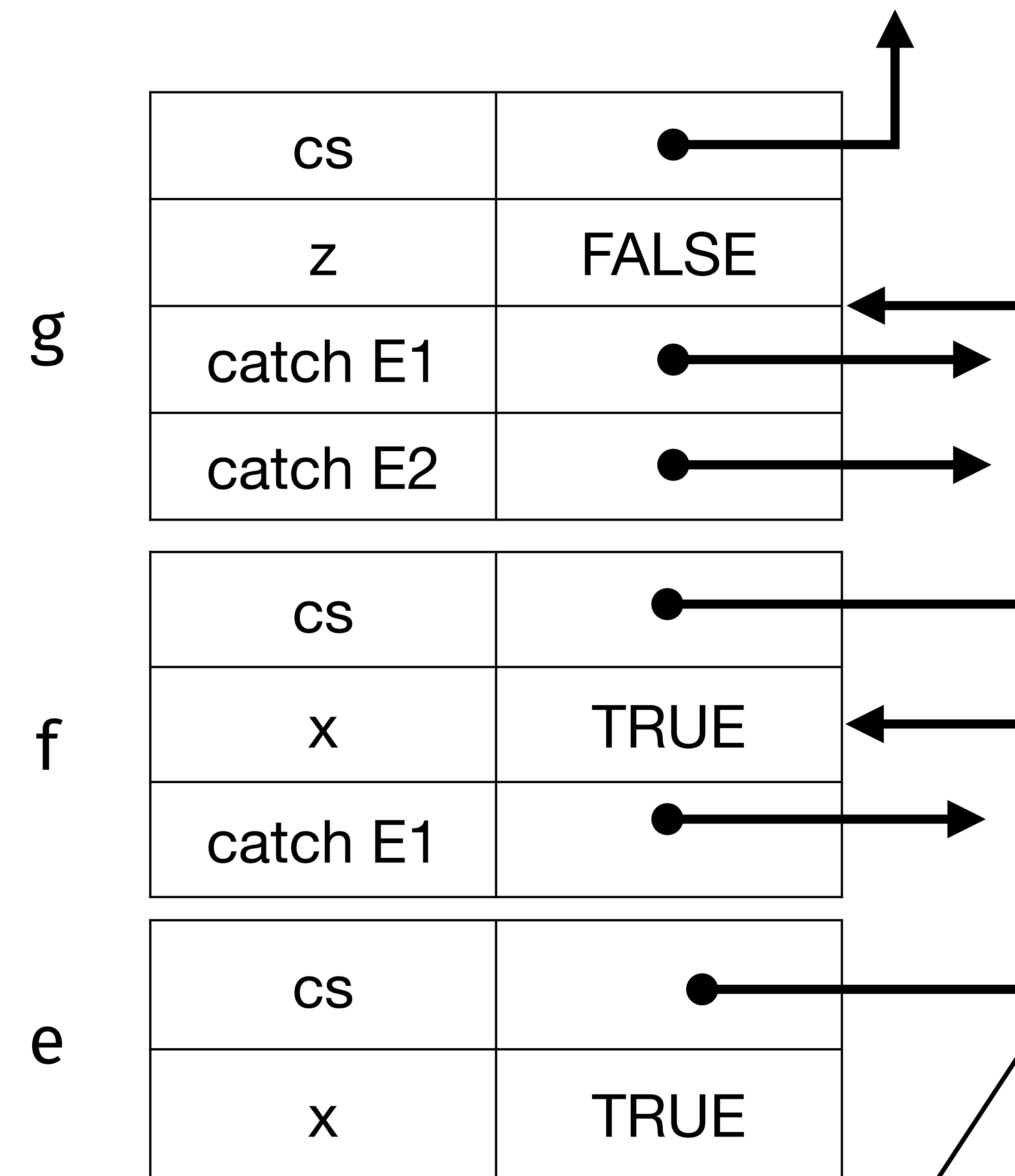


Implementare il try-catch

