A Compiler Representation for Incremental Parallelization

Christoph Angerer and Thomas Gross
ETH Zurich

Parallel Continuation Passing Style

- Unified Intermediate Representation for:
  - Fully sequential code
  - Parallel code
  - Advanced control flow
- Allows gradual parallelization of sequential programs

Background

- Compilers use suitable internal representations for:
  - Analysis
  - Program Transformations / Optimizations
- Common IRs:
  - Static Single Assignment (SSA)
  - Continuation-Passing Style (CPS)
- Automated translation from source into IR

Problem

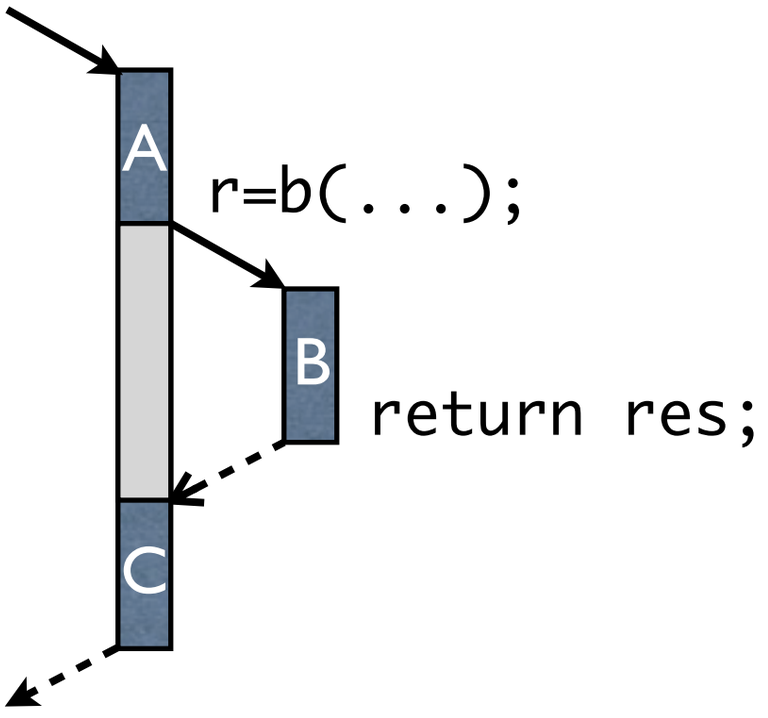
- Current IRs lack support for parallelism:
  - No way for compiler to trade off/select grain size of parallelization
  - No way to incrementally transform sequential programs into parallel versions
- in CPS: There can be only one tail-call

⇒ It is impossible to fork computation

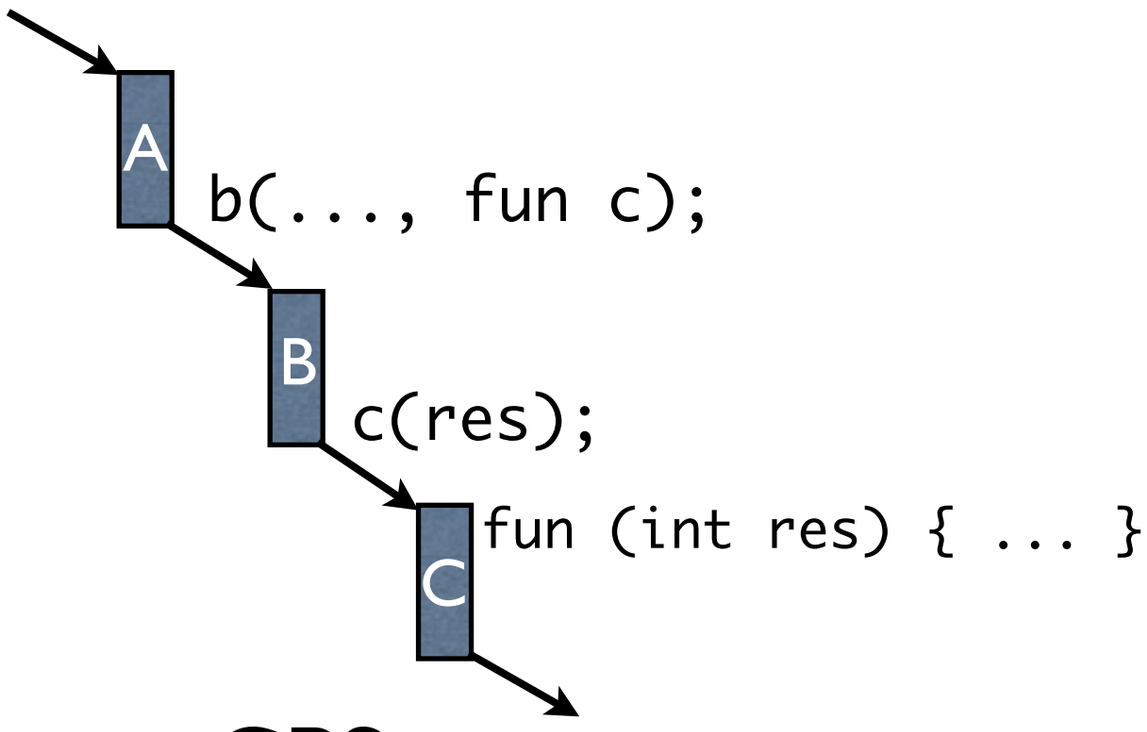
  - (in the IR)

Brief Review of CPS

- A function never returns to its caller
- Instead, a function expects a continuation function as an additional parameter
- A return is replaced with a (tail-)call to this continuation, passing the result value



call/return



CPS

fib with call/return

```
int fib(int k) {

   if (k <= 2)
      return 1;
   else
      return fib(k-1) + fib(k-2);

}
```

fib with call/return

```
int fib(int k) {

  if (k <= 2)
    return 1;
  else
    return fib(k-1) + fib(k-2);

}
```

CPS Example

```
fun fib(int k, fun ret) {

  if (k <= 2)
    ret(1);
  else
    ret(
      //fib(k-1) + fib(k-2)
    );
}
```

CPS Example

```
fun fib(int k, fun ret) {

  if (k <= 2)
    ret(1);
  else
    ret(
      //fib(k-1) + fib(k-2)
    );
}
```

CPS Example

```
fun fib(int k, fun ret) {

  if (k <= 2)
    ret(1);
  else
    ret(
      //fib(k-1) + fib(k-2)
    );
}
```

1 3 2

CPS Example

```
fun fib(int k, fun ret) {

  if (k <= 2)
    ret(1);
  else
    //fib(k-1) → left          1
      //fib(k-2) → right       2
        ret(left + right);     3
}
```

CPS Example

```
fun fib(int k, fun ret) {

  if (k <= 2)
    ret(1);
  else
    fib(k-1, fun(left) {     //fib(k-1) → left
      fib(k-2, fun(right) { //fib(k-2) → right
        ret(left + right);
    }))
}
```

Basic Idea of pCPS

- Relax tail-call restriction
  - Allow more than one successor
  - Enable forking of computation
- Explicit happens-before relationships
  - Part of the IR
  - Can be analyzed and changed by the compiler

Parallel CPS

```
class Main {
   task t() {
      schedule(this.foo());
   }

   task foo() { ... }
   ...
}
```

Parallel CPS

```
class Main {
   task t() {
      schedule(this.foo());
   }

   task foo() { ... }
   ...
}
```

Parallel CPS

```
class Main {
  task t() {
     schedule(this.foo());
   }

  task foo() { ... }
     ...
}
```

Parallel CPS

```
class Main {
   task t() {
      schedule(this.foo());
   }

   task foo() { ... }
   ...
}
```

