Plan

Sémantique et implémentations

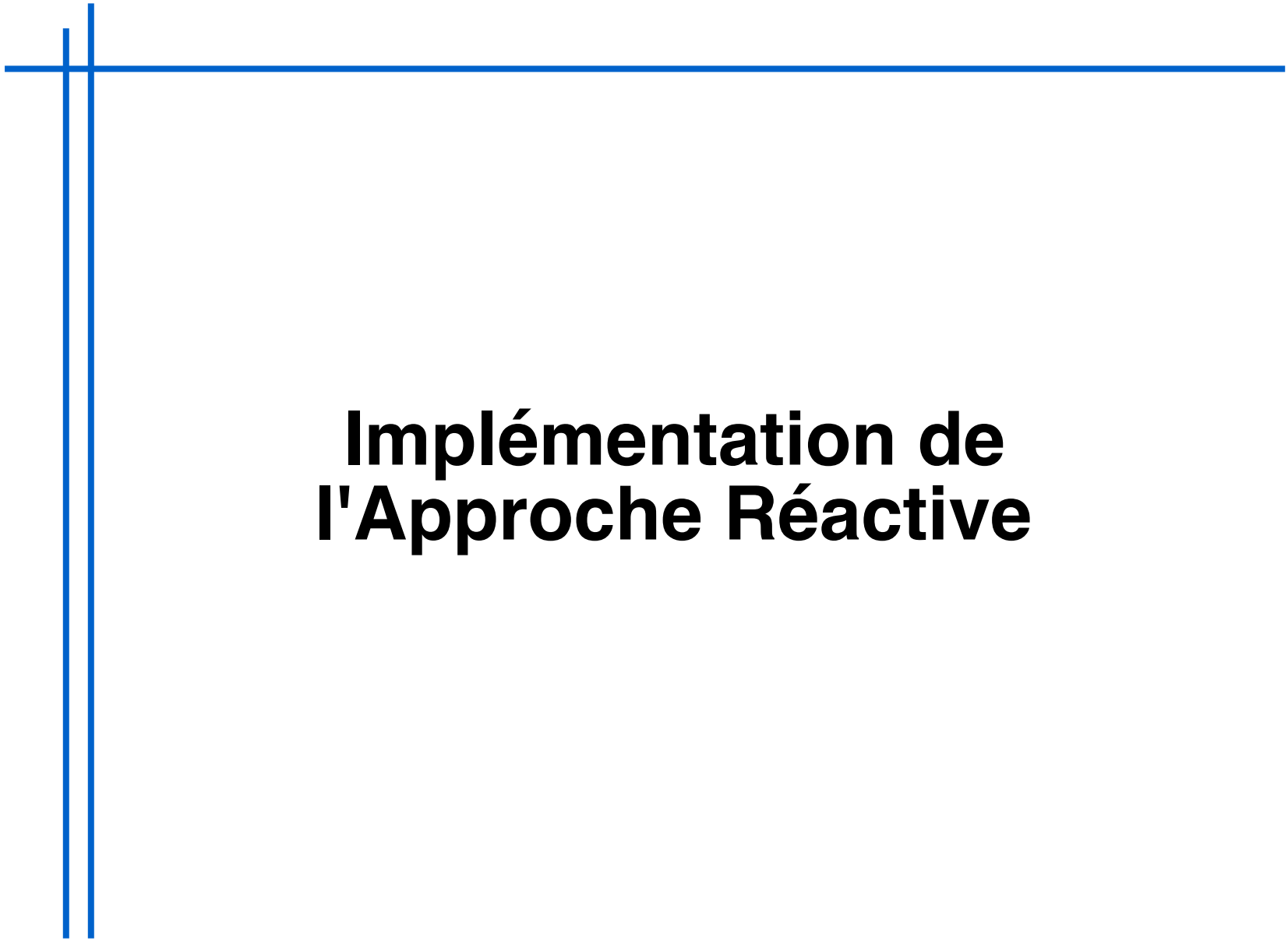
- **Rewrite/Replace**
- **Storm/Simple**

SugarCubes :

- **Objets réactifs : les Cubes**
- **Les scripts réactifs**

Distribution

- **Java RMI**
- **Migration**
- **Machines réactives distribuées**



Implémentation de l'Approche Réactive

Junior

Sémantique formelle
+
implémentations efficaces

Implémentation de référence :

-Rewrite (Sémantique formelle SOS)

Implémentation plus efficace :

-Replace (règle SOS dérivées de Rewrite)

Grand nombre de composants parallèles et d'événements :

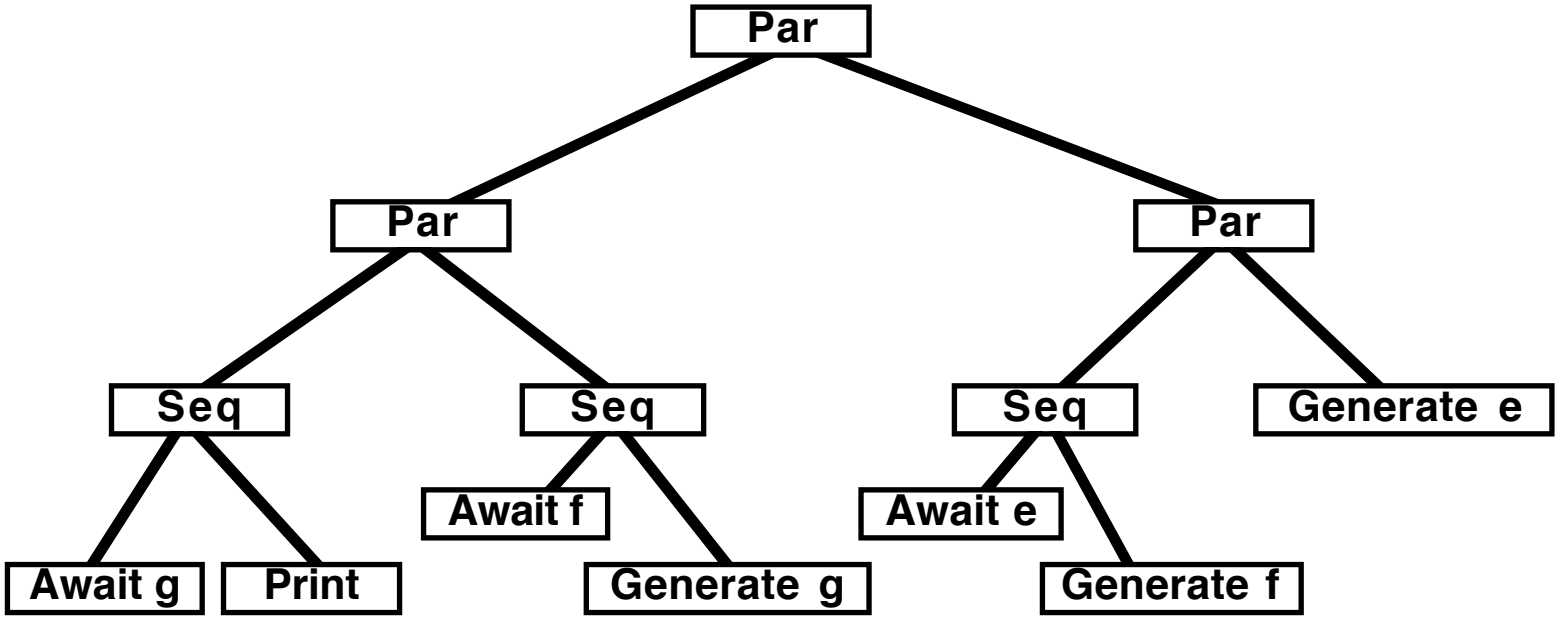
-Simple* (mécanisme de files d'attente)

Rewrite -> Replace -> Storm -> Simple

*Simple est une implémentation de FT R&D écrite par L. Hazard

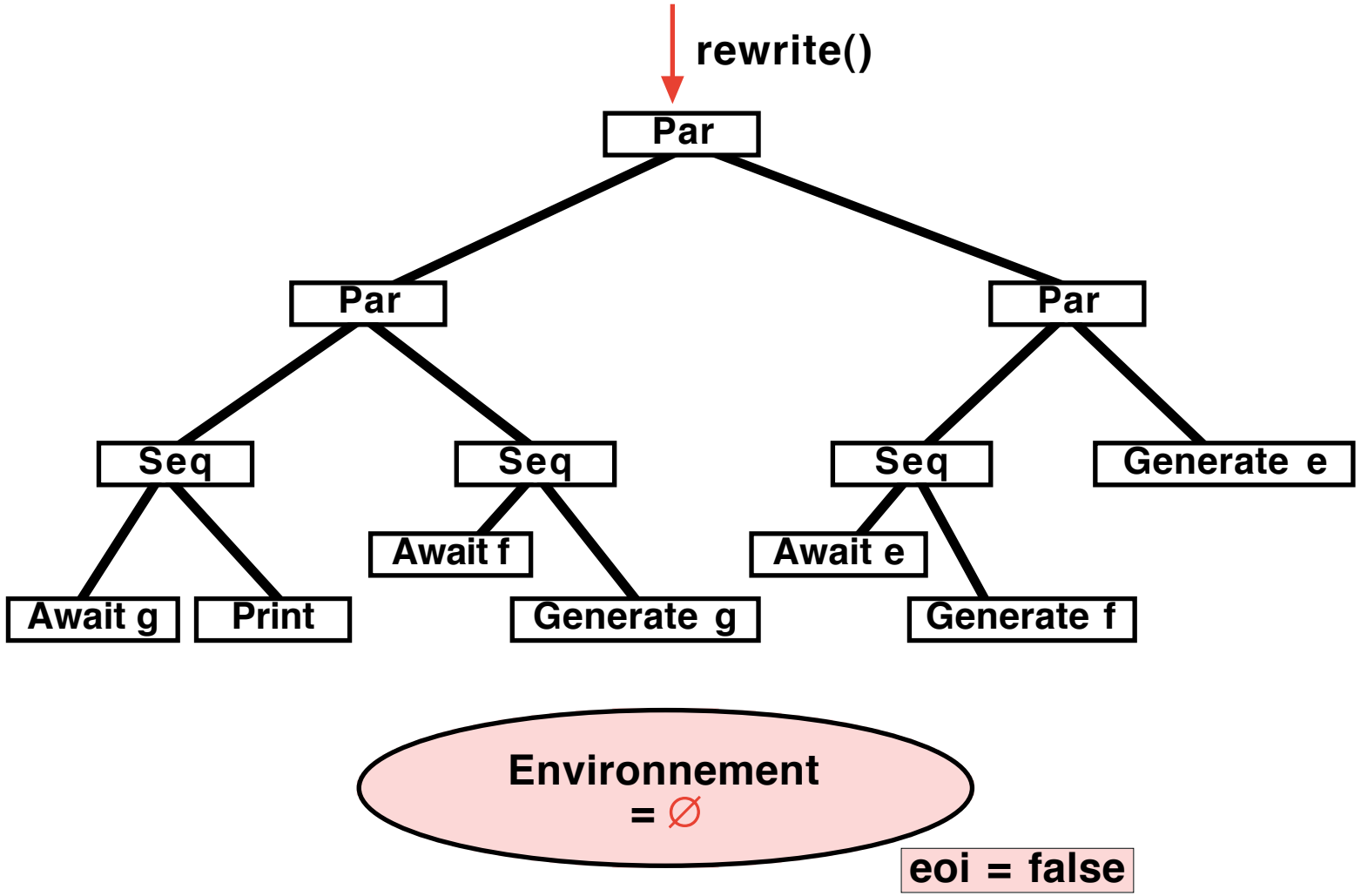
Exécution d'un programme

Programme = Arborescence d'objet-instructions

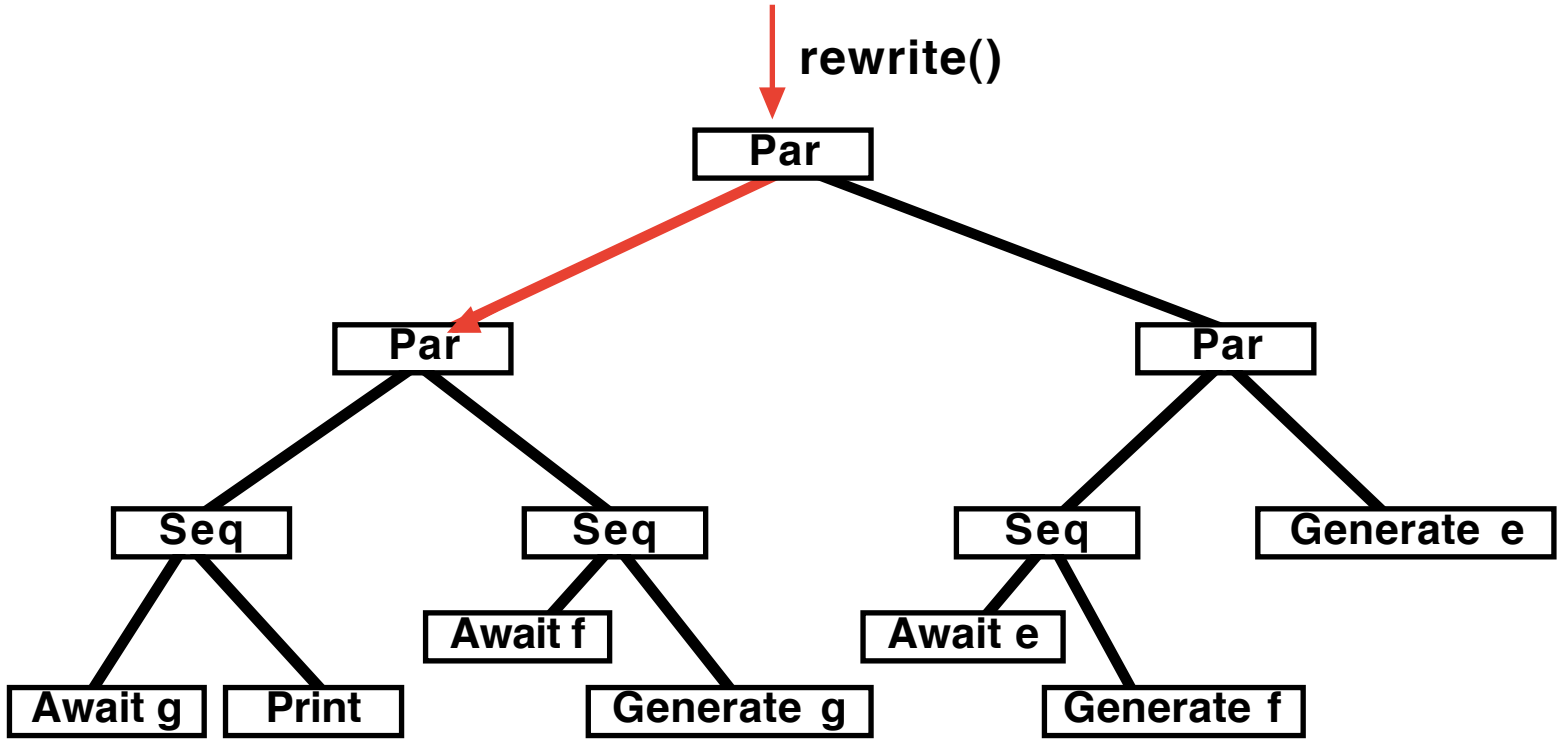


Les composants parallèles sont ordonnancés par la machine réactive selon la sémantique de l'opérateur de parallélisme