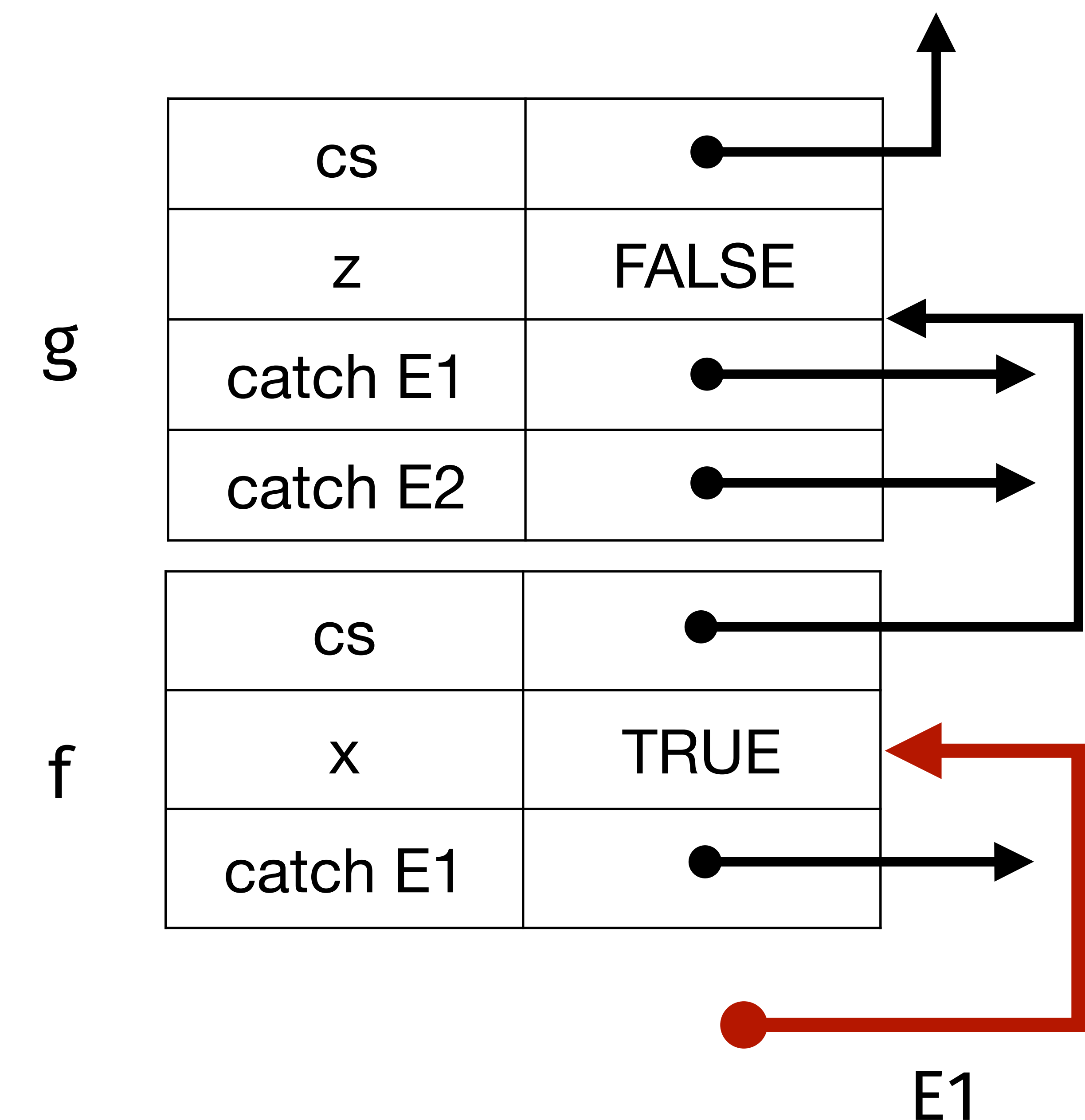
```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
g();
```