Parallel CPS

```
class Main {
   task t() {
      schedule(this.foo());
      schedule(this.bar(42));
   }
   task foo() { ... }
   task bar(int x) { ... }
   ...
}
```

Parallel CPS

```
class Main {
   task t() {
      schedule(this.foo());
      schedule(this.bar(42));
   }
   task foo() { ... }
   task bar(int x) { ... }
   ...
}
```

Parallel CPS

```
class Main {
   task t() {
      schedule(this.foo());
      schedule(this.bar(42));
   }
   task foo() { ... }
   task bar(int x) { ... }
   ...
}
```

Parallel CPS

```
class Main {
  task t() {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

Parallel CPS

```
class Main {
  task t() {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

Parallel CPS

```
class Main {
  task t() {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

Parallel CPS

```
class Main {
  task t() {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

Parallel CPS

```
class Main {
  task t() {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));
    a → c;
    b → c;
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

Parallel CPS

```
class Main {
  task t() {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));

    a → c;
    b → c;
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

Parallel CPS

```
class Main {
  task t() {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));
    a → c;
    b → c;
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

Parallel CPS

```
class Main {
  task t(Activation later) {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));
    a → c;
    b → c;
    c → later;
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

Parallel CPS

```
class Main {
  task t(Activation later) {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));
    a → c;
    b → c;
    c → later;
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

Parallel CPS

```
class Main {
  task t(Activation later) {
    Activation a = schedule(this.foo());
    Activation b = schedule(this.bar(42));
    Activation c = schedule(this.blubb(a, b));
    a → c;
    b → c;
    c → later;
  }

  task foo() { ... }
  task bar(int x) { ... }
  task blubb(Activation x, Activation y) { ... }
  ...
}
```

pCPS in a Nutshell

- A `task` is similar to a method:
  - code that is executed in the context of `this`
- Instead of *calling* a task, one *schedules* it for later execution:

```
Activation b = schedule(this.bar(42));
```

- →-Statement creates explicit happens-before relationship:

```
a → b;
```

pCPS in a Nutshell (2)

- The currently executing Activation is accessible through the keyword now
- Implicit happens-before relationship between now and a newly scheduled task:

```
Activation a = schedule(this.bar(42));

/* implicit: now → a; */
```

- At runtime, a scheduler constantly chooses executable ⟨object, task()⟩ pairs (Activations)

fib in pCPS

```
task fib(int k, Activation later) {
   ...
   } else {
      //make sum available in closures
      Activation sum;

      Activation left  = schedule(fib(k-1, sum));
      Activation right = schedule(fib(k-2, sum));

      sum = schedule(this.sum(left, right));

      left  → right; //left-to-right evaluation
      right → sum;
      sum   → later;
   }
}

task sum(Activation left,
         Activation right) { ... }
```

fib in pCPS

```
task fib(int k, Activation later) {
   ...
   } else {
      //make sum available in closures
      Activation sum;

      Activation left  = schedule(fib(k-1, sum));
      Activation right = schedule(fib(k-2, sum));

      sum = schedule(this.sum(left, right));

      left  → right; //left-to-right evaluation
      right → sum;
      sum   → later;
   }
}

task sum(Activation left,
         Activation right) { ... }
```

fib in pCPS

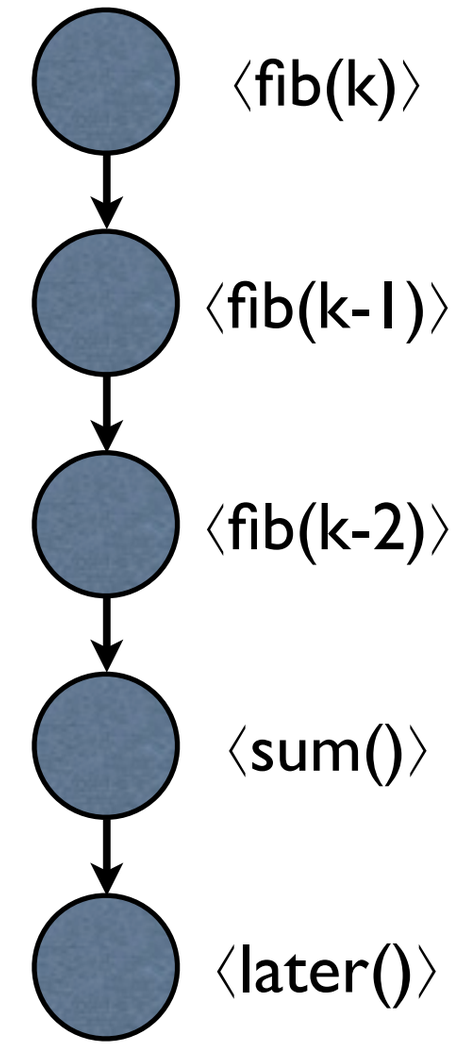
```
task fib(int k, Activation later) {
   ...
   } else {
      //make sum available in closures
      Activation sum;

      Activation left  = schedule(fib(k-1, sum));
      Activation right = schedule(fib(k-2, sum));

      sum = schedule(this.sum(left, right));

      left  → right; //left-to-right evaluation
      right → sum;
      sum   → later;
   }
}

task sum(Activation left,
         Activation right) { ... }
```

fib in pCPS

```
task fib(int k, Activation later) {
   ...
   } else {
      //make sum available in closures
      Activation sum;

      Activation left  = schedule(fib(k-1, sum));
      Activation right = schedule(fib(k-2, sum));

      sum = schedule(this.sum(left, right));

      left  → right; //left-to-right evaluation
      right → sum;
      sum   → later;
   }
}