Exécution d'un programme



Exécution d'un programme

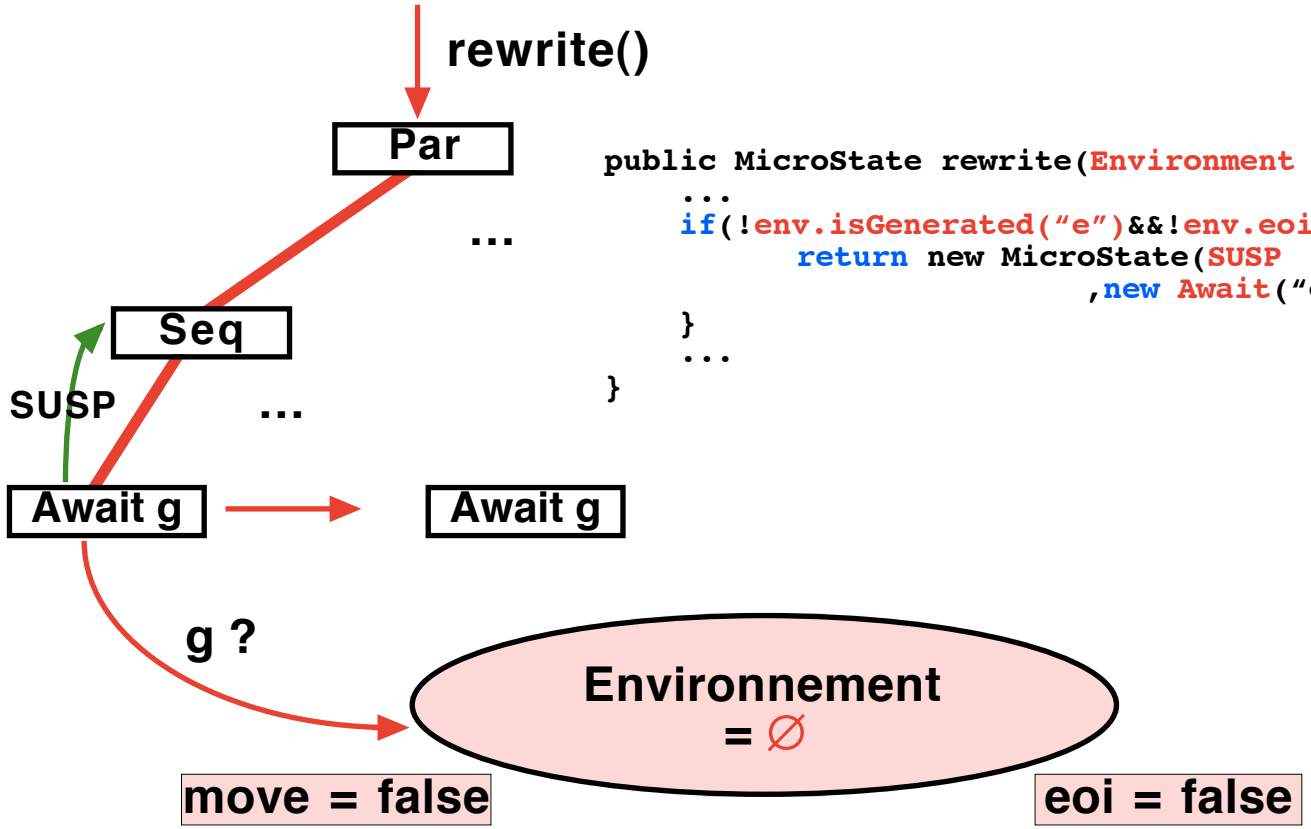


Environnement
= ∅

eoi = false

Exemple en Rewrite

$$\frac{e \in E \quad eoi = false}{Await(e), E \xrightarrow{SUSP} Await(e), E}$$

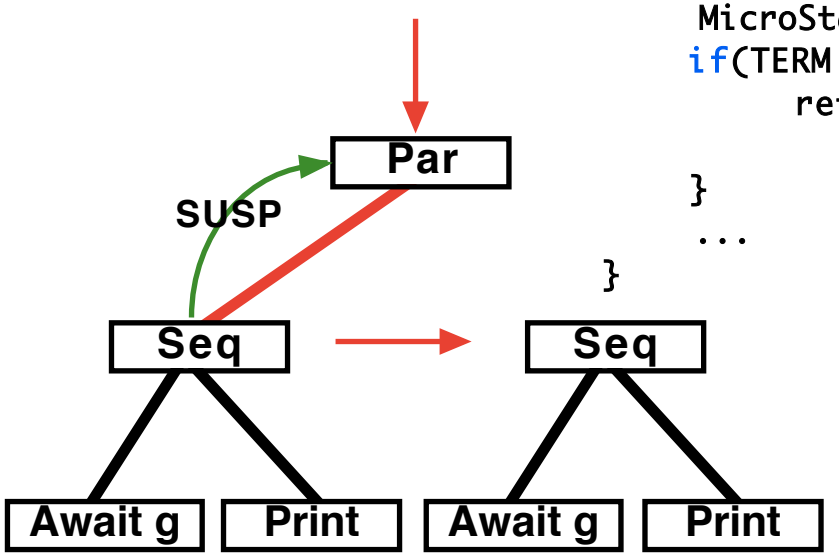



```
public MicroState rewrite(Environment env){
    ...
    if(!env.isGenerated("e")&&!env.eoi){
        return new MicroState(SUSP
                              ,new Await("e"));
    }
    ...
}
```

Exemple en Rewrite

$$\frac{t, E \xrightarrow{} t', E' \quad t' \neq TERM}{Seq(t,u), E \xrightarrow{} Seq(t',u), E'}$$

```
public MicroState rewrite(Environment env){
     MicroState s = left.rewrite(env);
     if(TERM != s.flag){
          return new MicroState(s.flag
                    ,new Seq(s.term,right));
     }
     ...
}
```



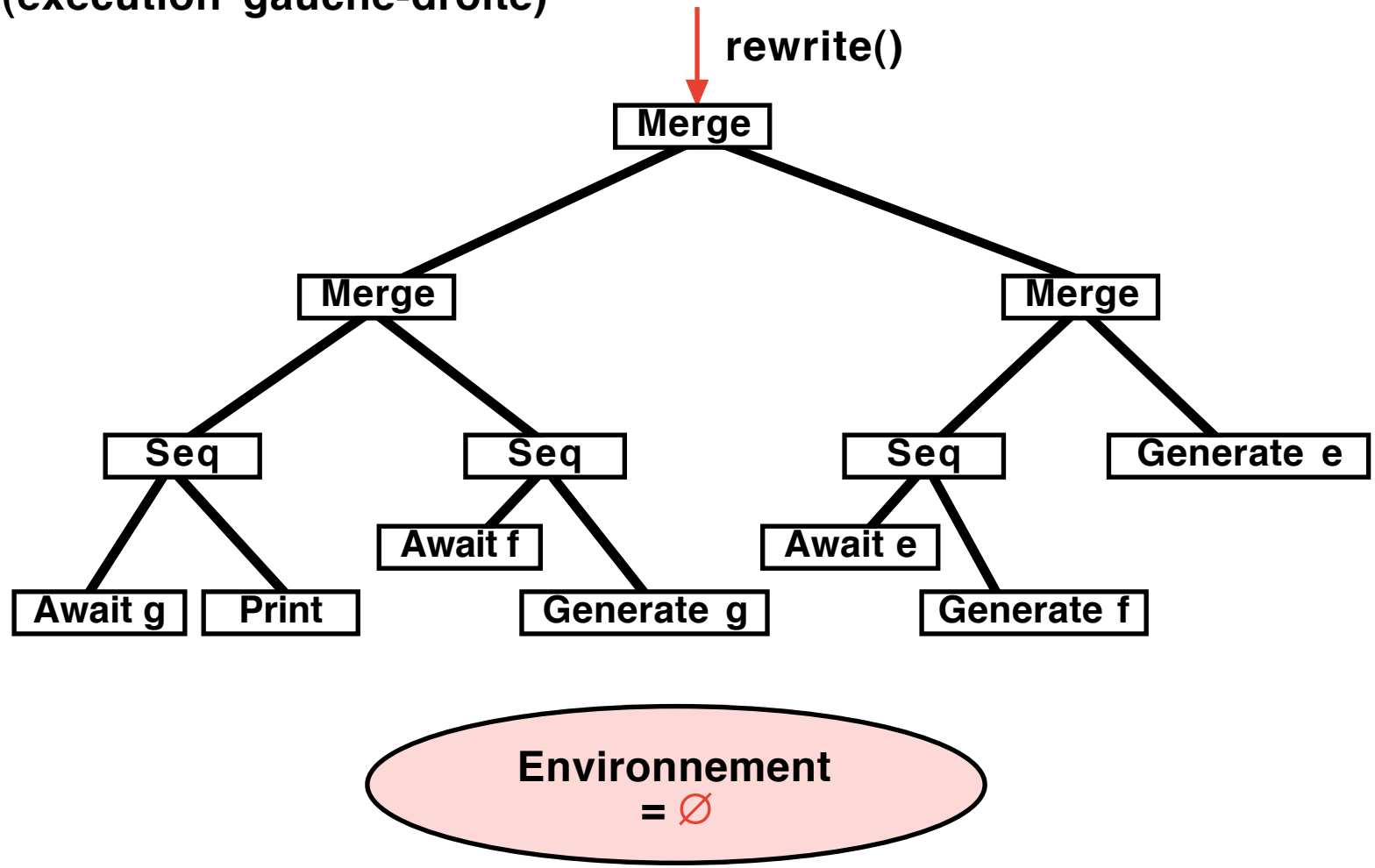
Environnement
= ∅

move = false

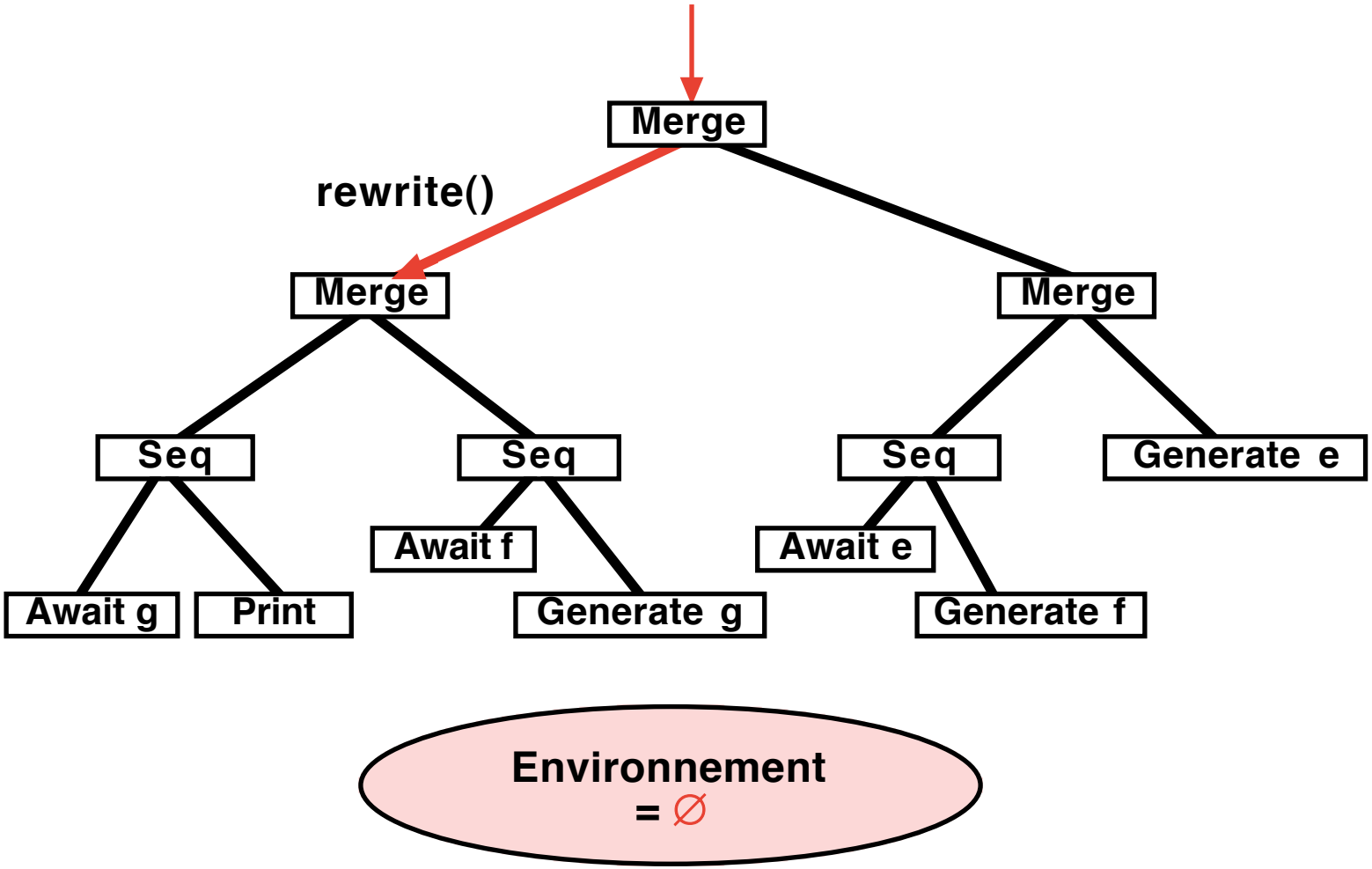
eoi = false

Rewrite-Replace

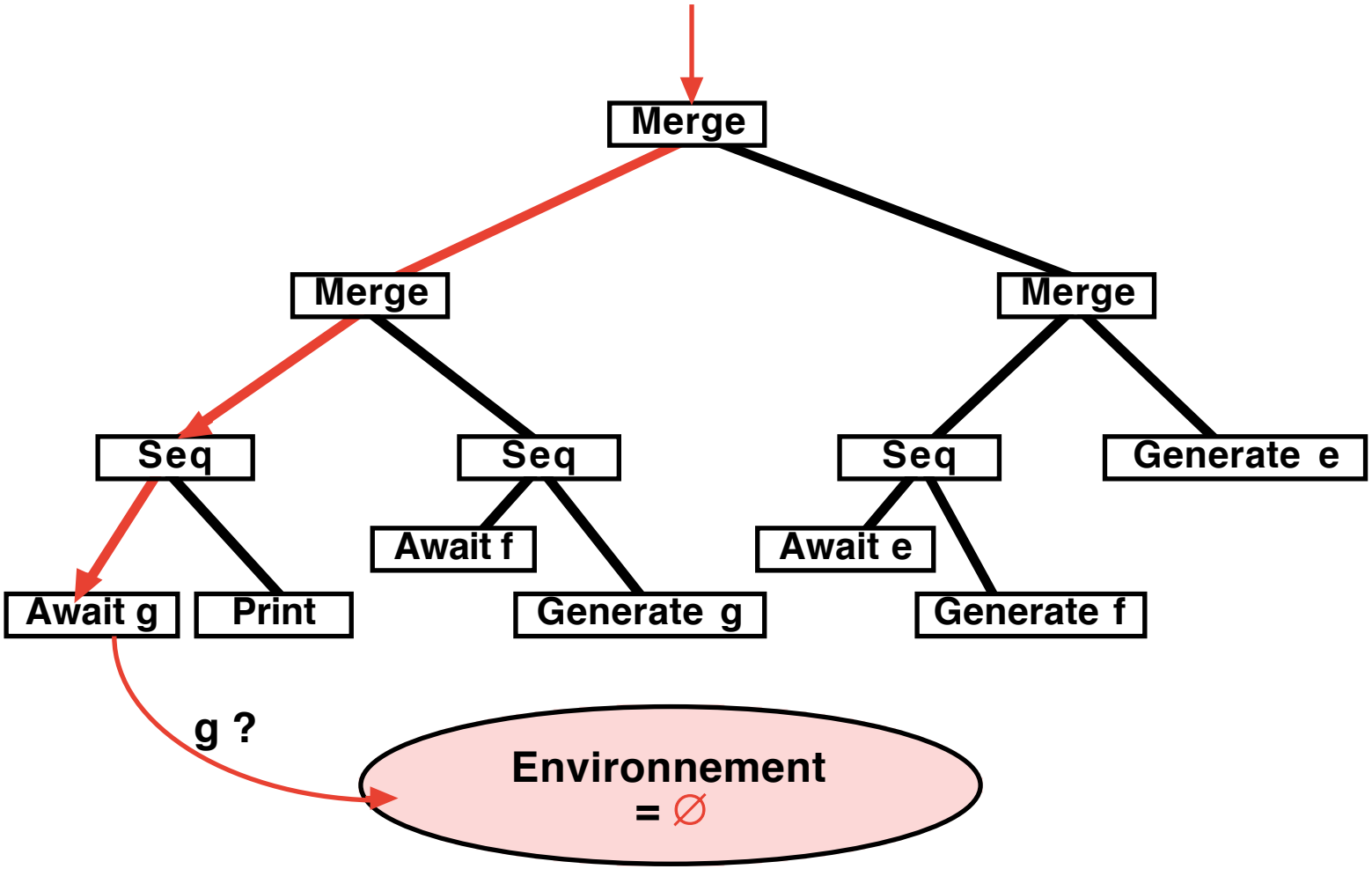
Implémentation de l'opérateur Par = opérateur Merge
(exécution gauche-droite)