exec. point

3



4

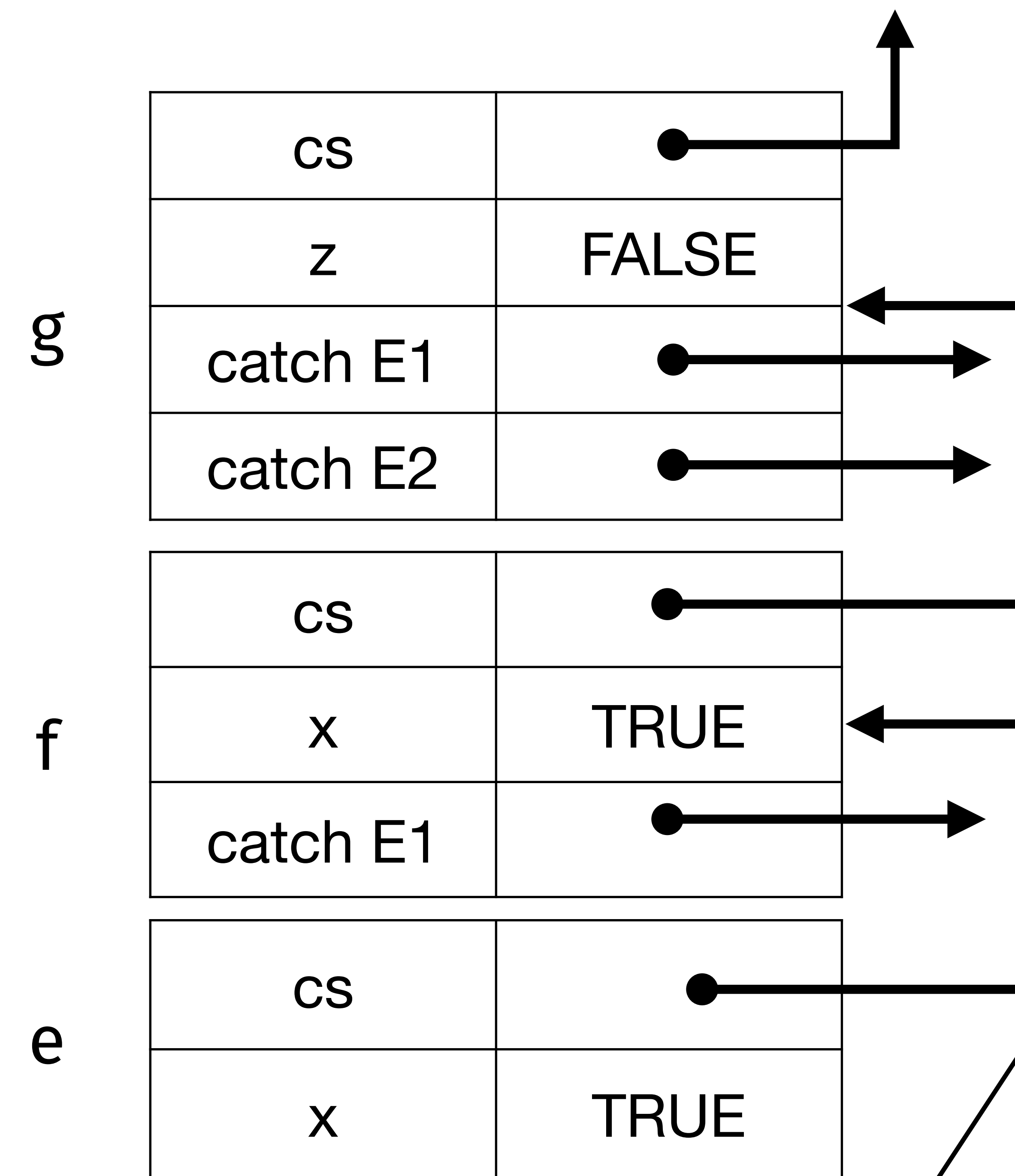


Implementare il try-catch