task sum(Activation left,
         Activation right) { ... }
```

fib in pCPS

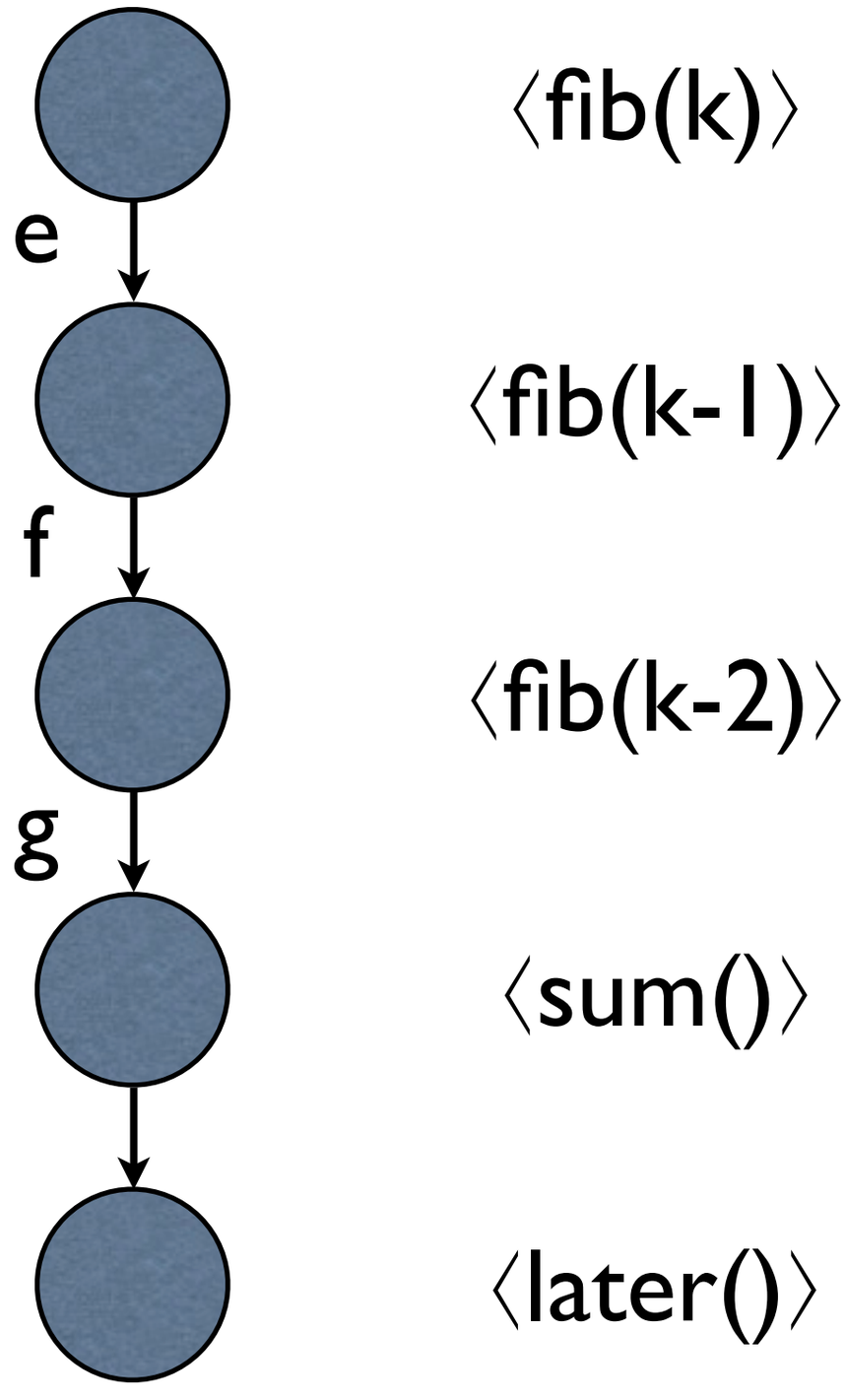
```
task fib(int k, Activation later) {
   ...
   } else {
      //make sum available in closures
      Activation sum;

      Activation left  = schedule(fib(k-1, sum));
      Activation right = schedule(fib(k-2, sum));

      sum = schedule(this.sum(left, right));

      left  → right; //left-to-right evaluation
      right → sum;
      sum   → later;
   }
}