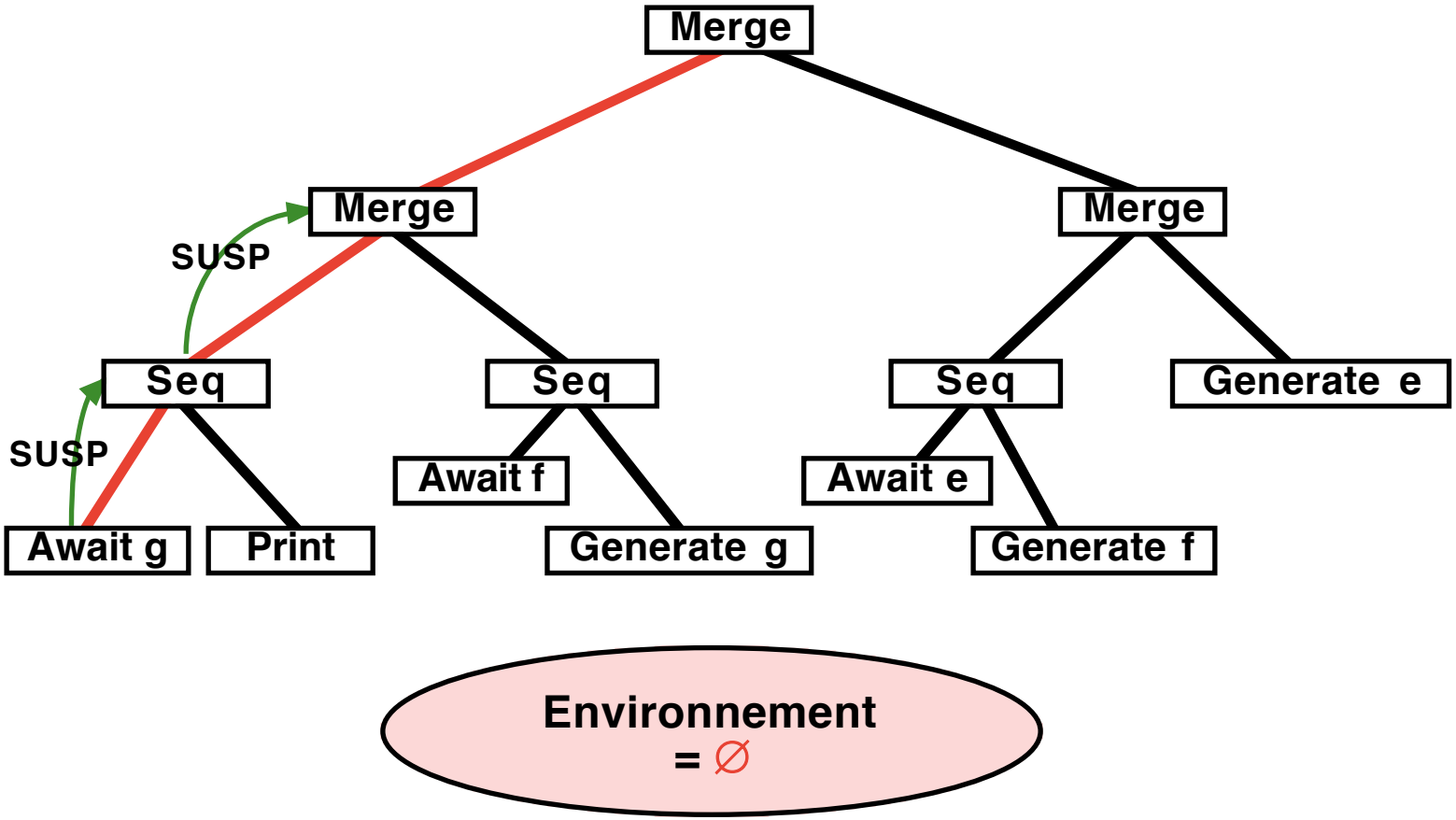
1ère micro-étape



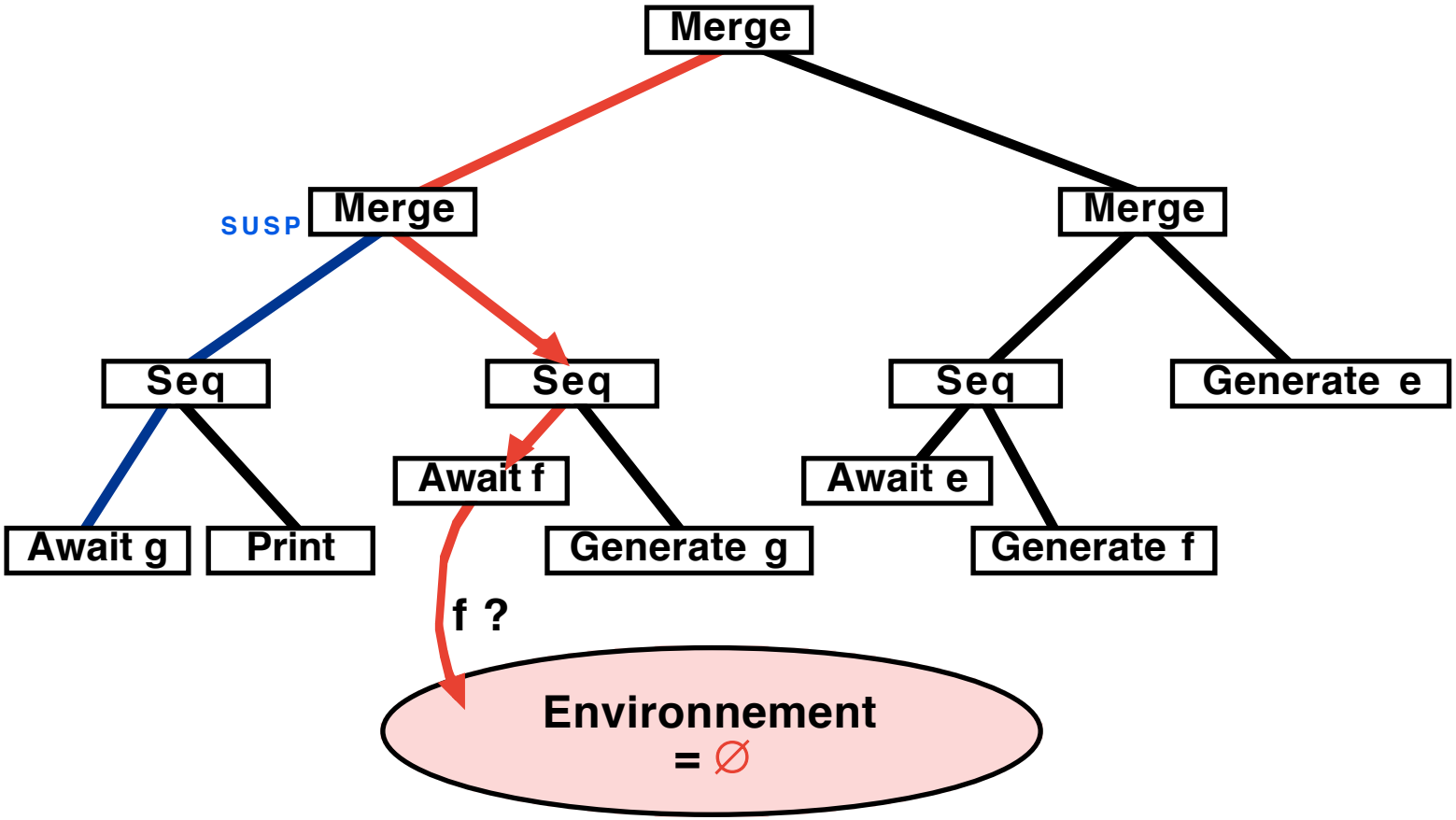
1ère micro-étape



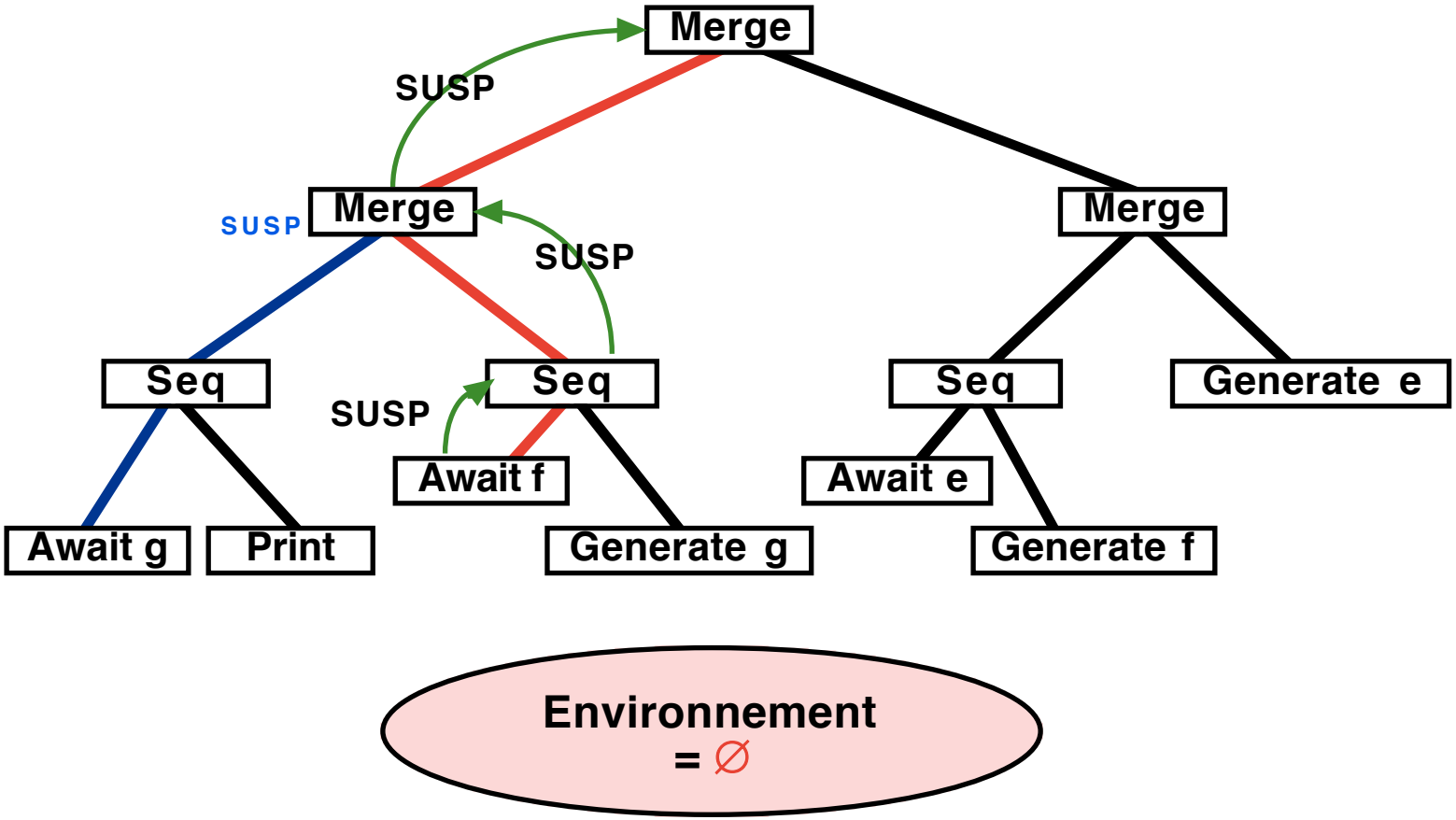
1ère micro-étape



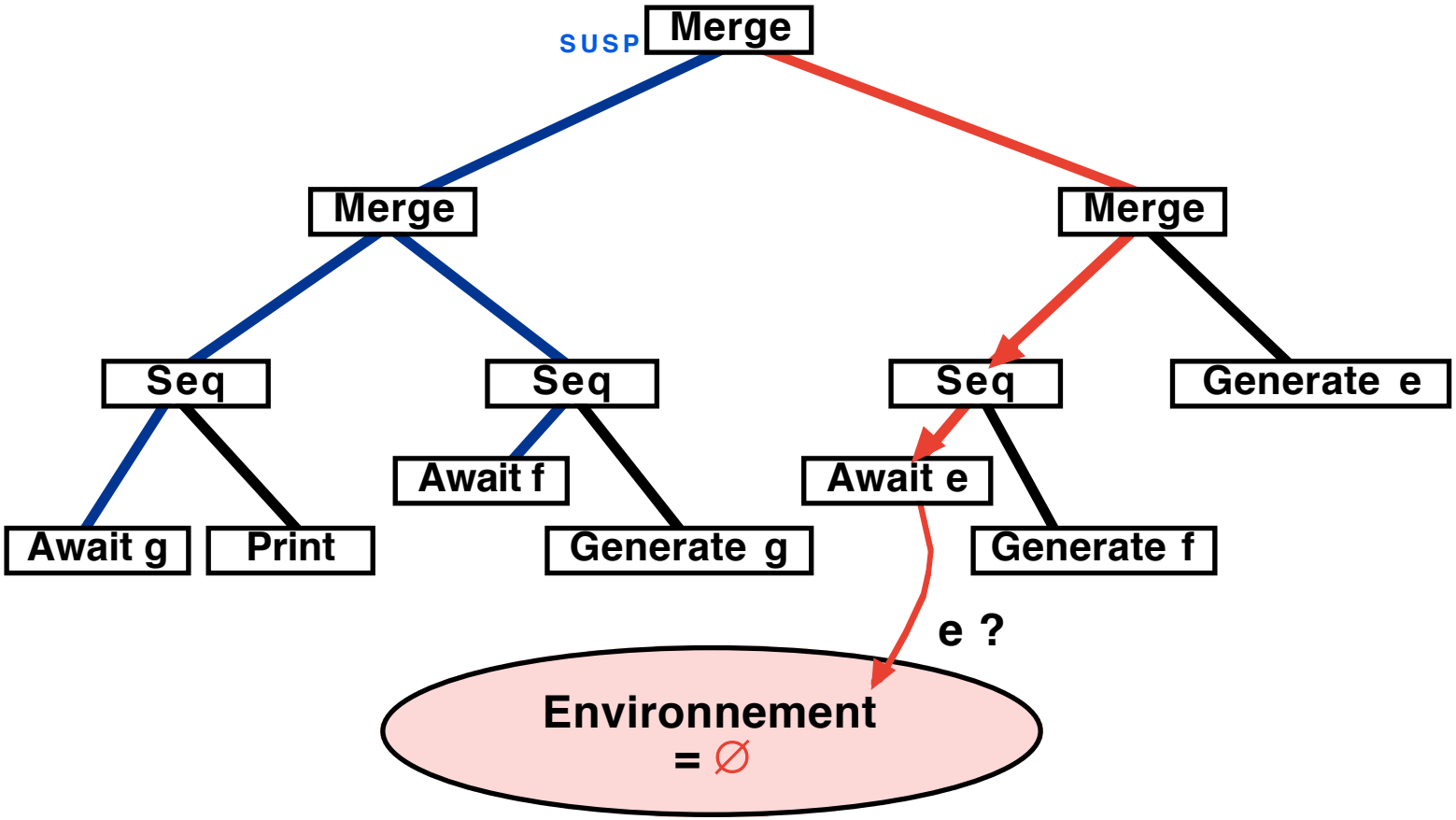
1ère micro-étape



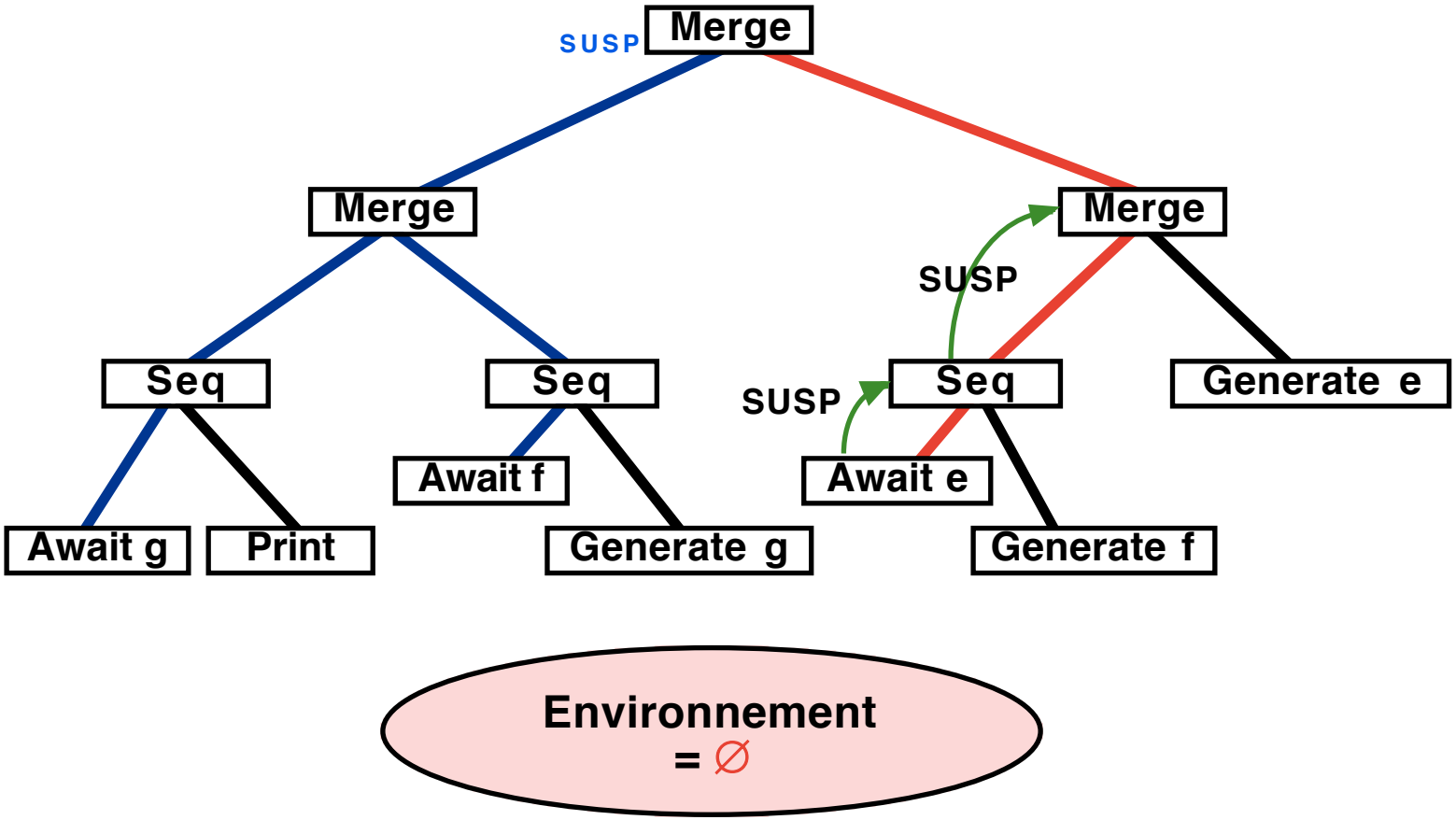
1ère micro-étape



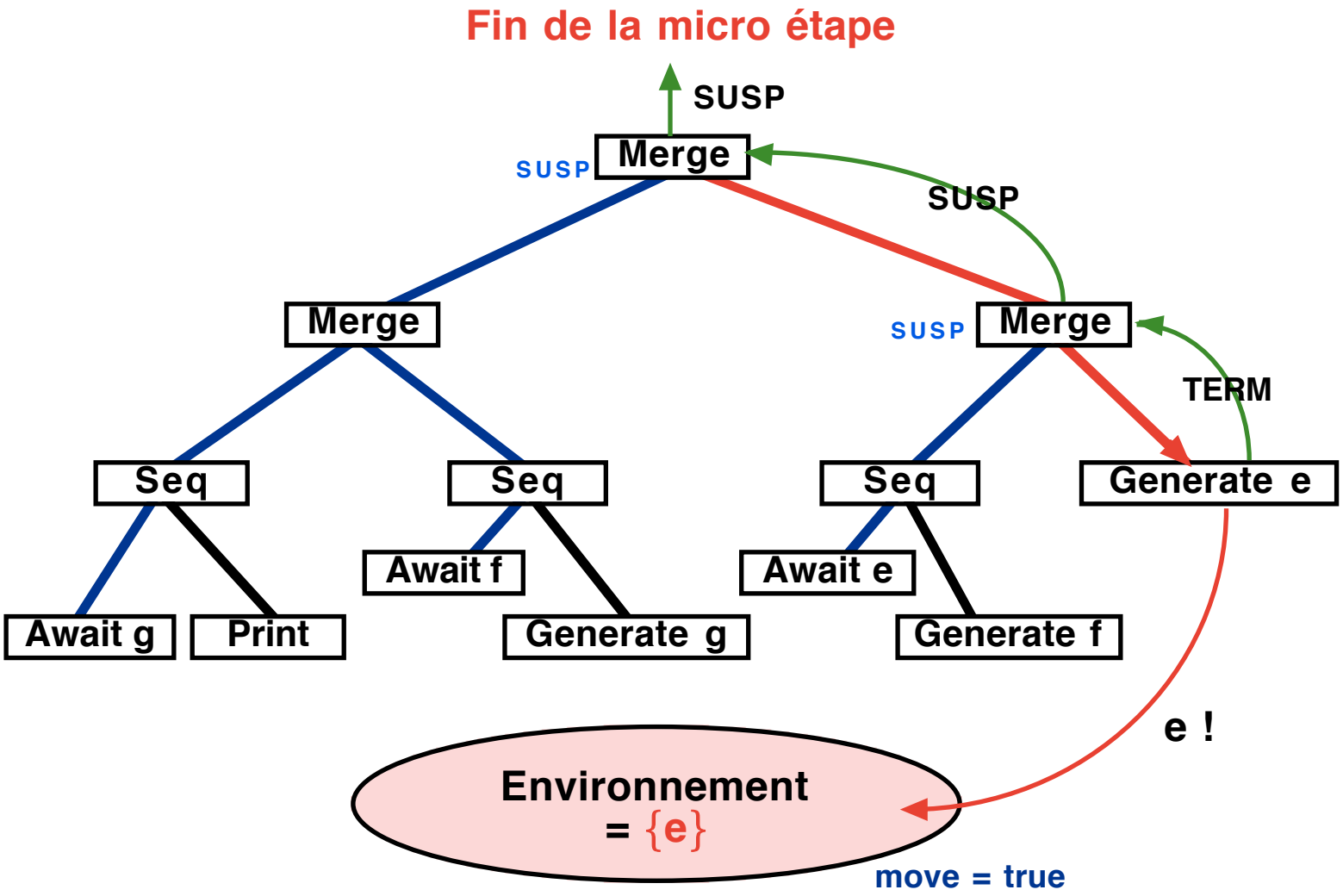
1ère micro-étape



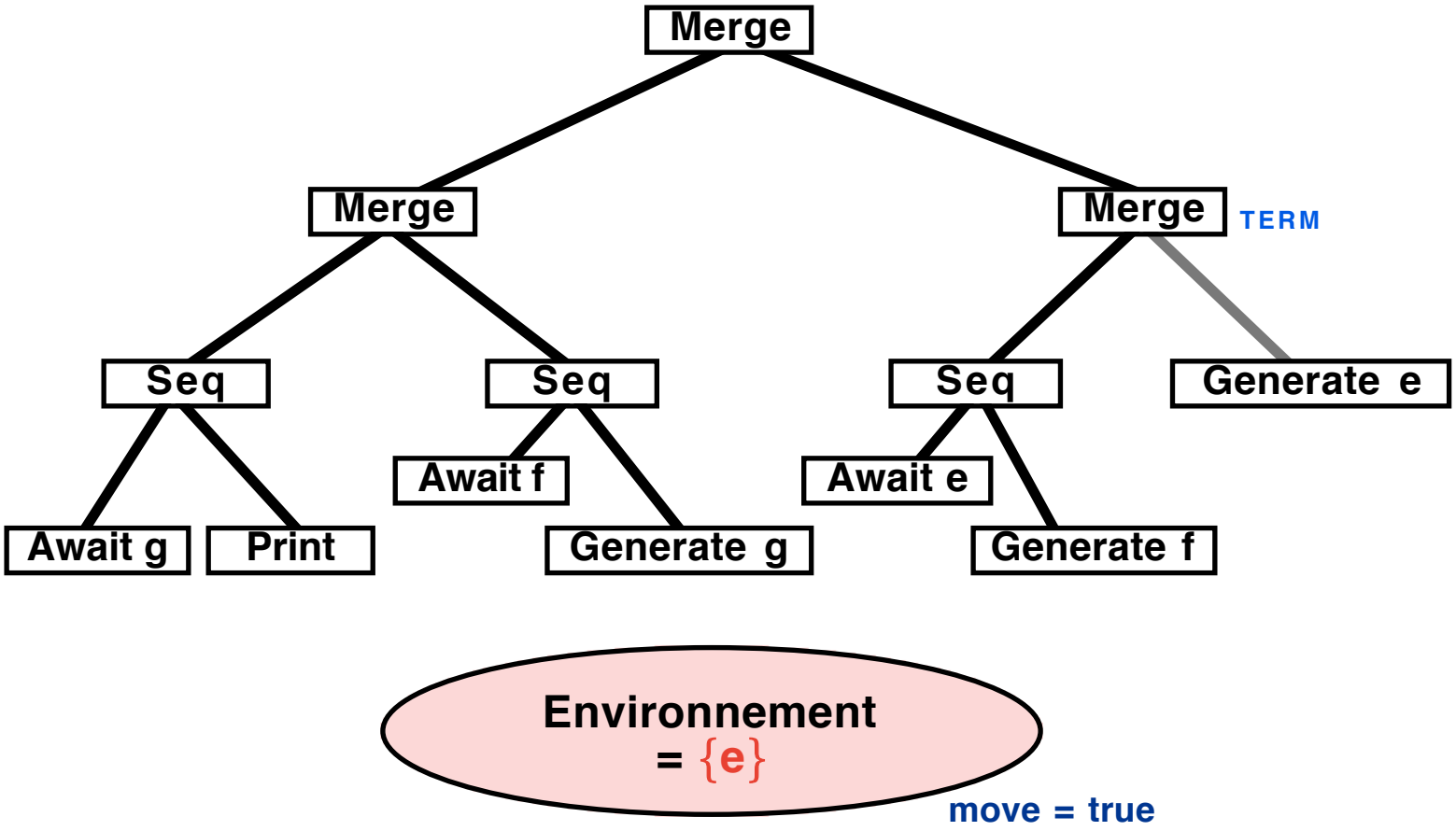
1ère micro-étape



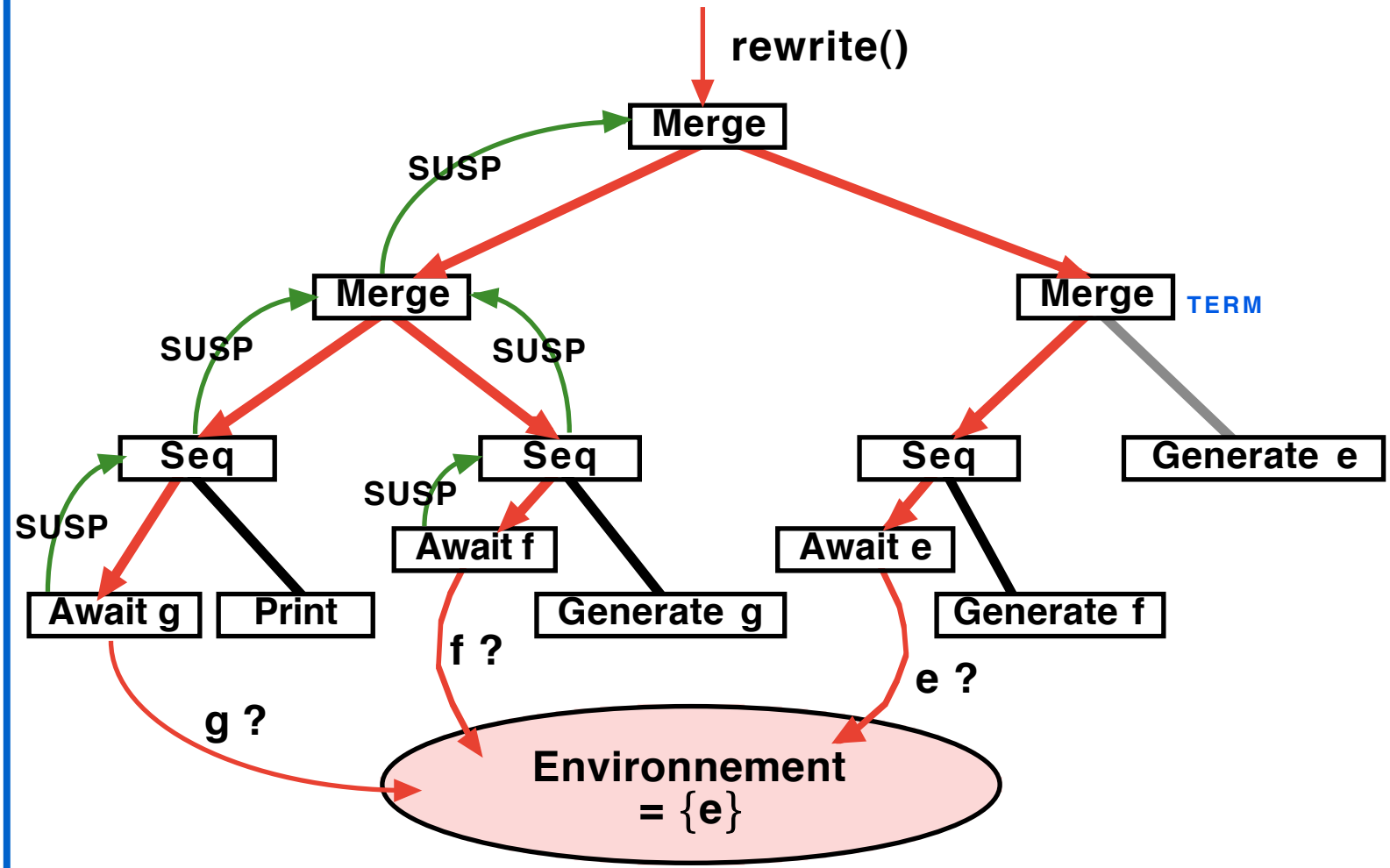
1ère micro-étape



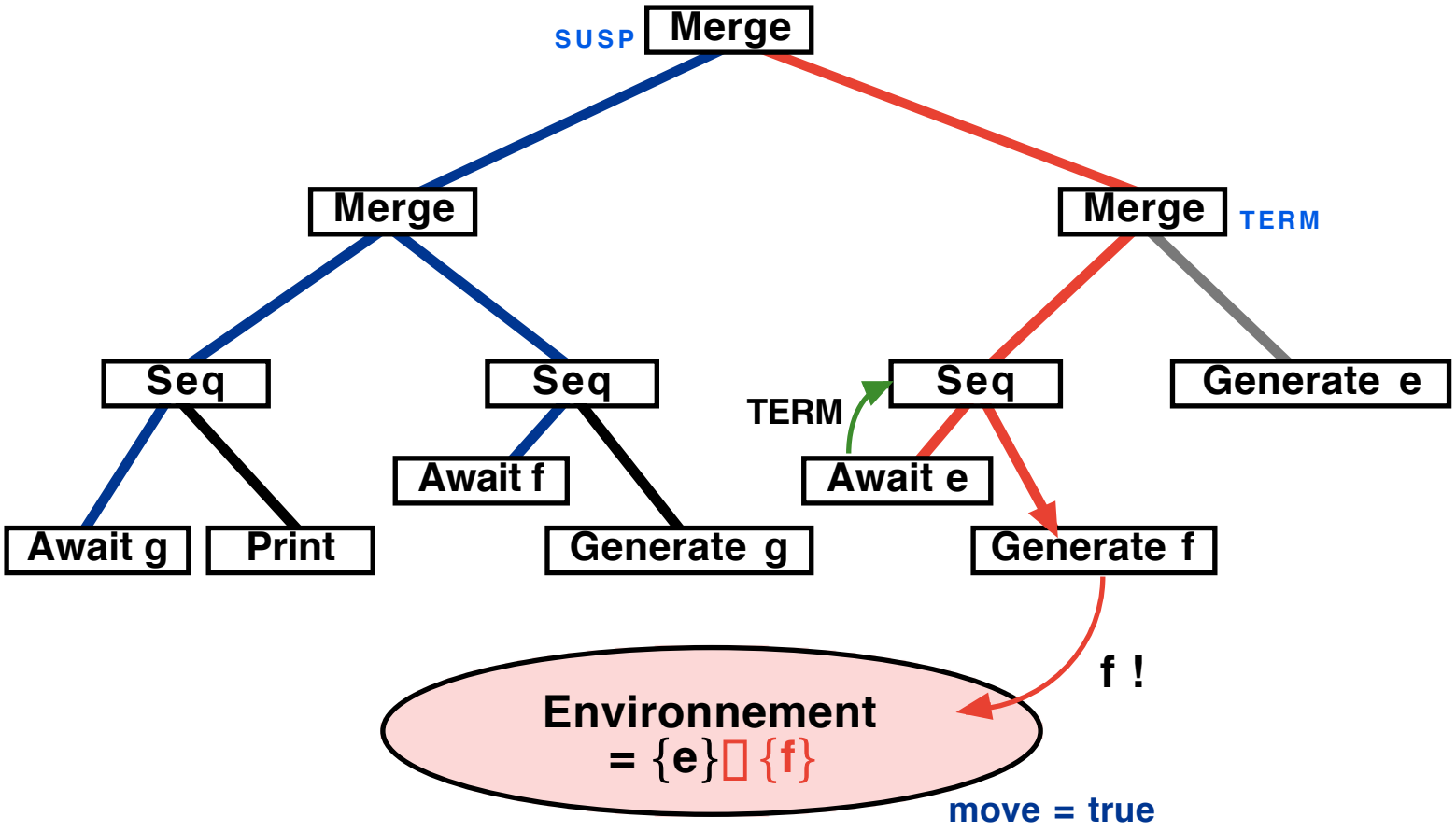
Bilan 1ère micro-étape



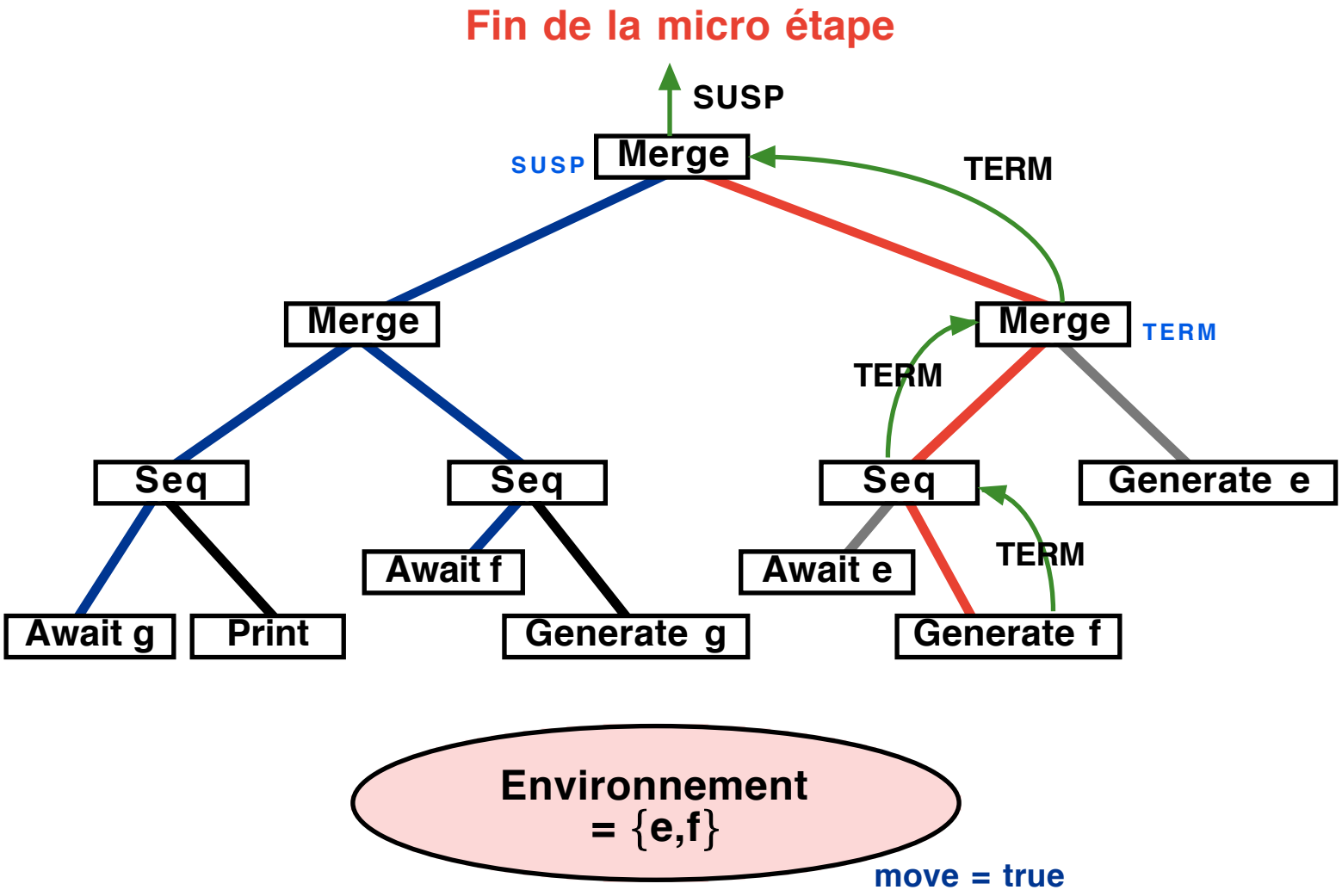
2ème micro étape



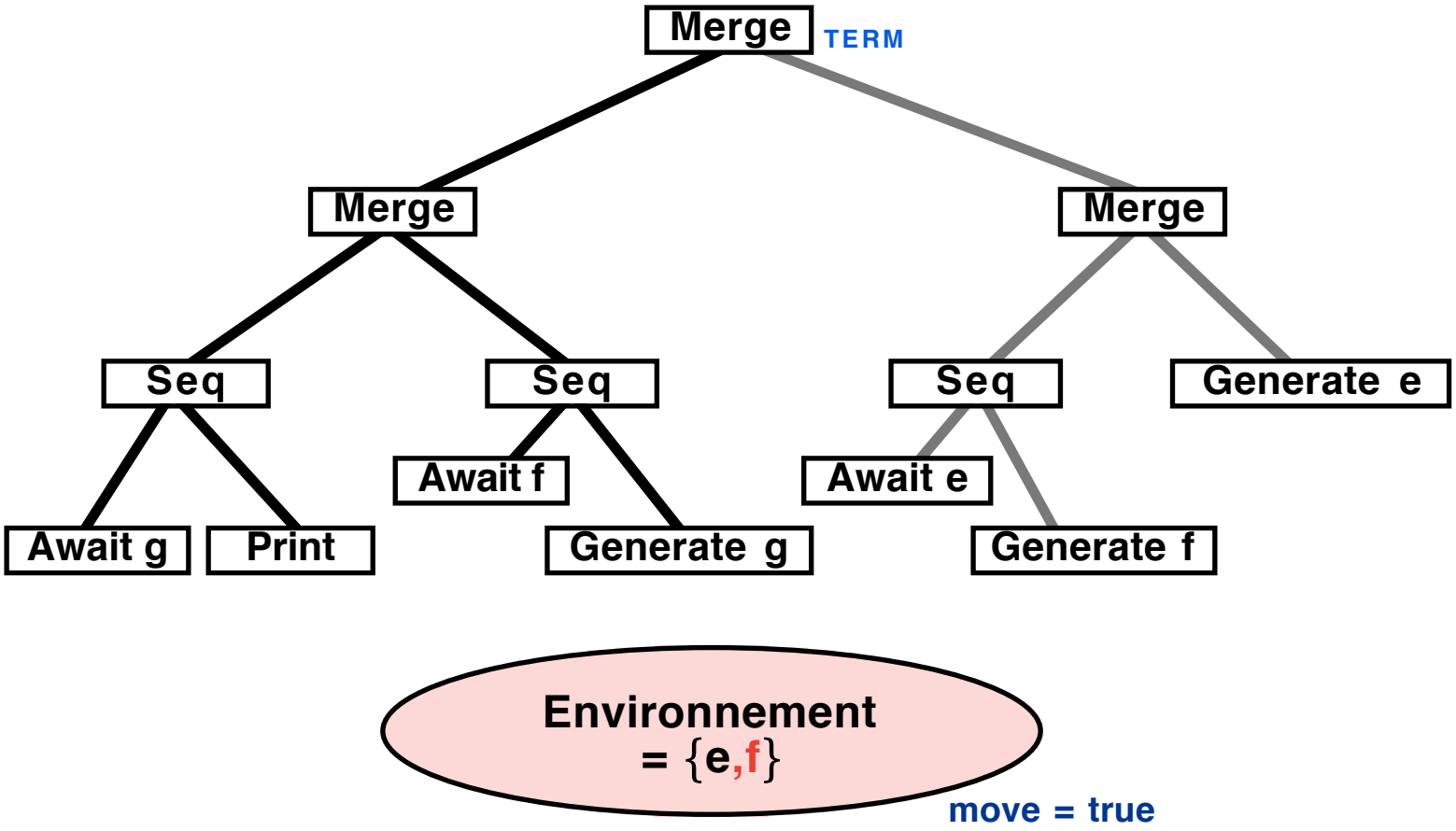
2ème micro étape