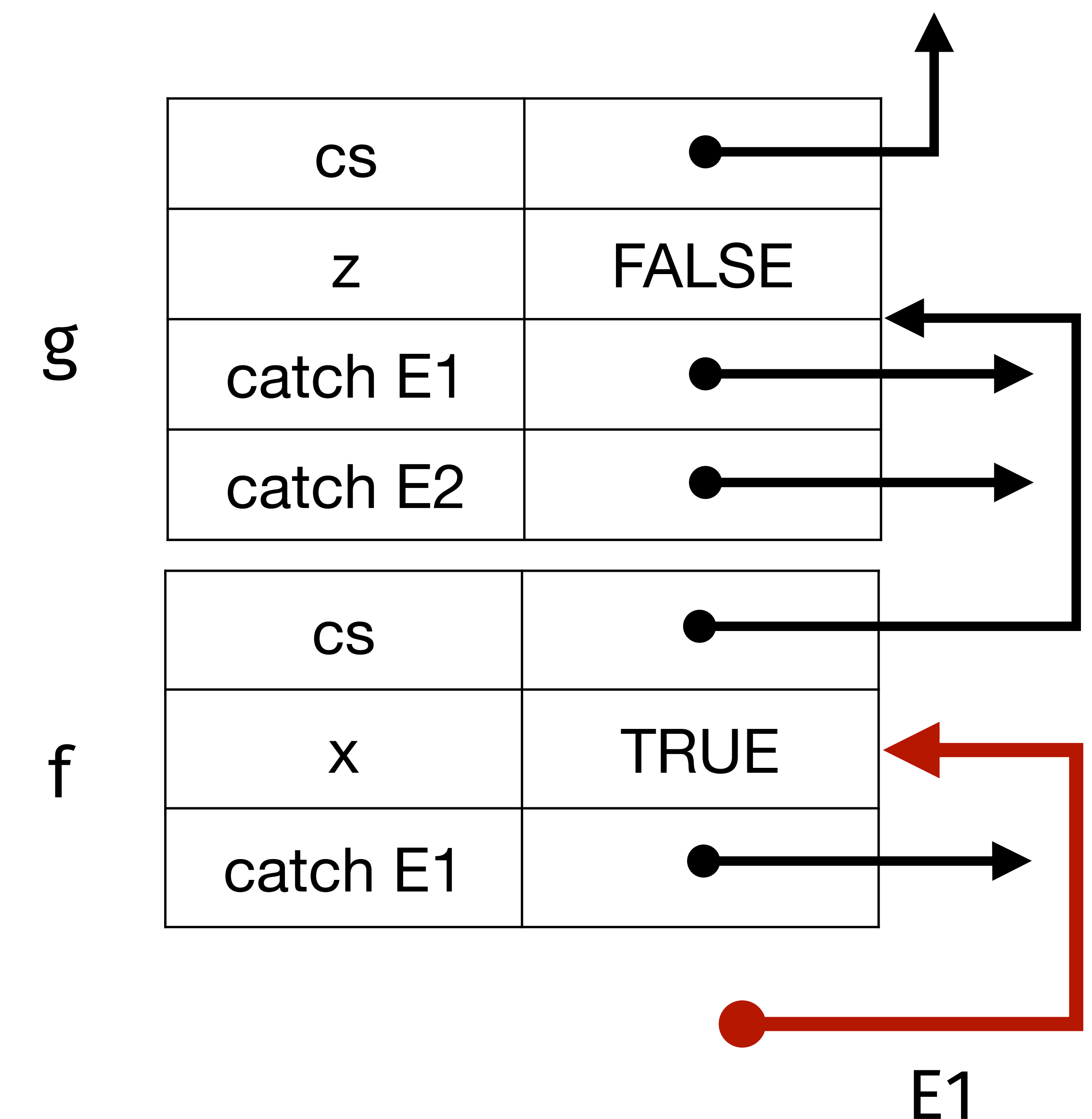
```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
g();
```