task sum(Activation left,
         Activation right) { ... }
```



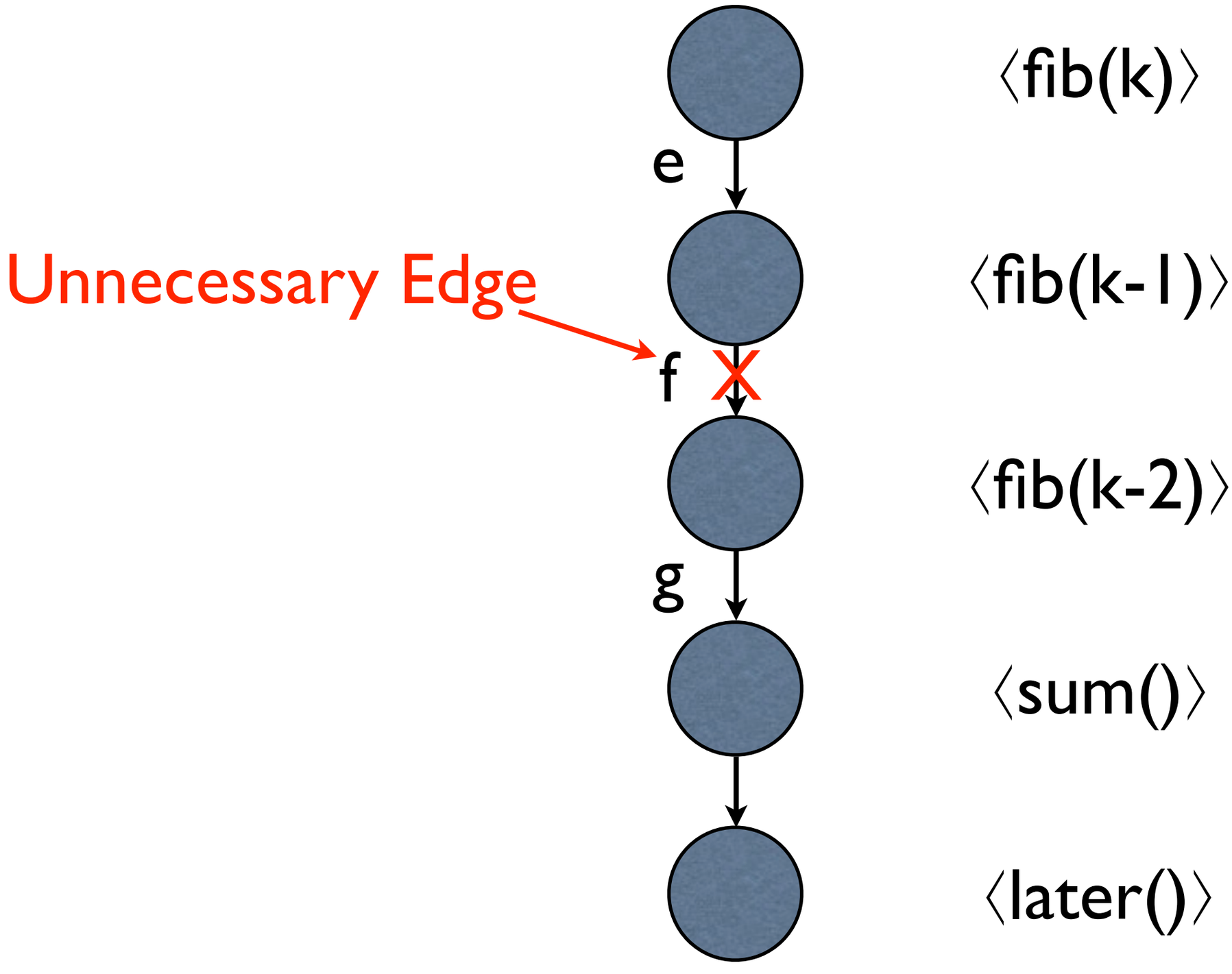
pCPS as an IR

- Compiler can gradually increase/decrease parallelism
  - By adding/removing happens-before relationships
  - By combining/splitting tasks
  - Programmer may provide annotations to allow/disallow/support certain optimizations