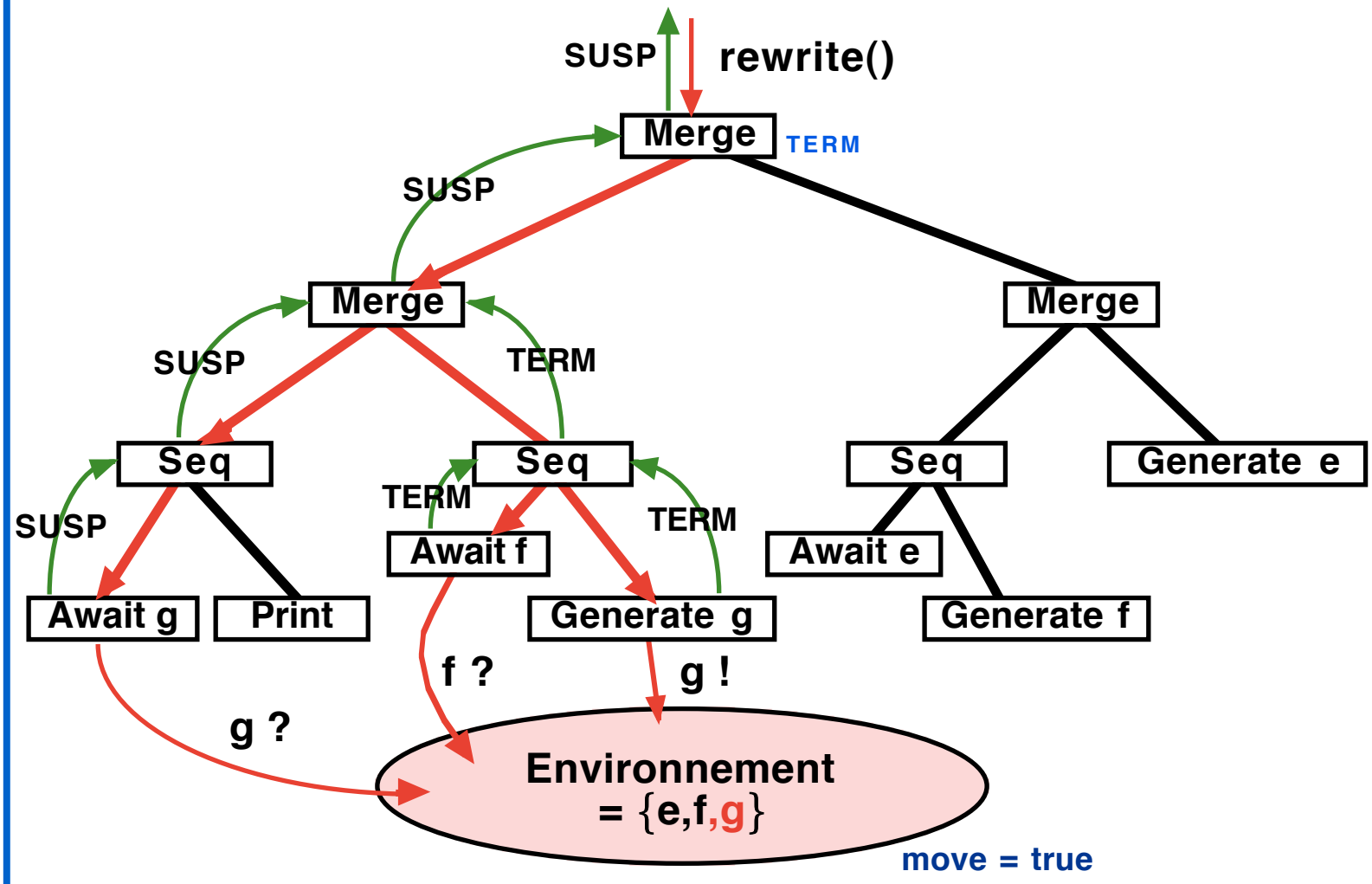
2ème micro étape



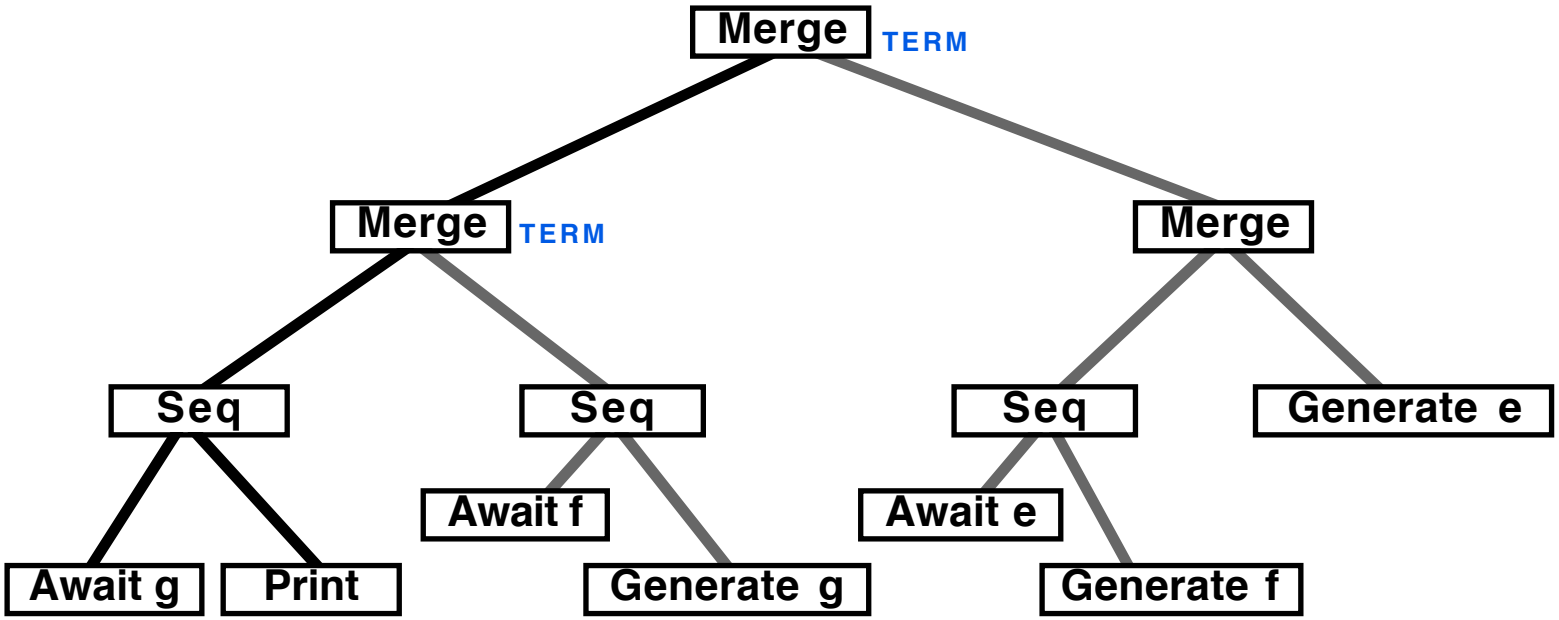
Bilan 2ème micro étape



3ème micro étape



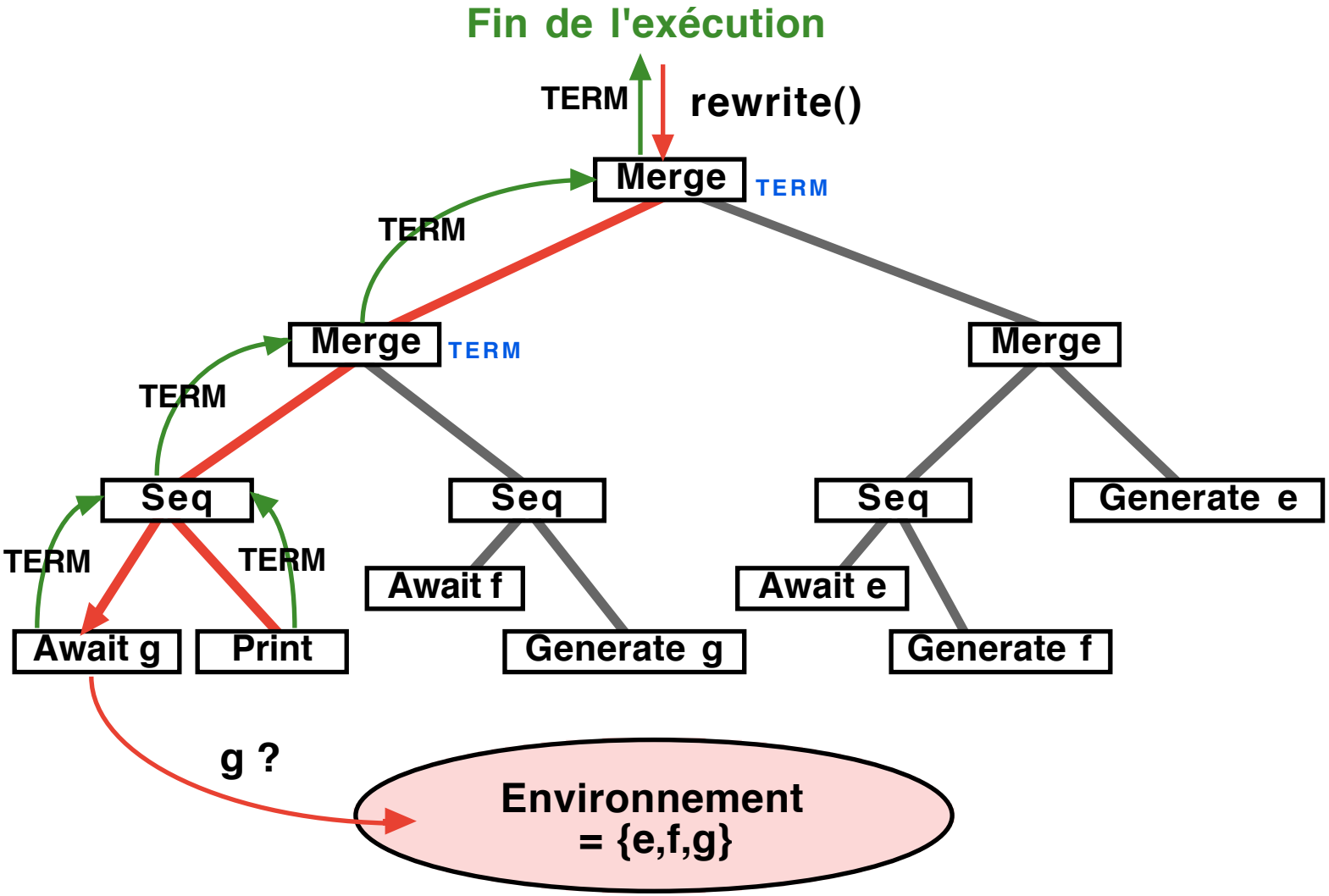
Bilan 3ème micro étape



Environnement
= {e,f,g}

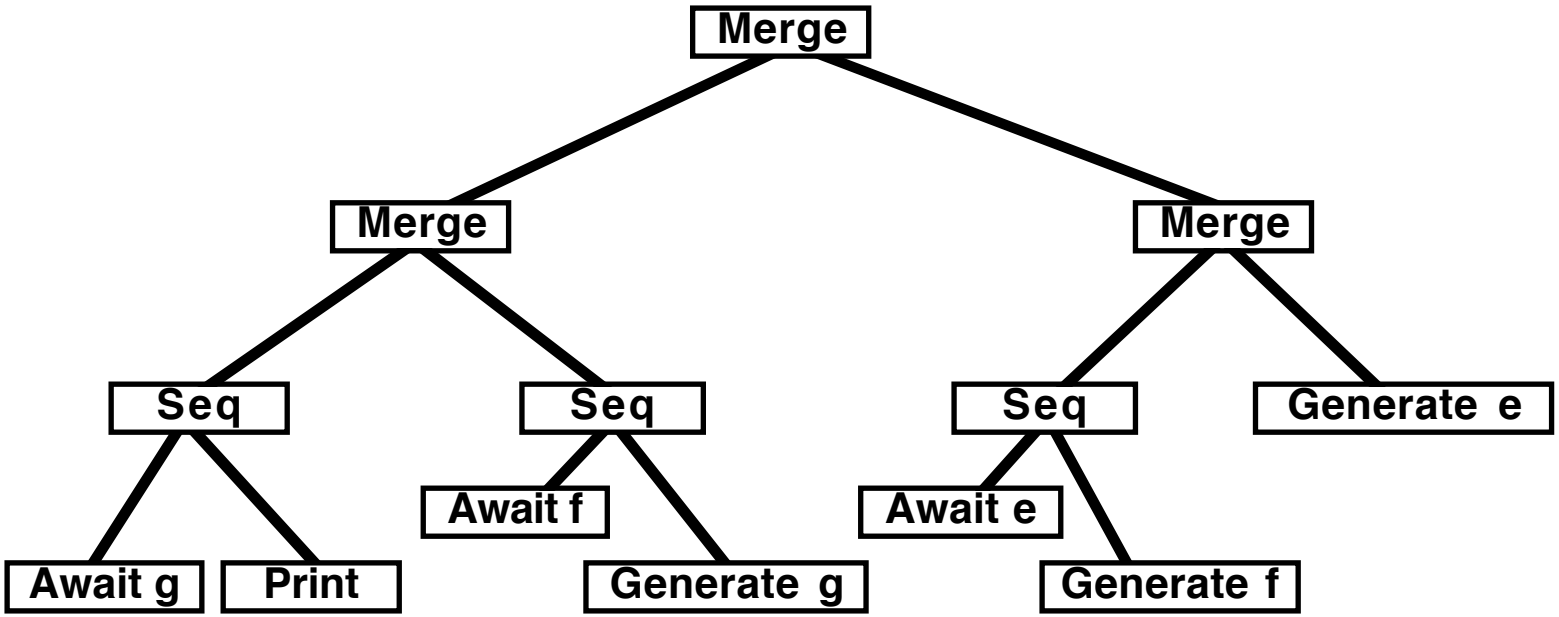
move = true

4ème micro étape



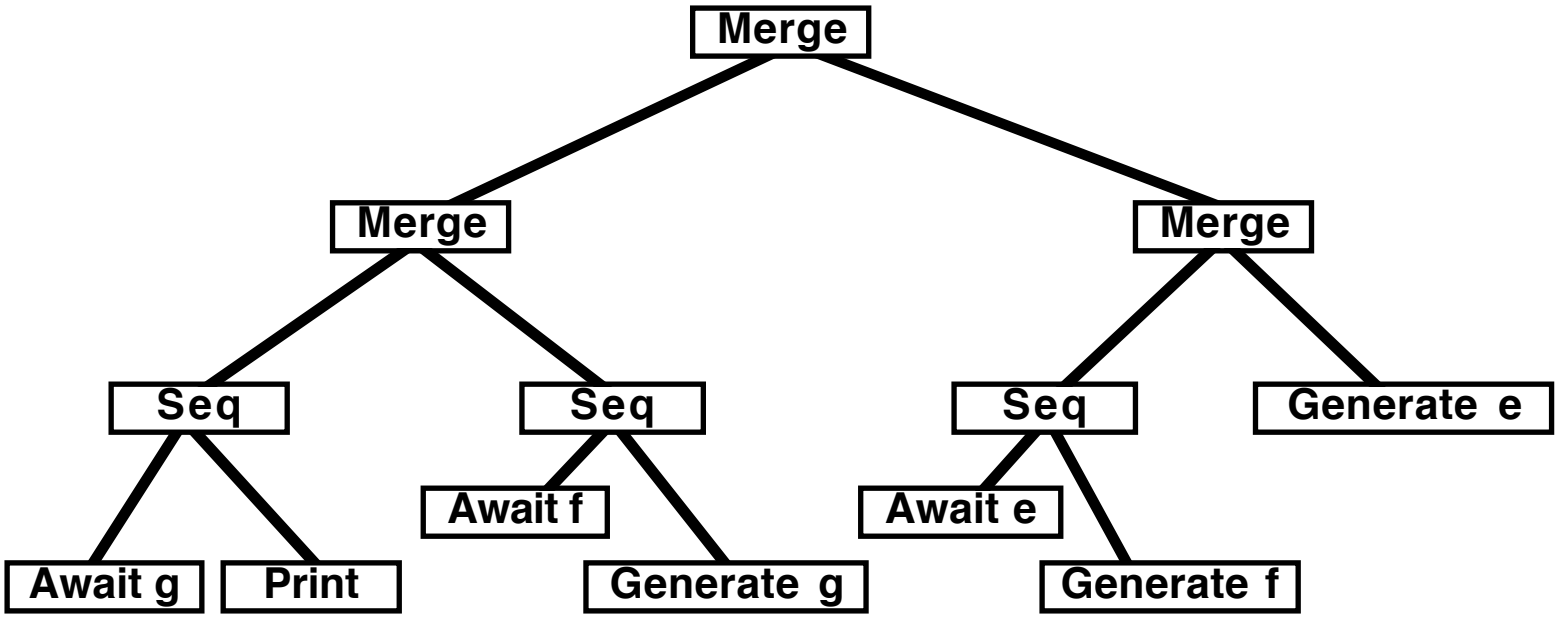
Bilan

4 micro étapes
6 activations inutiles



Complexité en : $\frac{n(n+1)}{2}$

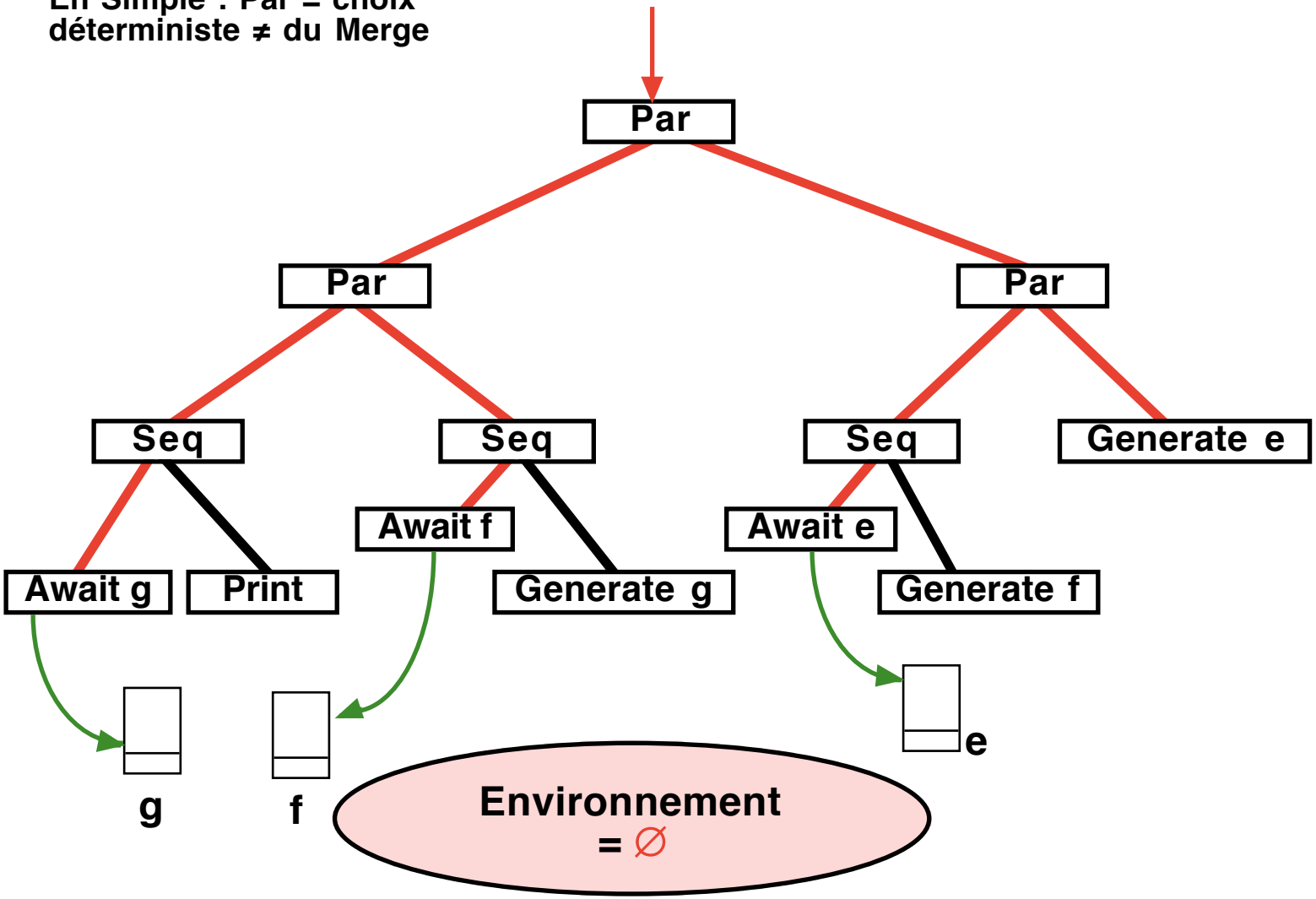
Ordonnancement contraire



Le programme s'exécute en une seule micro étape

Simple

En Simple : Par = choix déterministe ≠ du Merge

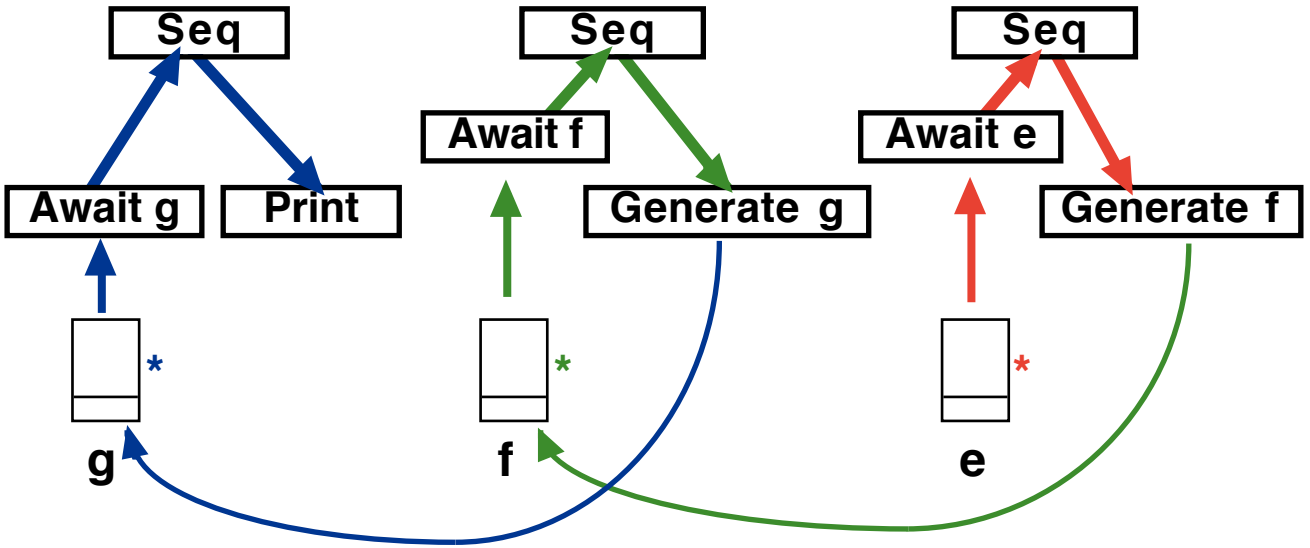


Simple

La structure du programme est éclatée dans les files d'attente.