exec. point

3



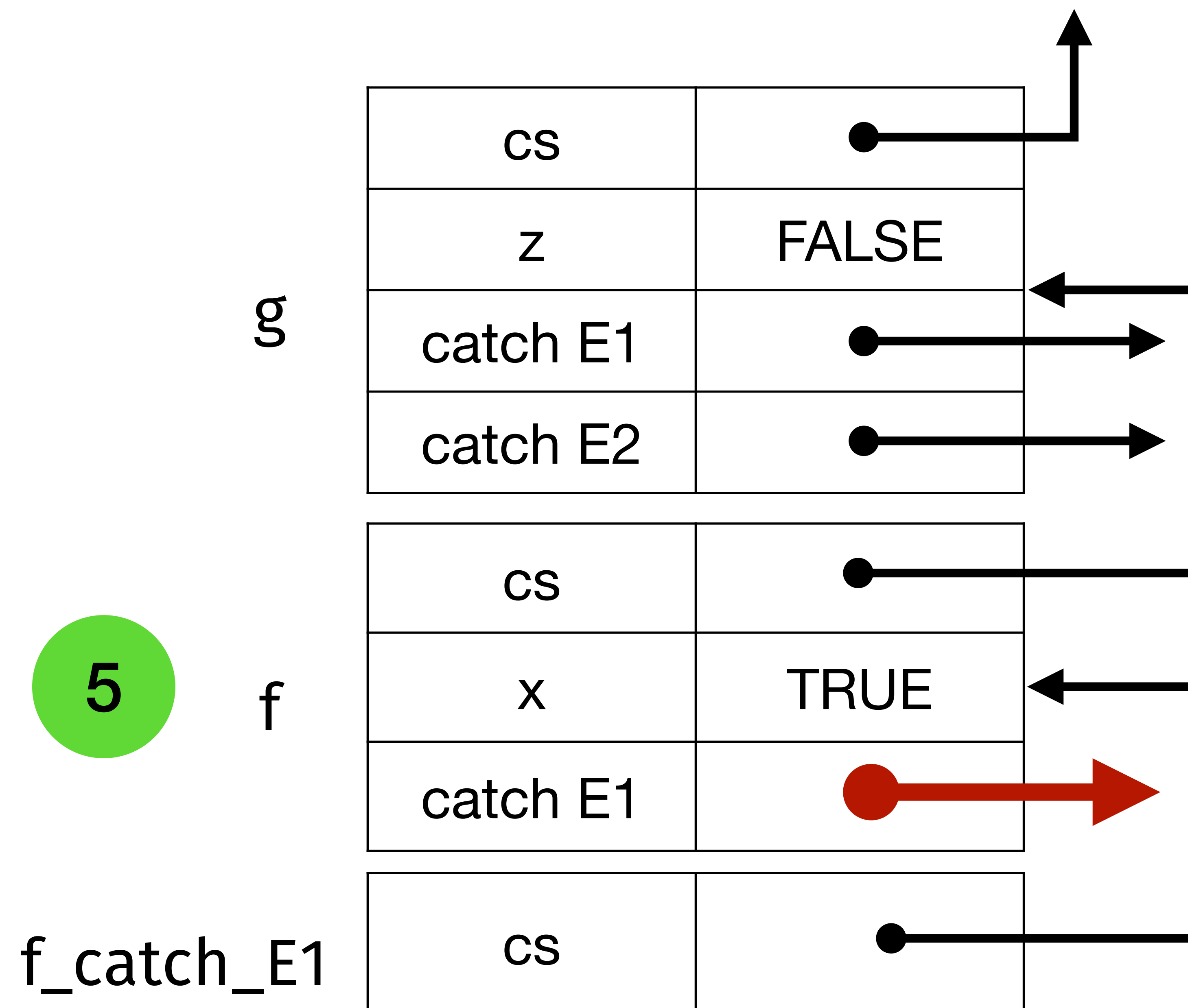
4



Implementare il try-catch

```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool [illegible] false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
g();
```