- in fib(): computation of left and right hand side of the + can be done in parallel

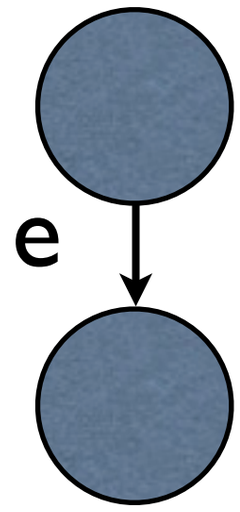
Removing Unnecessary Edges



Removing Unnecessary Edges

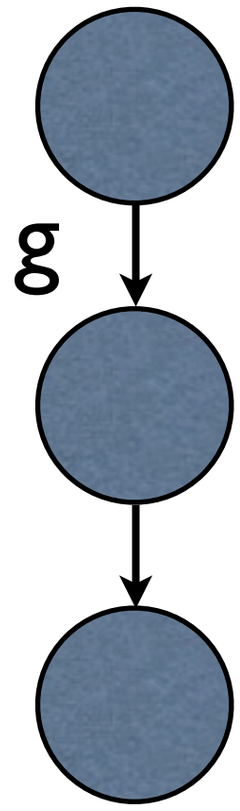


Removing Unnecessary Edges



⟨fib(k)⟩

⟨fib(k-1)⟩



⟨fib(k-2)⟩

⟨sum()⟩

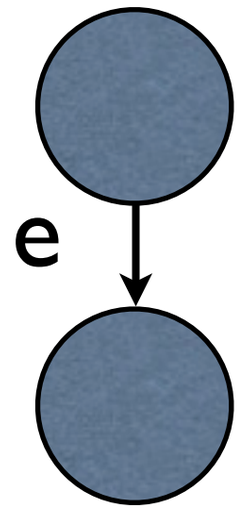