Les opérateurs Par de plus haut niveau disparaissent

Croissance linéaire avec le nombre de composants et d'événements à traiter effectivement



Simple

Simple réduit la complexité inter-instant : évite la diffusion de fin d'instant inutile

Ex : Await("e") si e n'est pas là au cours de l'instant

Simple introduit une file d'attente des composants stoppés.

Problèmes :

algorithme complexe (abonnement désabonnement dans des files d'attente)

2 sens d'exécution :

- descendante (ex : pour la toute première activation)
- remontante (activation par les fils d'attentes)

Exécution non structurelle => pas de règles SOS

Mise au point et analyse délicate.

Storm

Optimisation fondée sur une variation des règles de réécriture

- **Storm étend directement Replace**
- **Différencie attente sur événement et suspension**

$$t, E \xrightarrow{} t, E' \quad \text{si } E \not\models t \rightsquigarrow \{SUSP, WAIT, STOP, TERM\}$$

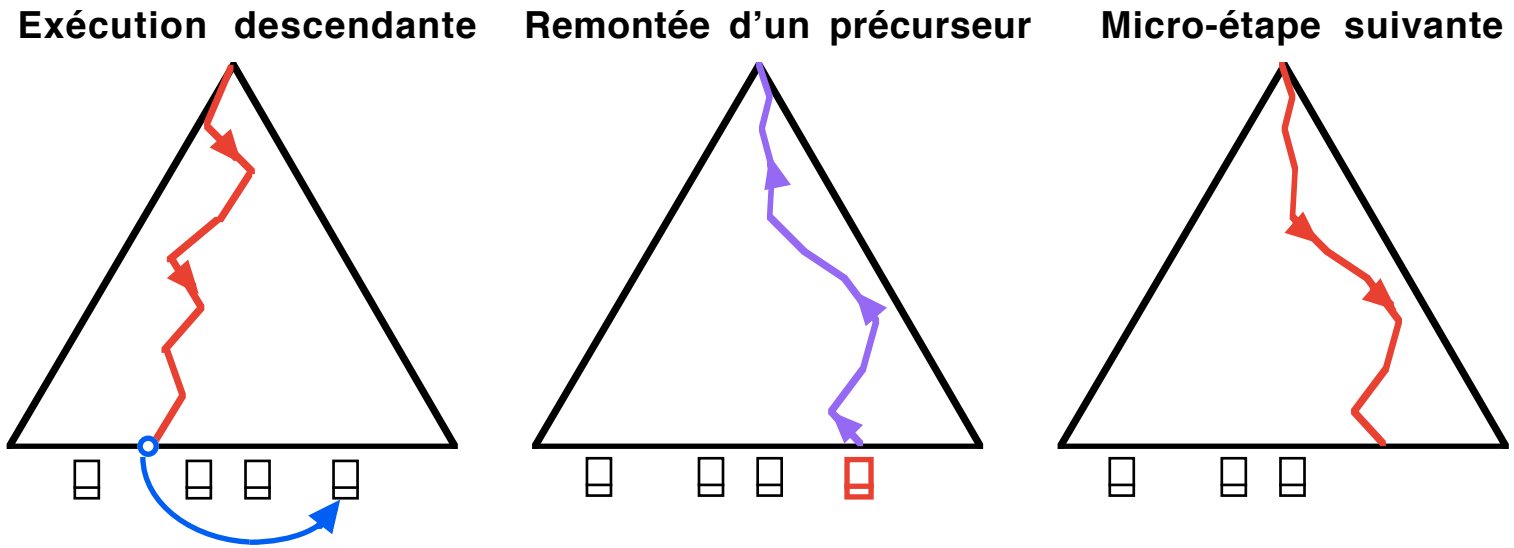
- **Exécution uniquement structurelle**
- **Implémentation efficace à base de files d'attente**
- **Mécanisme de précurseurs**

$$E \models t \overset{\Delta}{\rightsquigarrow} t' \quad \text{si } \Delta \neq \{SUSP, WAIT\}$$

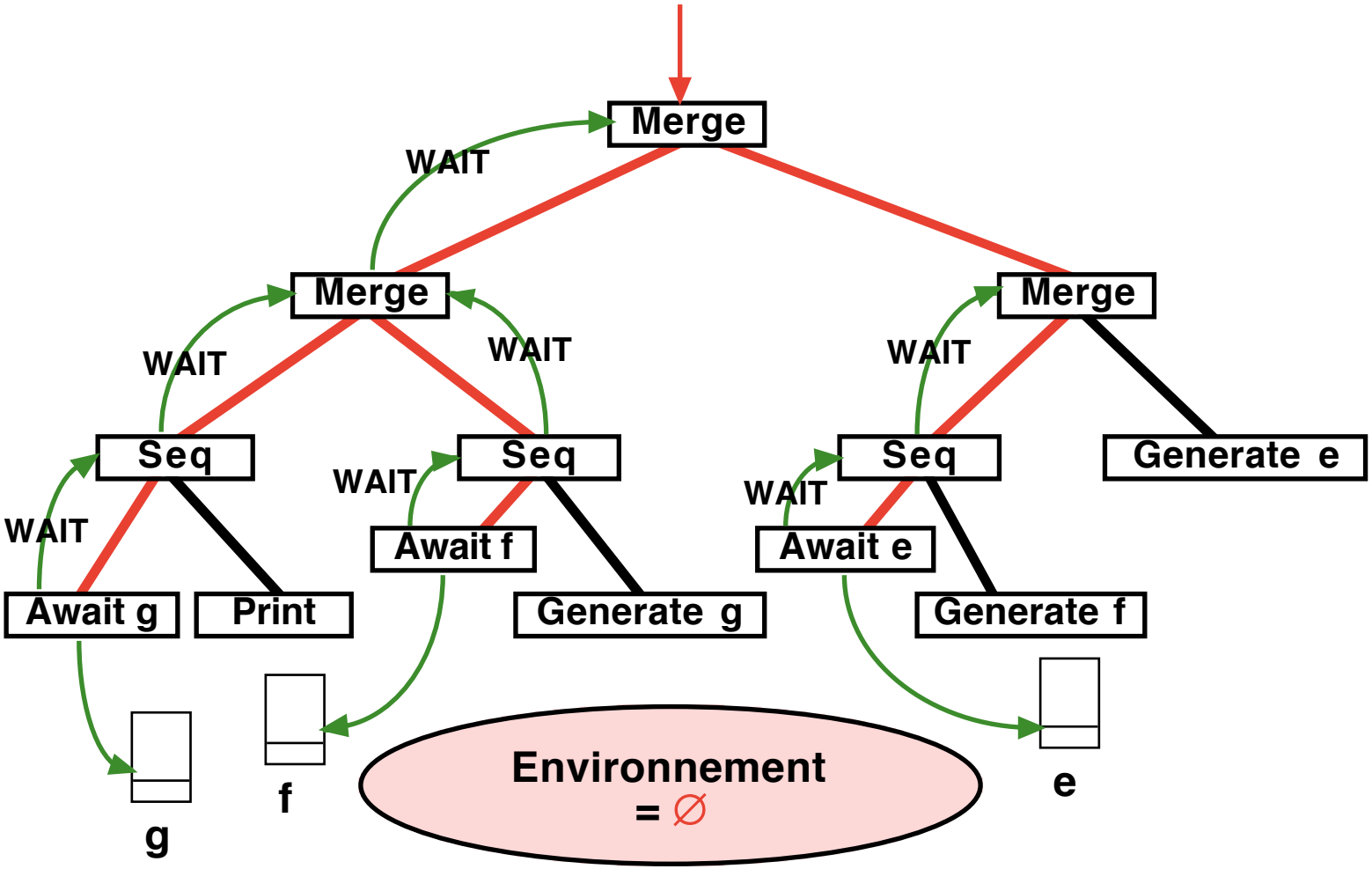
Précurseur +exécution = éclair

Exécution descendante
Par n'exécute que des branches pouvant progresser (*SUSP*)