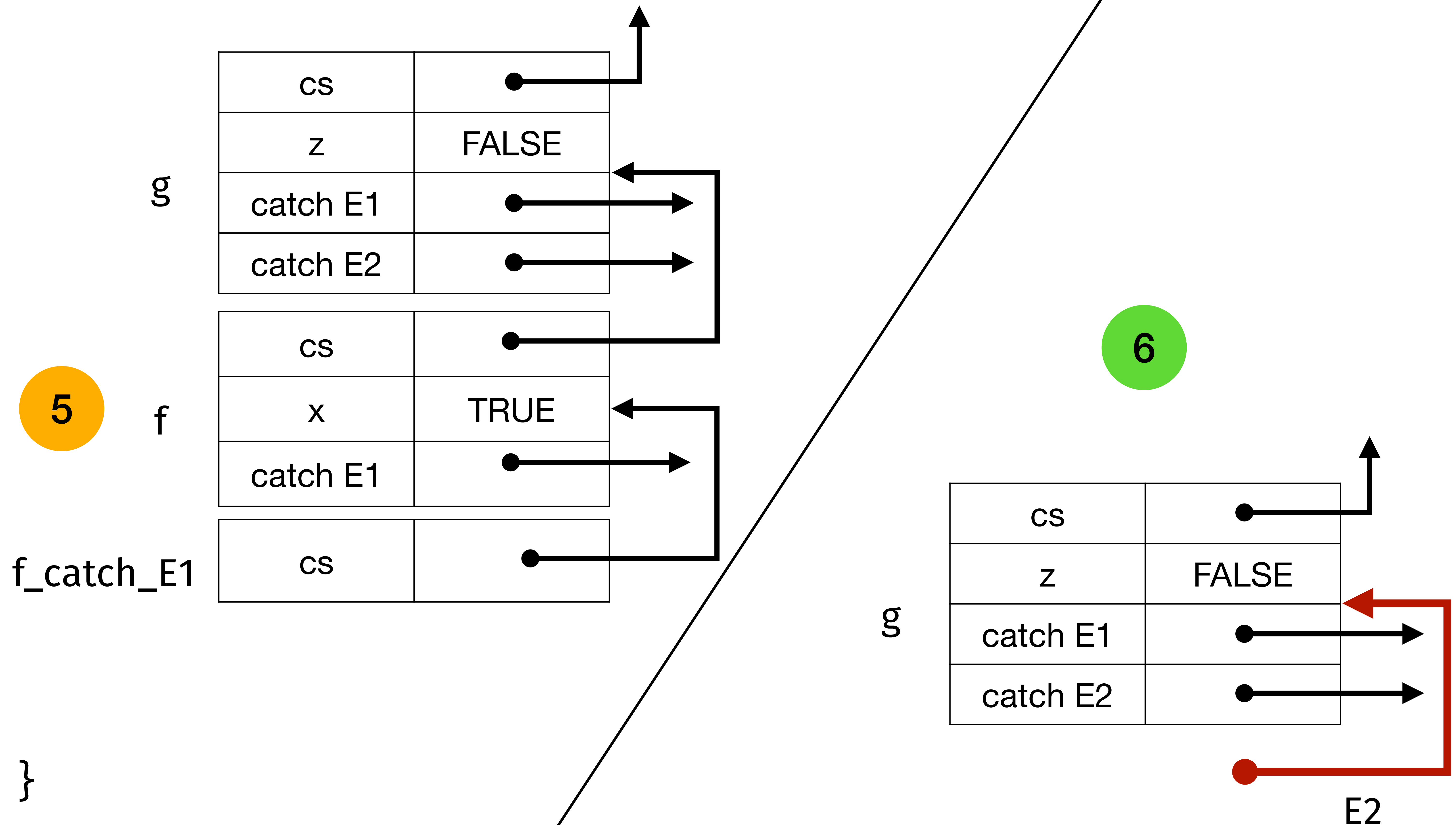
exec. point



Implementare il try-catch

```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
g();
```