⟨later()⟩

Removing Unnecessary Edges

Need to fix transitive ordering:

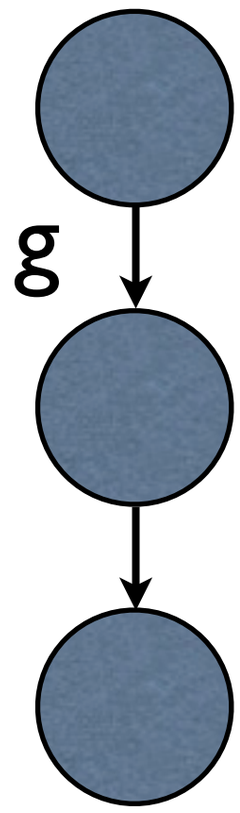
⟨fib(k)⟩ → ⟨fib(k-2)⟩

⟨fib(k-1)⟩ → ⟨sum()⟩



⟨fib(k)⟩

⟨fib(k-1)⟩

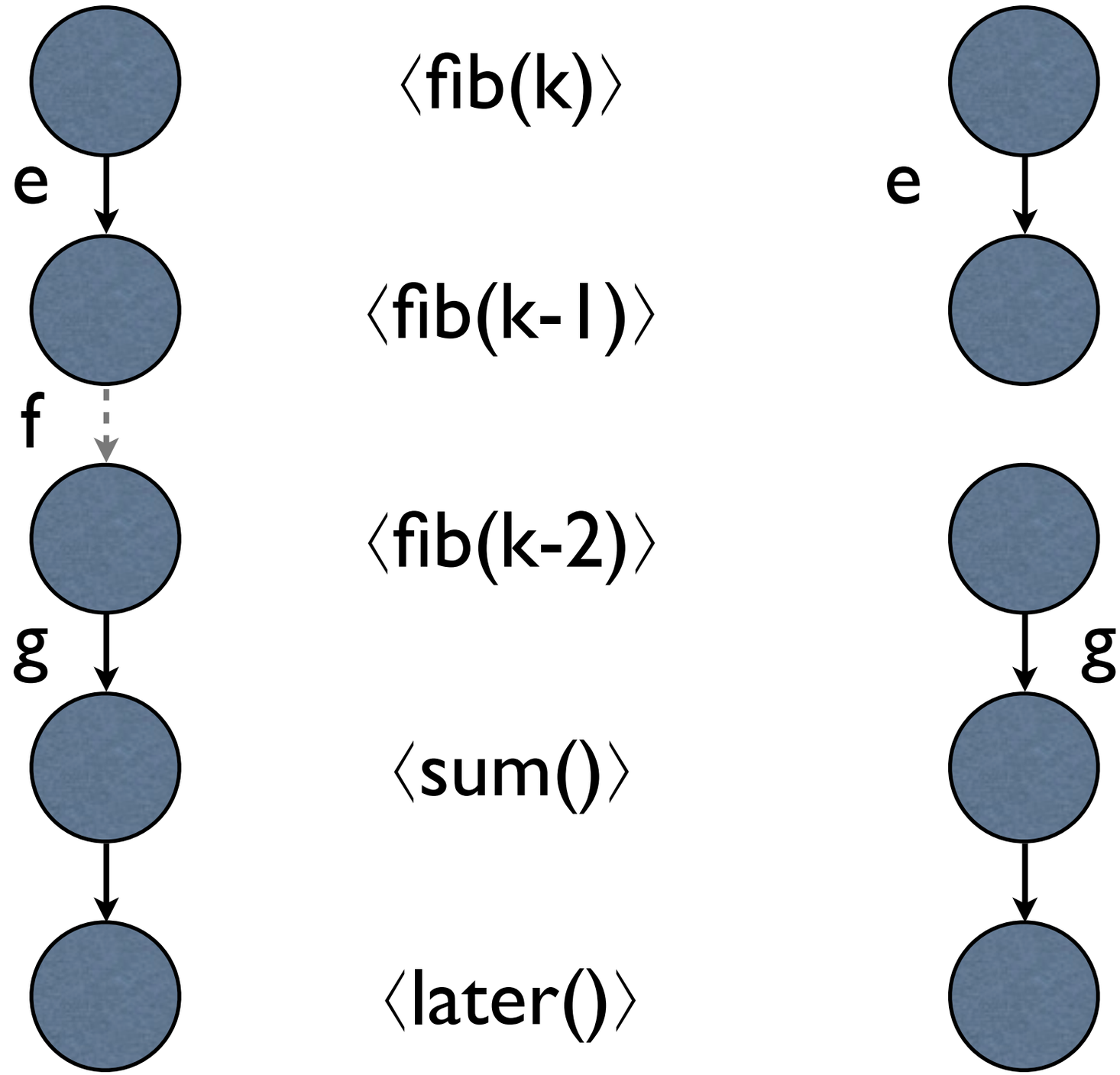


⟨fib(k-2)⟩

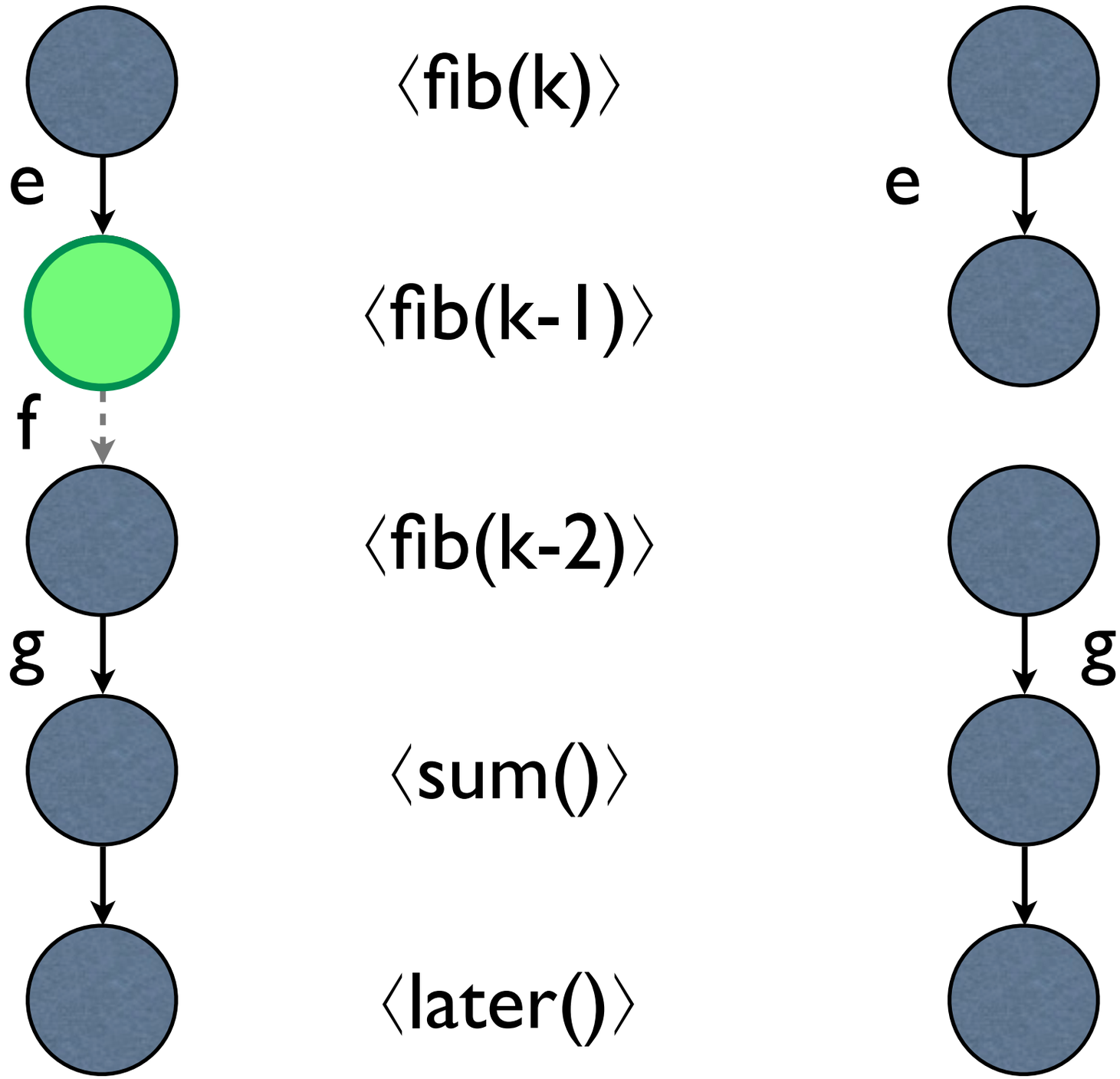
⟨sum()⟩

⟨later()⟩