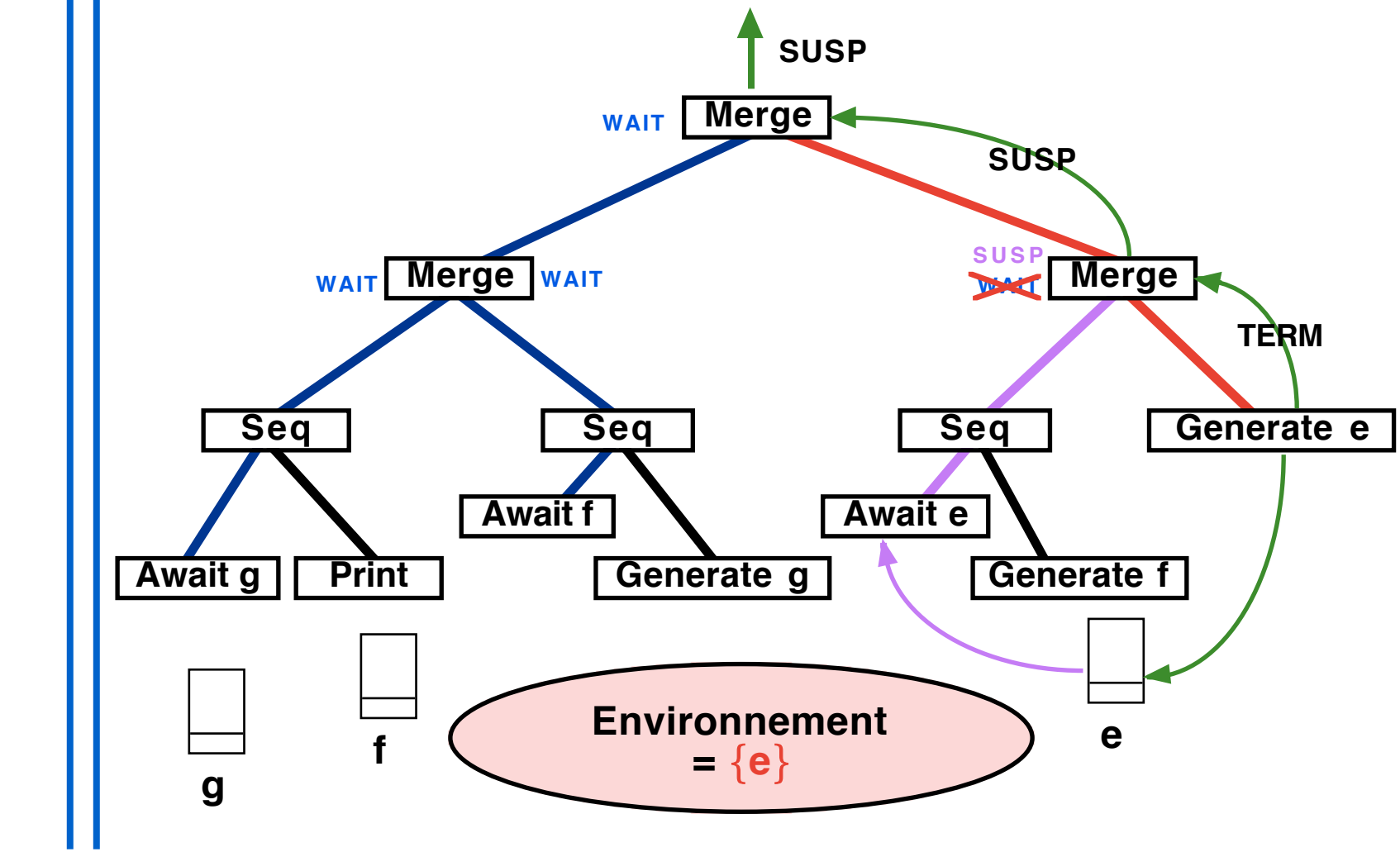
Précurseur
libère un chemin d'exécution par modification des status WAIT en SUSP



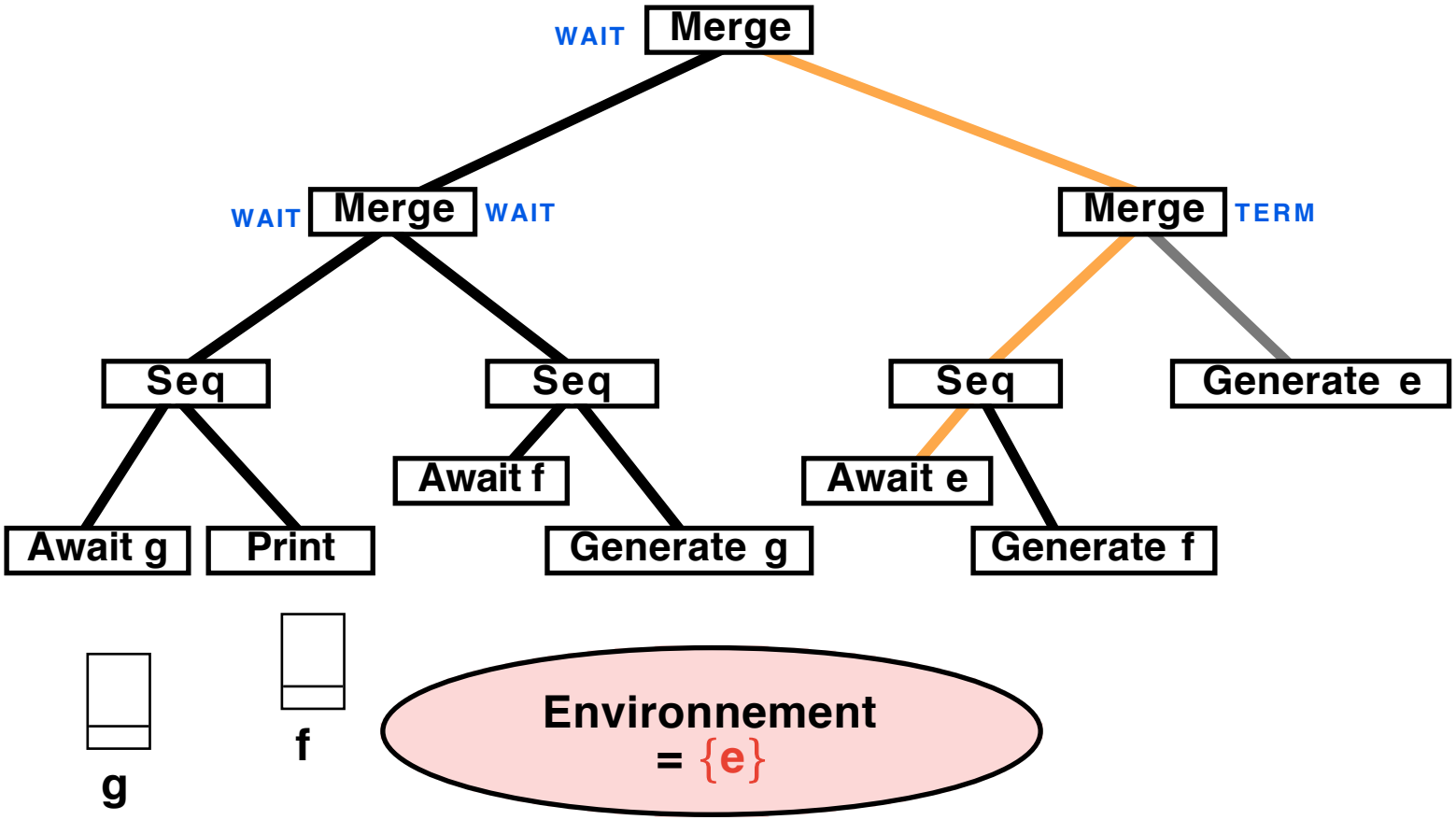
Storm


Merge
WAIT
Merge
Merge
WAIT
WAIT
WAIT
Seq
Seq
Seq
Generate e
WAIT
WAIT
WAIT
Await f
Await e
Await g
Print
Generate g
Generate f
g
f
e
Environnement
= ∅

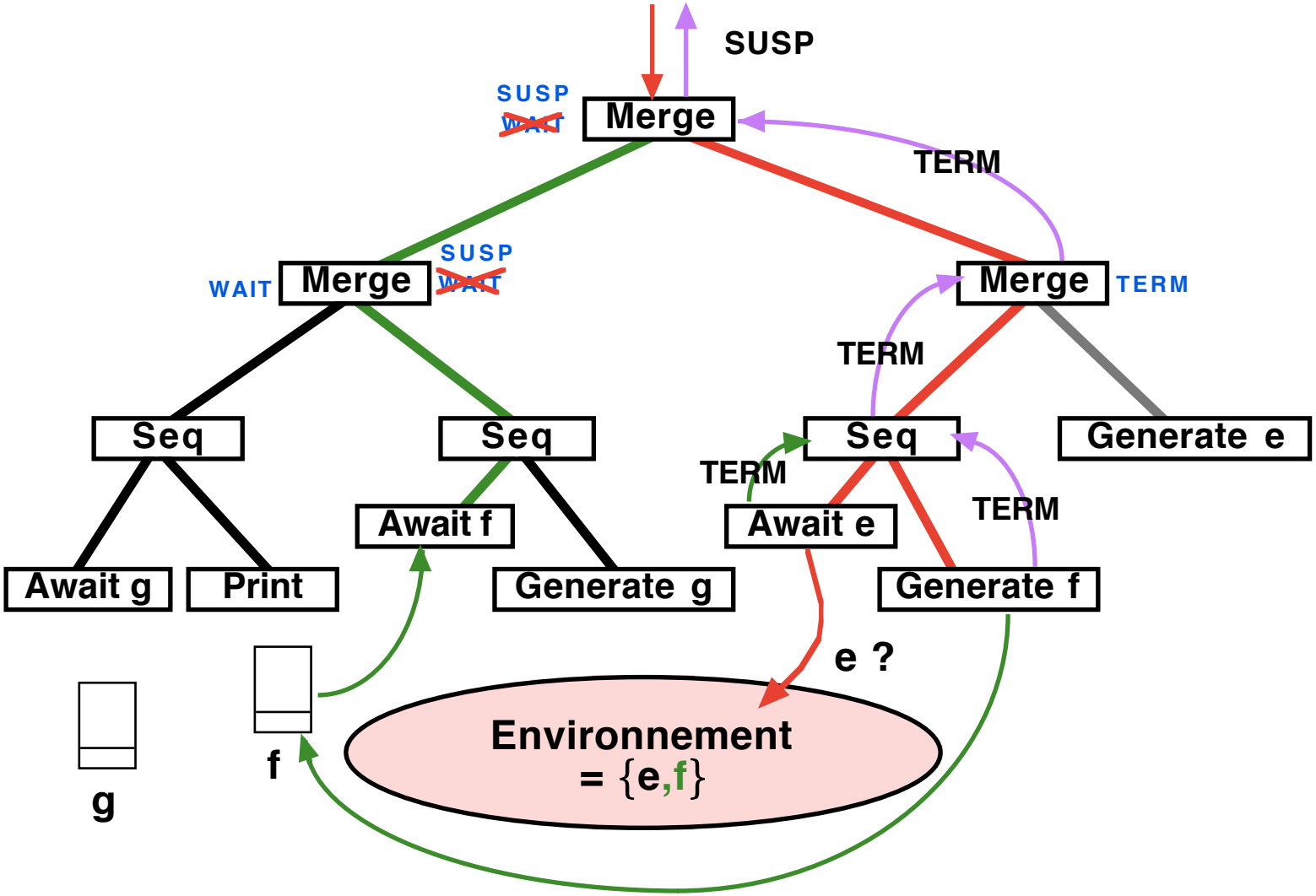
Storm


SUSP
WAIT
Merge
SUSP
WAIT
Merge
WAIT
SUSP
Merge
TERM
Seq
Seq
Seq
Generate e
Await f
Await e
Await g
Print
Generate g
Generate f
g
f
e
Environnement
= {e}

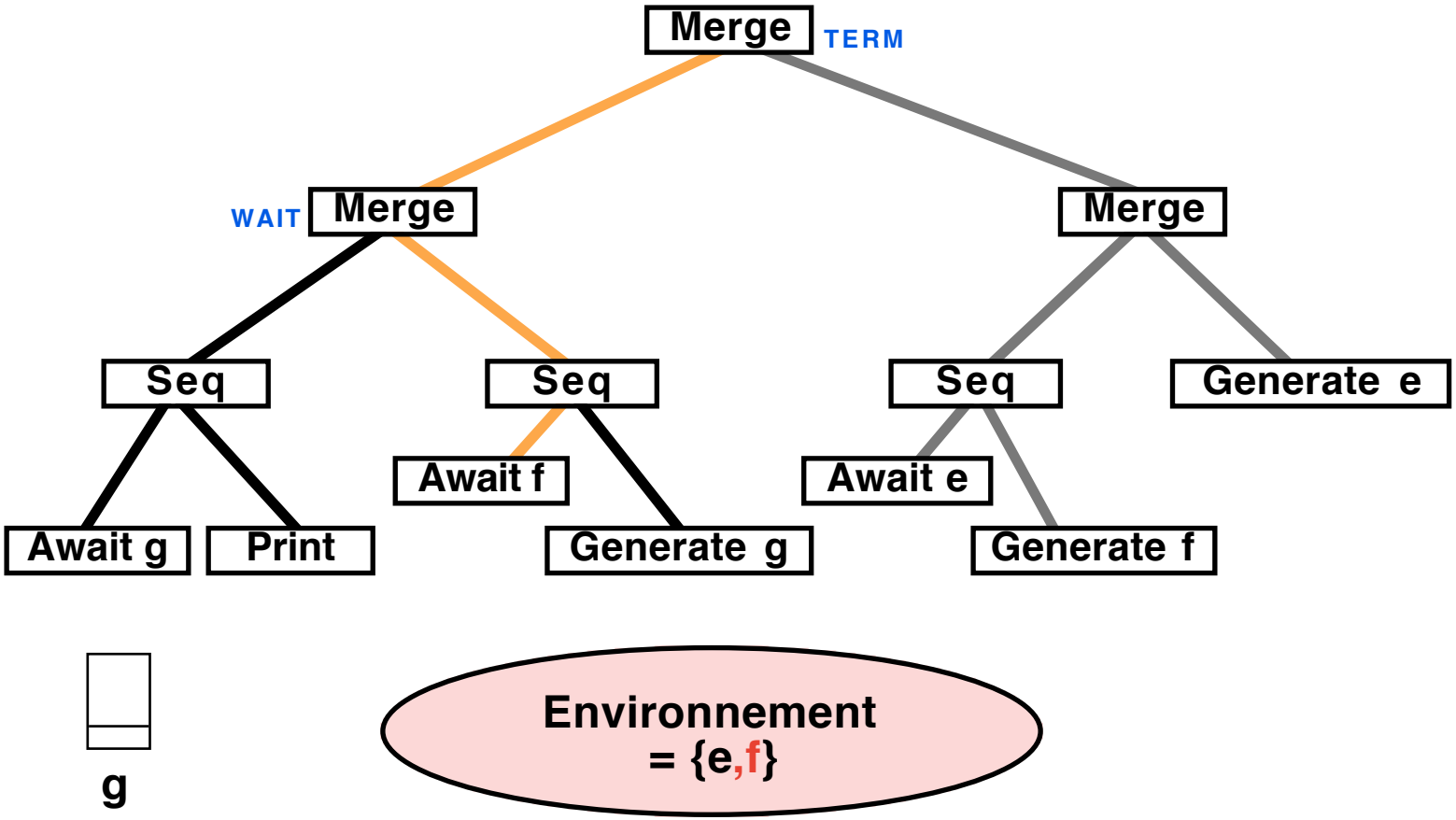
Storm


WAIT
Merge
WAIT
Merge
WAIT
Merge
TERM
Seq
Seq
Seq
Generate e
Await f
Await e
Await g
Print
Generate g
Generate f
g
f
Environnement
= {e}