exec. point

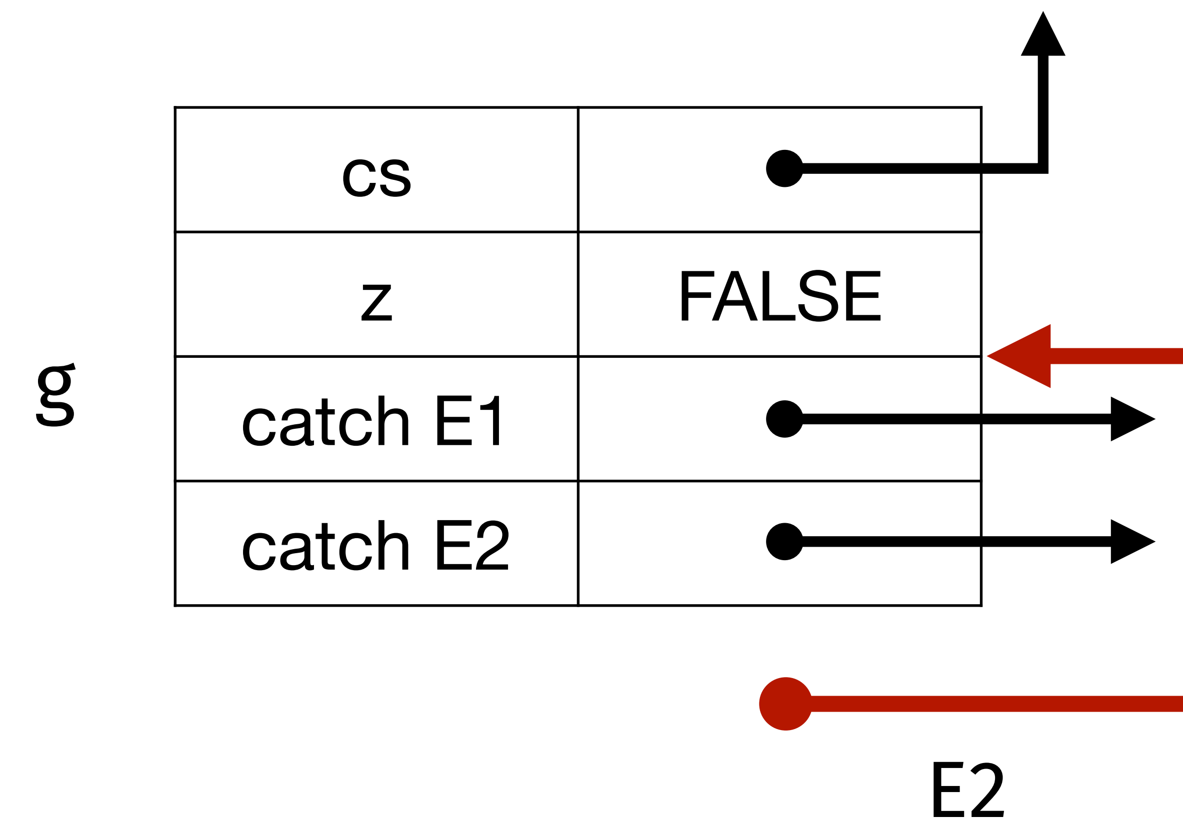


Implementare il try-catch

```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
g();
```

exec. point

7

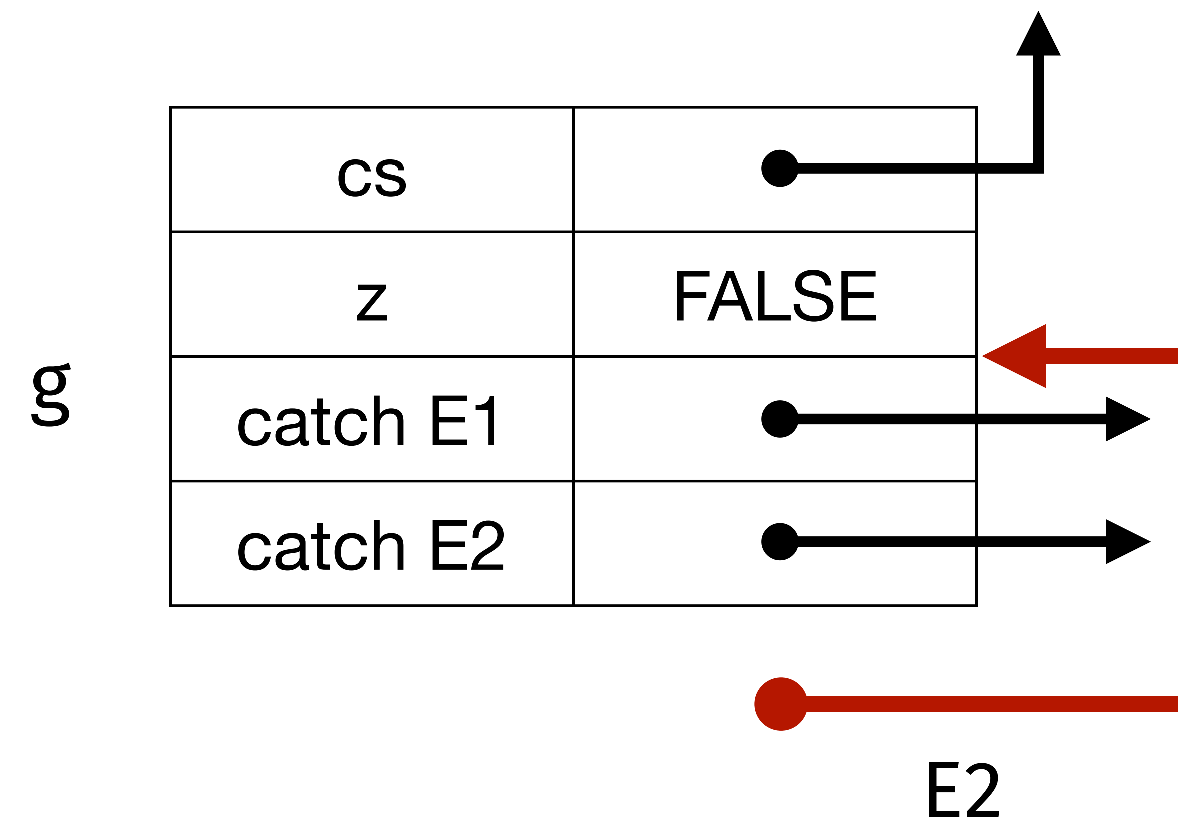


Implementare il try-catch

```
void e() throws E1 {
  bool x = true;
  throw new E1();
}

void f(bool x) throws E1, E2 {
 if( x ){
  try { e(); bool y = false; }
  catch( E1 e ){
    throw new E2();
   }
 } else {
    e();
} }

void g() {
 try { bool z = false; f( true ); }
 catch( E1 e ){}
 catch( E2 e ){}
}
g();
```

exec. point

7



8