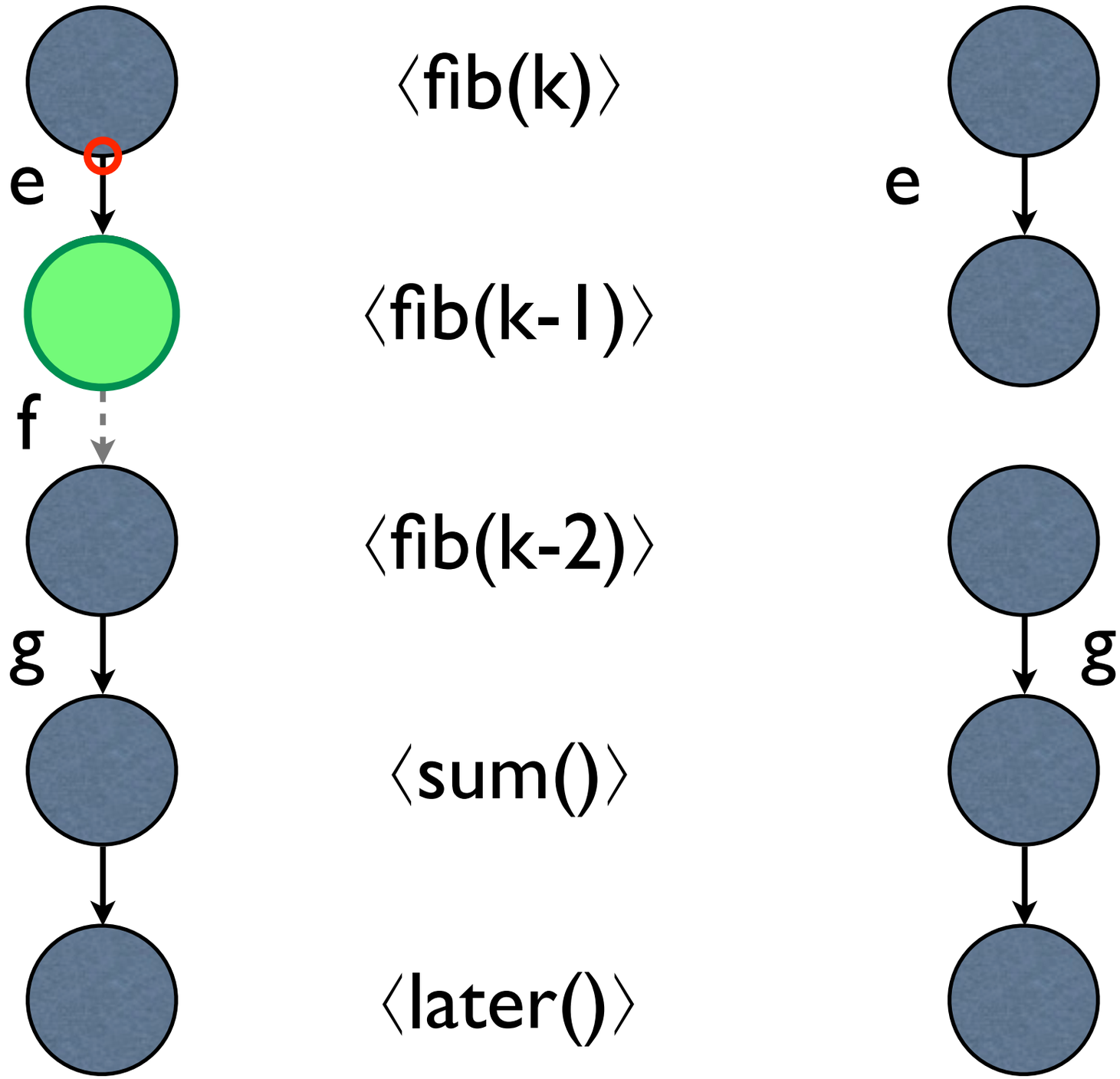
Fix Transitive Ordering



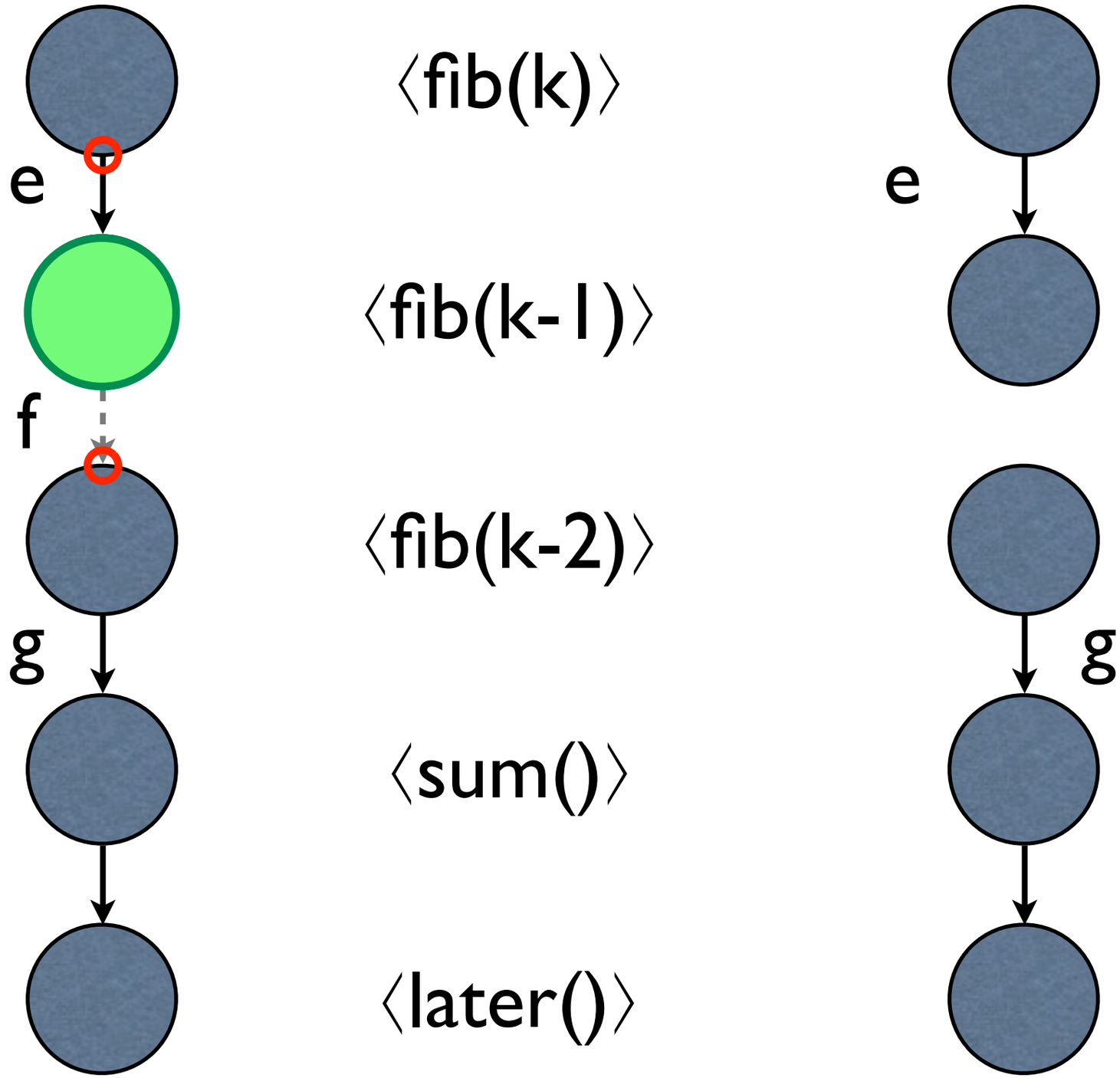
Fix Transitive Ordering



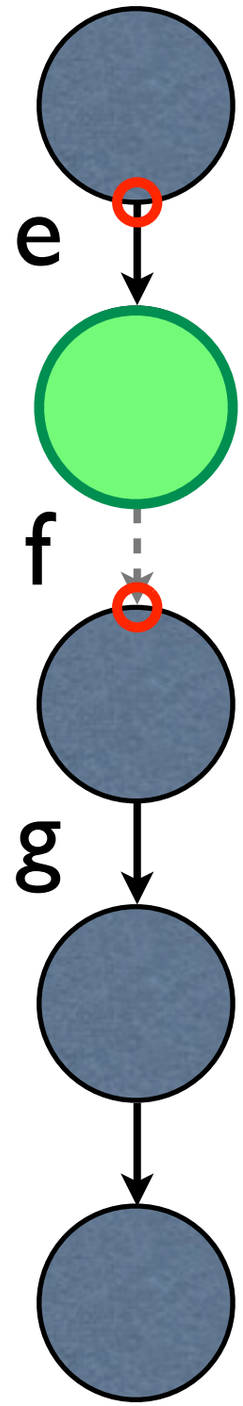
Fix Transitive Ordering



Fix Transitive Ordering



Fix Transitive Ordering



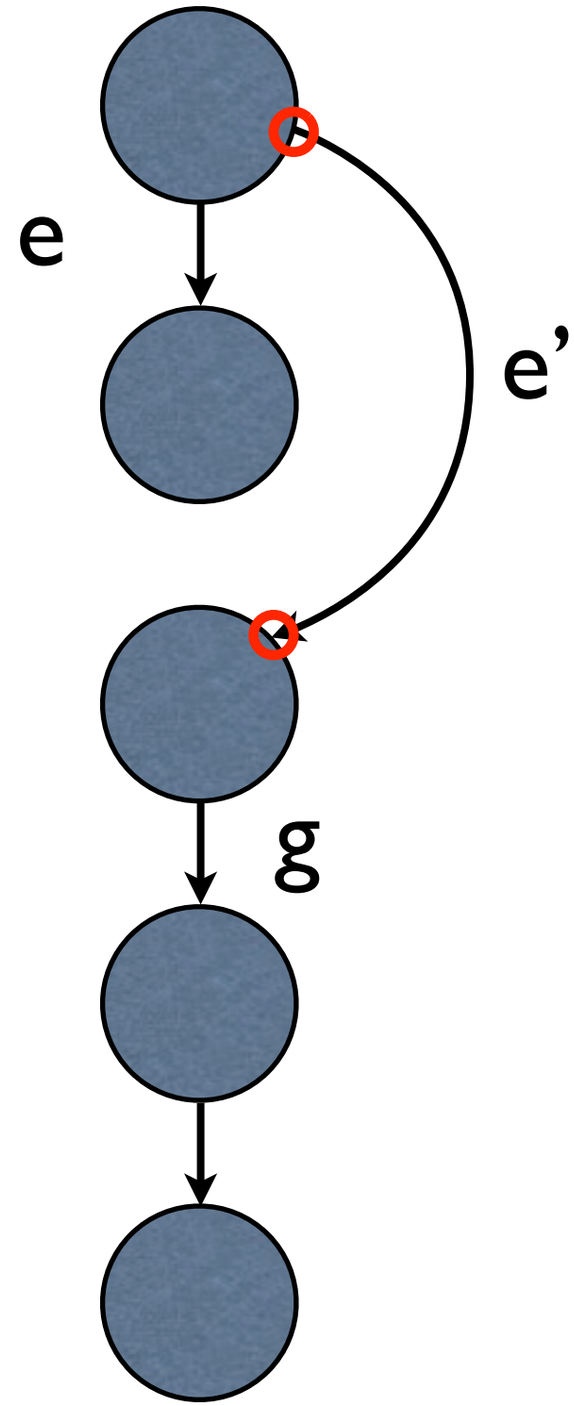
⟨fib(k)⟩

⟨fib(k-1)⟩

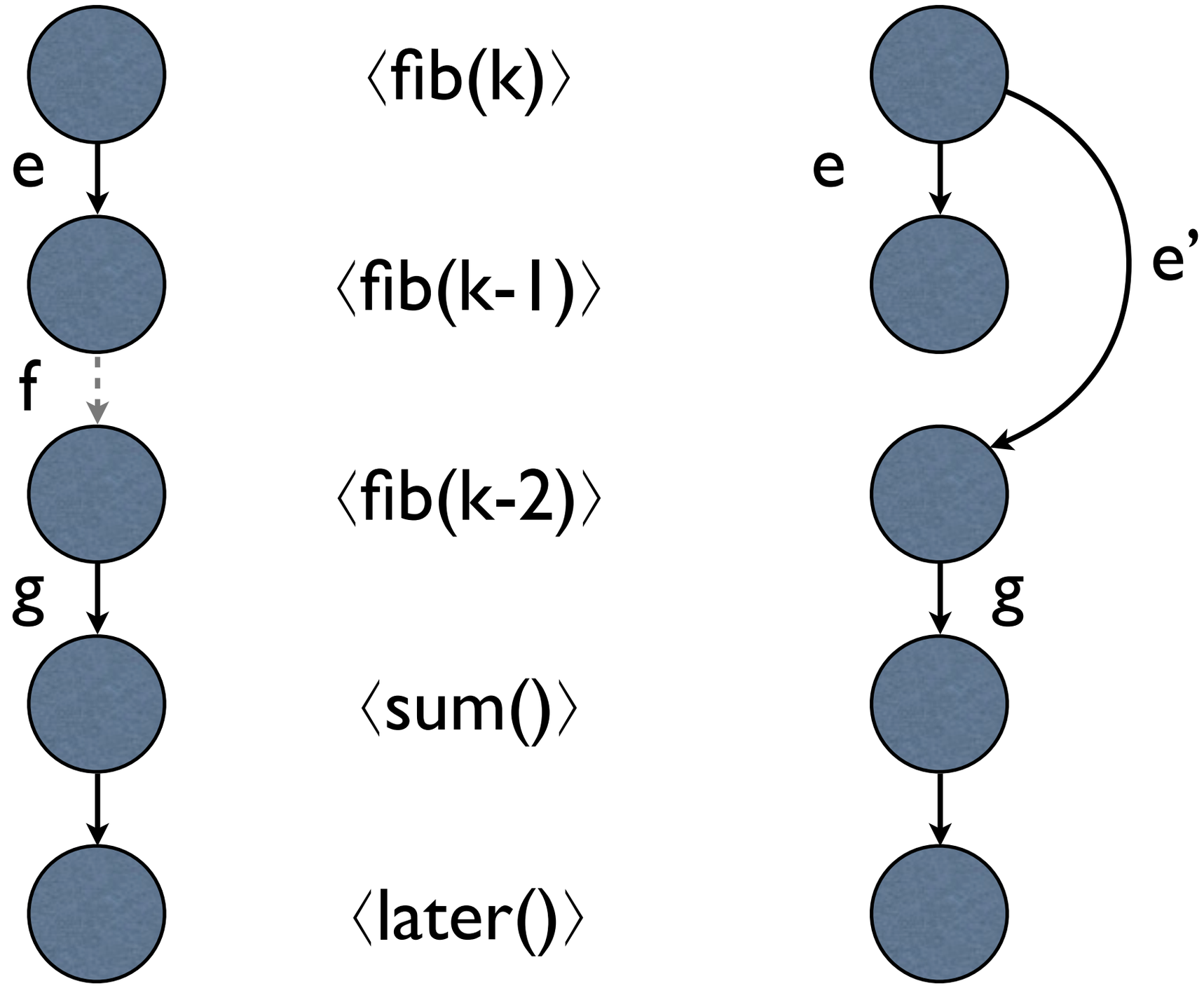