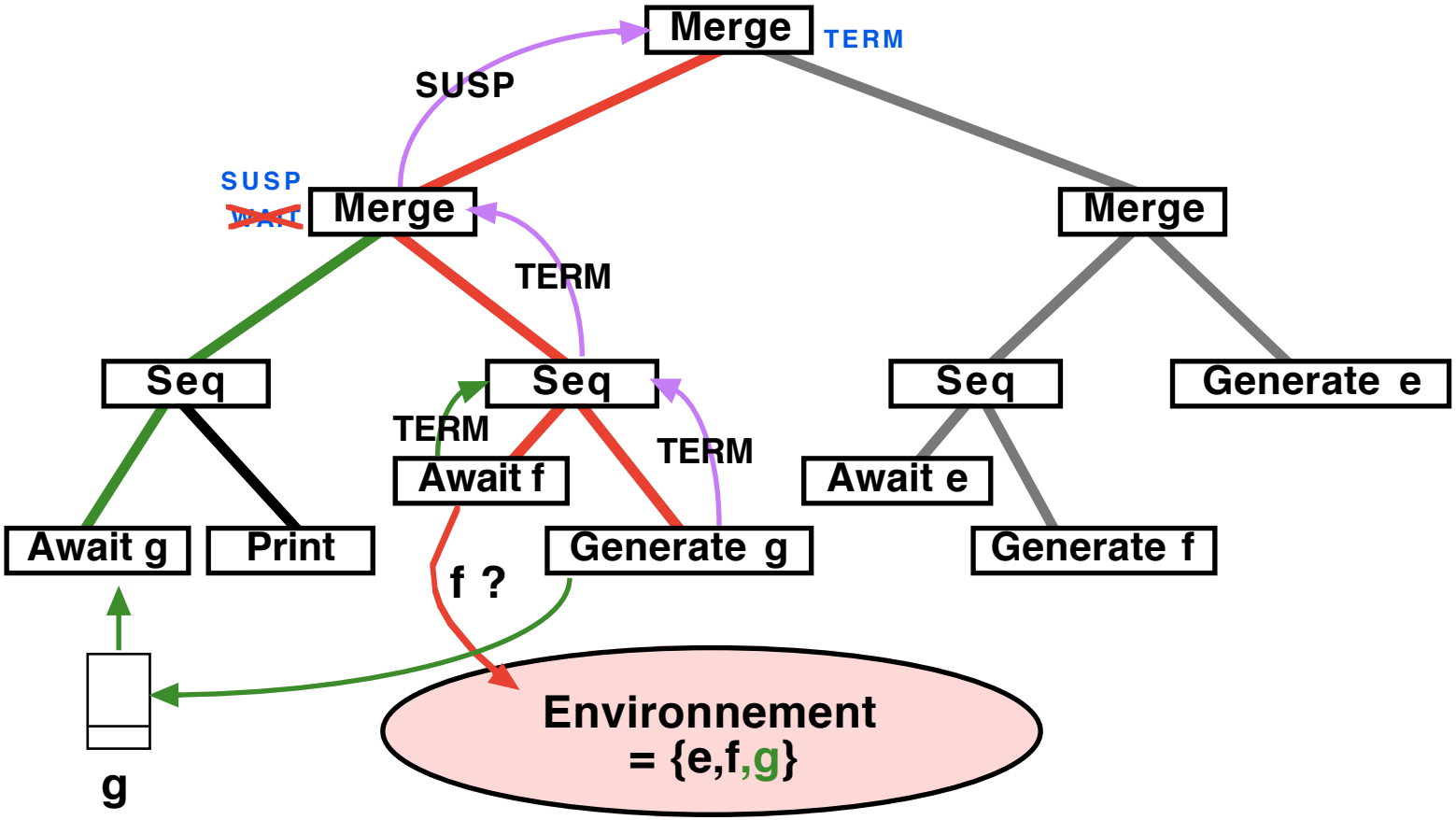
Storm


SUSP
SUSP
WAIT
Merge
TERM
WAIT
Merge
SUSP
WAIT
Merge
TERM
TERM
Seq
Seq
Seq
Generate e
TERM
TERM
TERM
Await g
Print
Await f
Generate g
Await e
Generate f
e ?
Environnement
= {e,f}
f
g

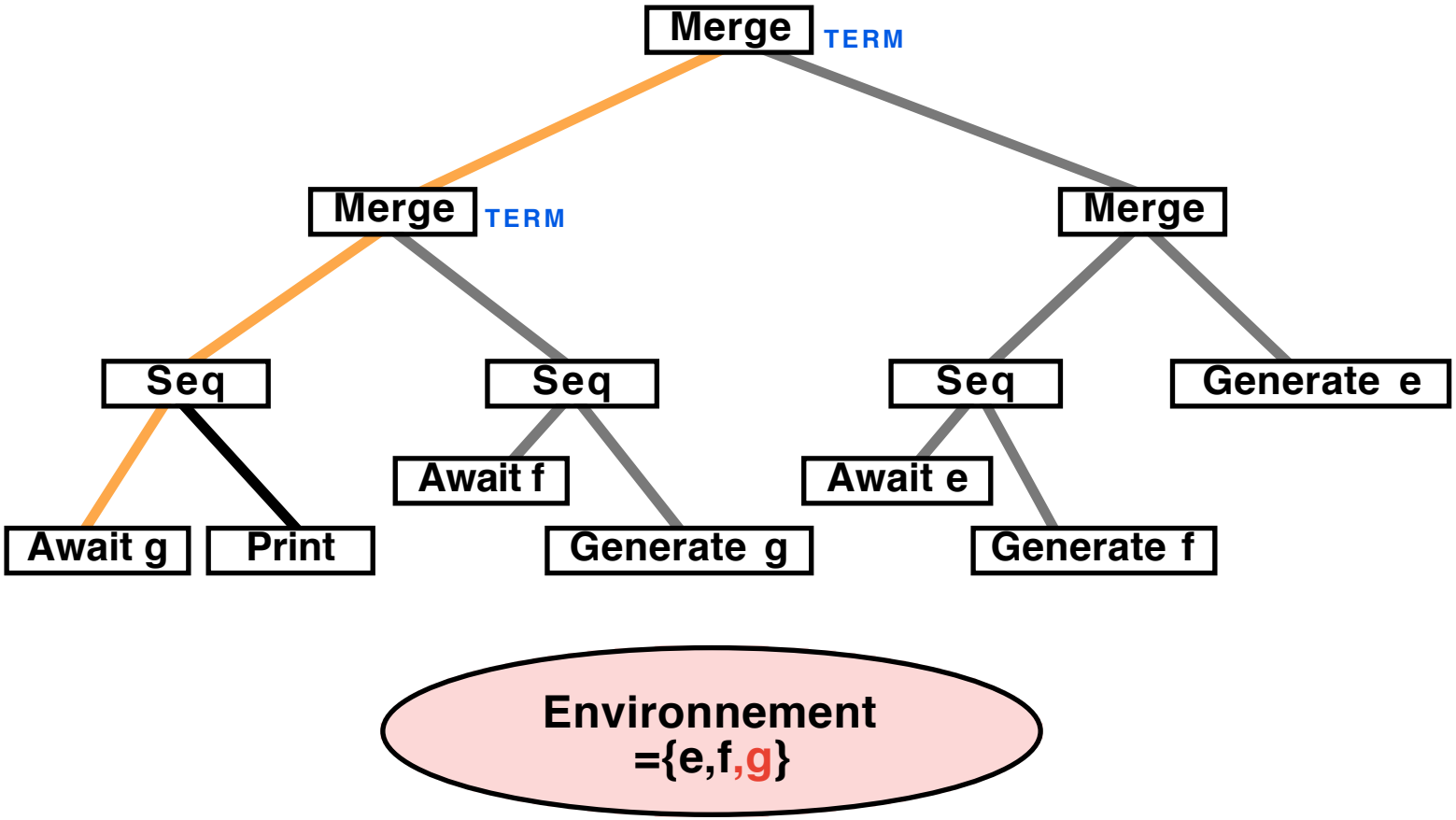
Storm


Merge
TERM
WAIT
Merge
Merge
Seq
Seq
Seq
Generate e
Await f
Await e
Await g
Print
Generate g
Generate f
g
Environnement
= {e,f}

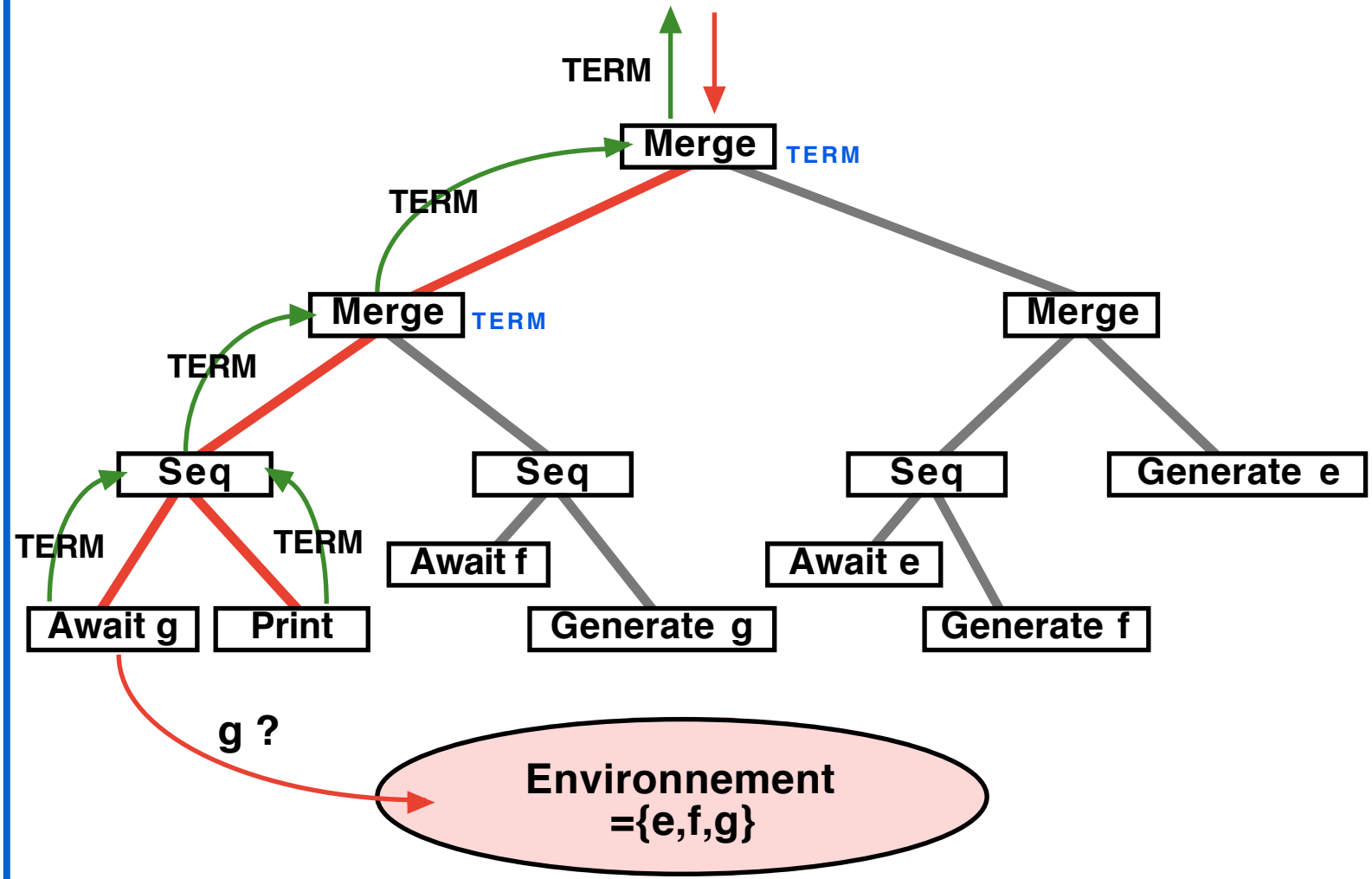
Storm


Merge
TERM
SUSP
SUSP
WAIT
Merge
Merge
TERM
Seq
Seq
Seq
Generate e
TERM
TERM
Await f
Await e
Await g
Print
Generate g
Generate f
f ?
g
Environnement
= {e,f,g}

Storm


Merge
TERM
Merge
TERM
Merge
Seq
Seq
Seq
Generate e
Await f
Await e
Await g
Print
Generate g
Generate f
Environnement
={e,f,g}

Storm


TERM
Merge
TERM
TERM
Merge
TERM
Merge
TERM
Seq
Seq
Seq
Generate e
TERM
TERM
Await f
Await e
Await g
Print
Generate g
Generate f
g ?
Environnement
={e,f,g}

Storm

- Réduit le nombre d'activations inutiles tout en préservant le déterminisme d'exécution gauche-droite

- Exécution toujours descendante

- Formalisation des précurseurs par un second système de règles

- Performances dépendantes de la structure de programme

- Simple : files d'attente inter-instants

	10	50	100	500	1000
REPLACE	12	53	112	1689	6643
Storm	12	41	78	173	287
Simple	11	37	64	149	273

cascade inverse
durée d'un instant en fonction du nombre de composants parallèles (temps mesuré en millisecondes)

Implémentation

Dérivée de Replace (environ 150 lignes de code en plus)

```
abstract public class Instruction implements ...
{
     public EnvironmentImpl env;
     public Instruction parent;
      public void bind(EnvironmentImpl env,Instruction aParent){
          this.env = env; parent = aParent;
     }
      public void setParent(Instruction aParent){ parent = aParent; }

     abstract public byte rewrite();
     abstract public void reset();

...
     public void zap(Instruction from){
           throw new InternalError("Not zappable Instruction");
     }
}
```

instruction parente dans l'arbre du programme

exécution structurelle selon les règles de réécriture

implémentation des précurseurs

par défaut une instruction ne peut pas être le point de départ d'un précurseur

Implémentation

```
public class Par extends BinaryInstruction
{
   public byte leftFlag = SUSP, rightFlag = SUSP;
   public Par(Program left,Program right){ super(left,right); }
   public void reset(){ super.reset(); leftFlag = rightFlag = SUSP; }
...
   public byte rewrite(){
       if(SUSP == leftFlag)||((WAIT == leftFlag)&&env.eoi)){
             leftFlag = left.rewrite();}
       if(SUSP == rightFlag)||((WAIT == rightFlag)&&env.eoi)){
             rightFlag = right.rewrite();}
         if((TERM == leftFlag)&&(TERM == rightFlag)){ return TERM; }
         if((SUSP == leftFlag)||(SUSP == rightFlag)){ return SUSP; }
         if((WAIT == leftFlag)||(WAIT == rightFlag)){ return WAIT; }
        if(STOP == rightFlag){ rightFlag = SUSP; }
        if(STOP == leftFlag){ leftFlag = SUSP; }
       return STOP;
  }
...
   public void zap(Instruction from){
       if(from == left){
           if(WAIT == leftFlag){
               leftFlag = SUSP; if(SUSP != rightFlag){ parent.zap(this); }
           }
       }
       else if(from == right){
           if(WAIT == rightFlag){
               rightFlag = SUSP; if(SUSP != leftFlag){ parent.zap(this); }
           }
       }
         else{ throw new InternalError("Awaked by unknown son"); }
  }
}
```

	SUSP	WAIT	STOP	TERM
SUSP	SUSP	SUSP	SUSP	SUSP
WAIT	SUSP	WAIT	WAIT	WAIT
STOP	SUSP	WAIT	STOP	STOP
TERM	SUSP	WAIT	STOP	TERM

Implémentation

```
public class Presence extends Config implements Precursor
{
    final public IdentifierWrapper wrapper;
   public boolean evaluated = false;
   public Identifier event;
   public boolean posted = false;

    public Presence(IdentifierWrapper wrapper){ this.wrapper = wrapper; }
   public boolean fixed(){
       if(evaluated == false){ event = wrapper.evaluate(env); evaluated = true; }
        if(!(env.isGenerated(event) || env.eoi||posted)){
          env.getEventData(event).postPrecursor(this); posted = true;
     }
       return env.isGenerated(event) || env.eoi;
  }
    public boolean eval(){ return env.isGenerated(event); }
   public void zapFromHere(){
         if(parent instanceof Instruction){
           ((Instruction)parent).zap(null); posted = false;
      }
         else{ ((Config)parent).zap(); }
  }
...
}
```

Implémentation

```
public class EventDataImpl implements EventData, Cloneable
{
    public long generated = 0, lastActualization = 0;
...
    public void generate(long instant,Object val){
        this.generated = instant;
...
       zapPrecursors();
    }
    public boolean isGenerated(long instant){
        return instant == generated;
    }
    PrecursorCell precursorList = null;

    public void postPrecursor(Precursor precursor){
          PrecursorCell cell = new PrecursorCell();
           cell.next = precursorList; precursorList = cell; cell.precursor = precursor;
    }
    public void zapPrecursors(){
          if(null == precursorList){ return; }
          while(null != precursorList){
            precursorList.precursor.zapFromHere();
            precursorList = precursorList.next;
        }
    }
}
```

```
class PrecursorCell{
     PrecursorCell previous;
     PrecursorCell next;
     Precursor precursor;
}
```

Conclusion

Exécution efficace de programmes réactifs => optimisation de l'exécution des opérateurs de parallélisme pour éviter les activations inutiles de composants ne pouvant progresser.

Objectif : support d'un très grand nombre de composants parallèles communiquant par un grand nombre d'événements diffusés.

Permet le dévelopement de modèles de communication entre objets réactifs, basés sur la diffusion d'événements, supportant un grand nombre d'intervenants.