⟨fib(k-2)⟩

⟨sum()⟩

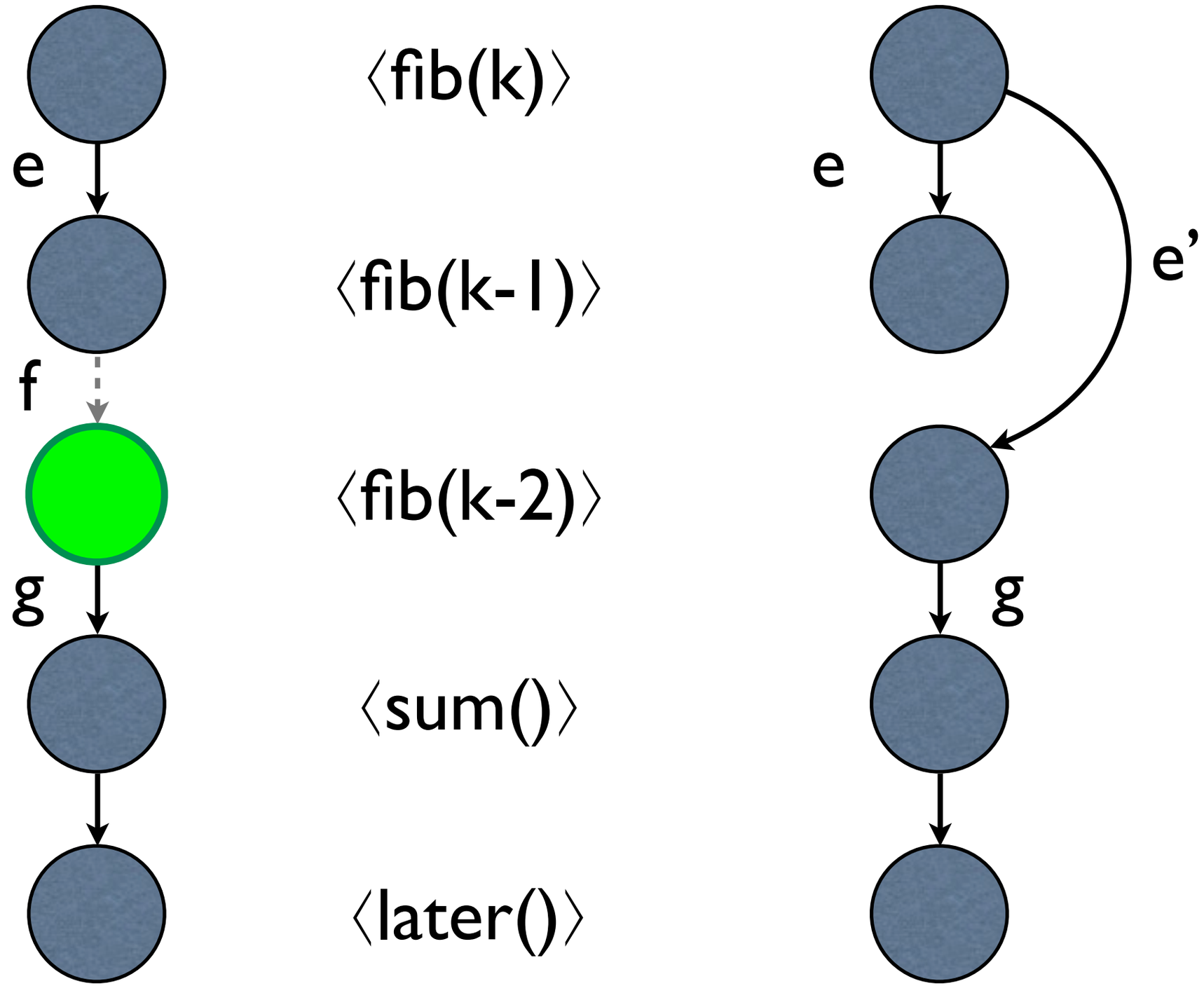
⟨later()⟩



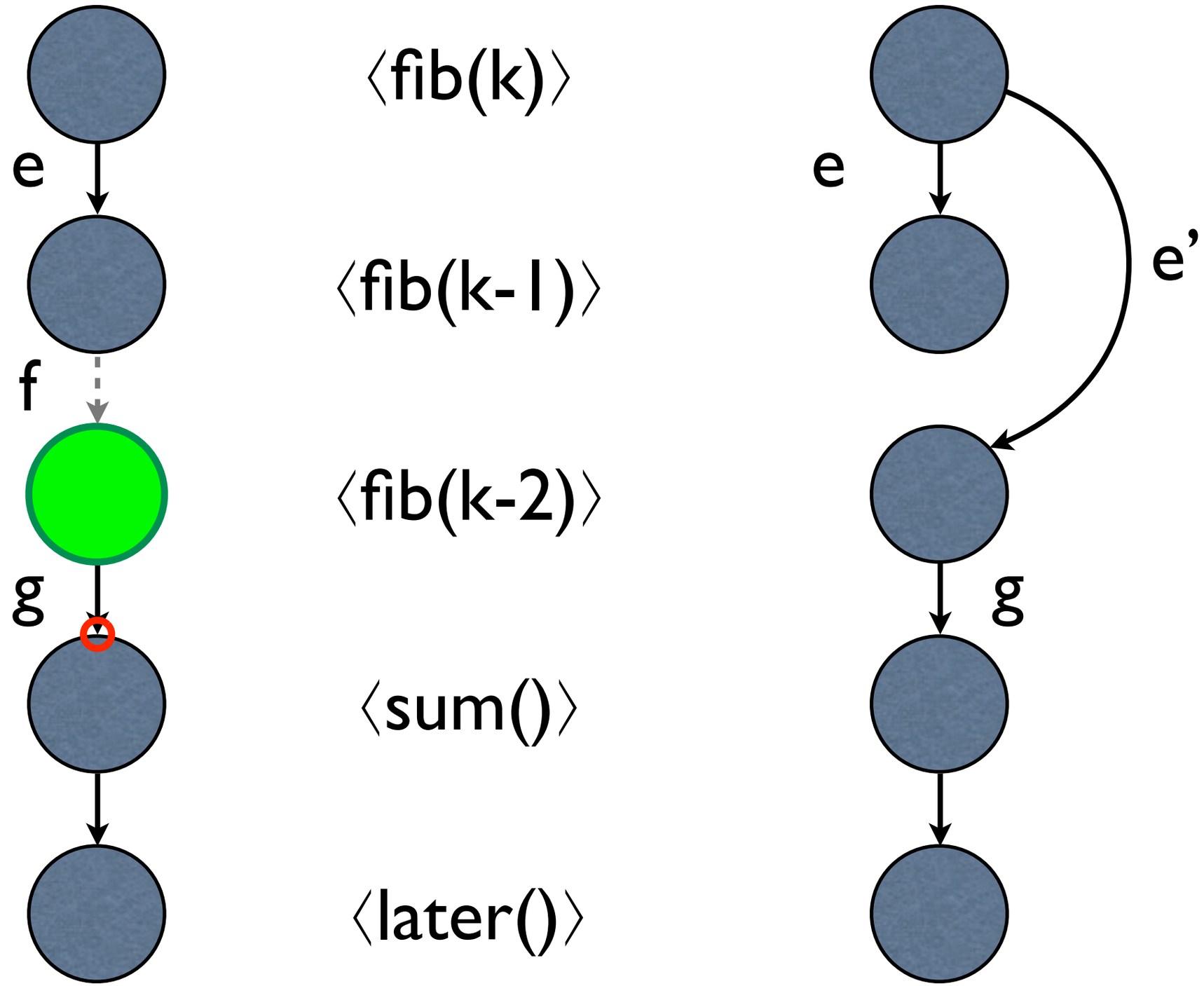
Fix Transitive Ordering



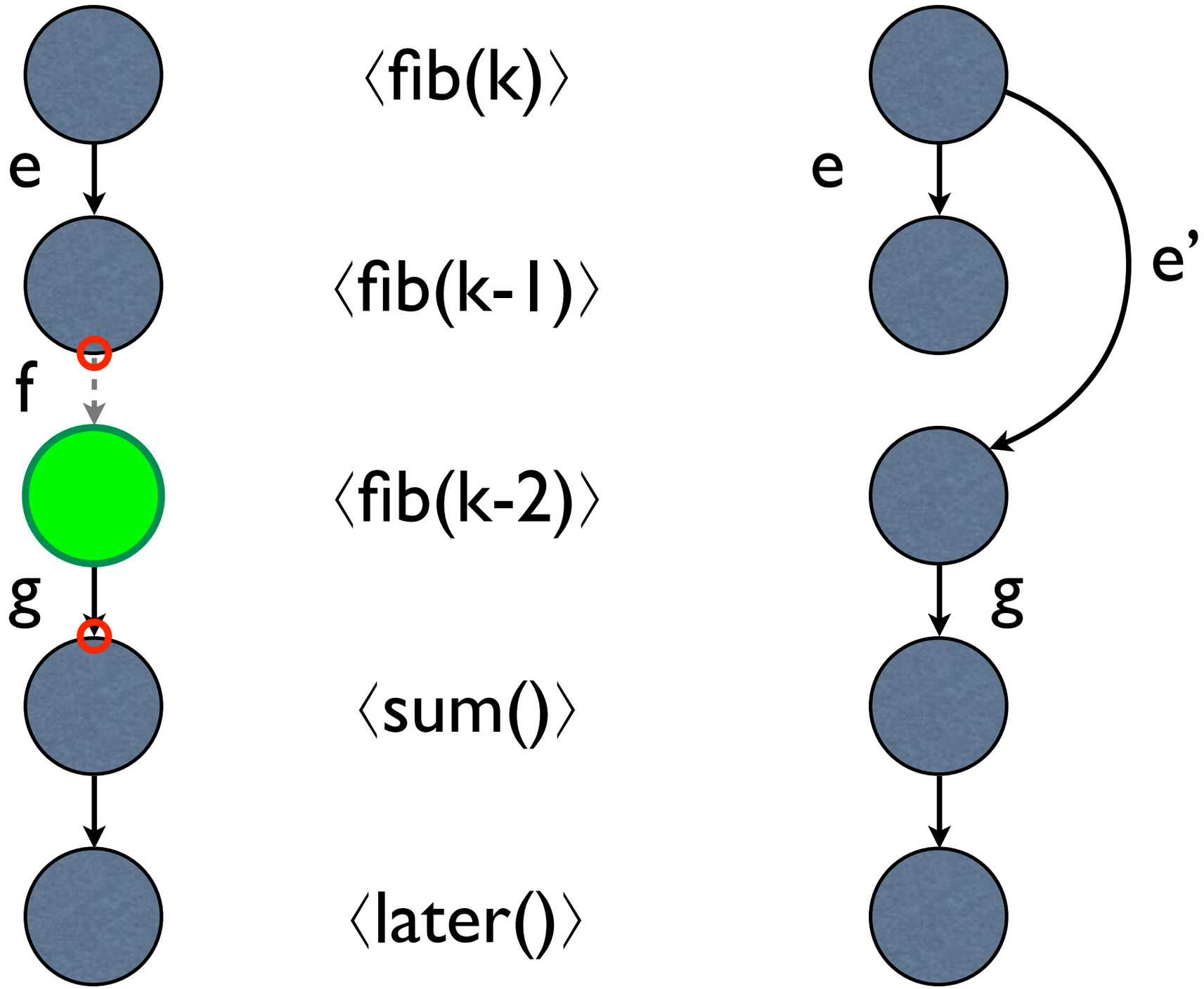
Fix Transitive Ordering



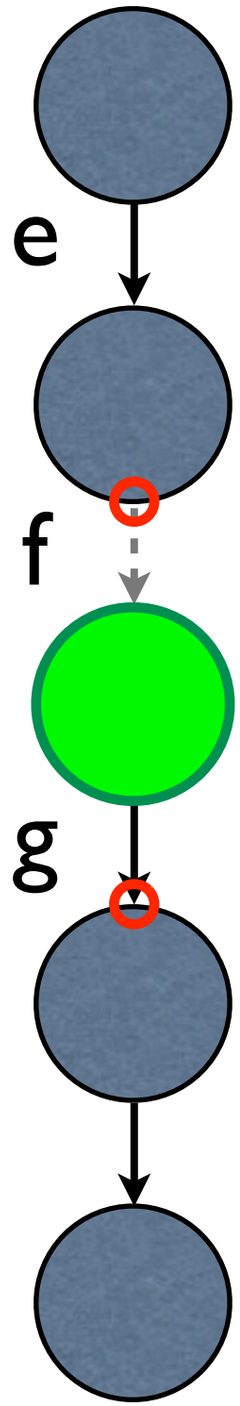
Fix Transitive Ordering



Fix Transitive Ordering



Fix Transitive Ordering



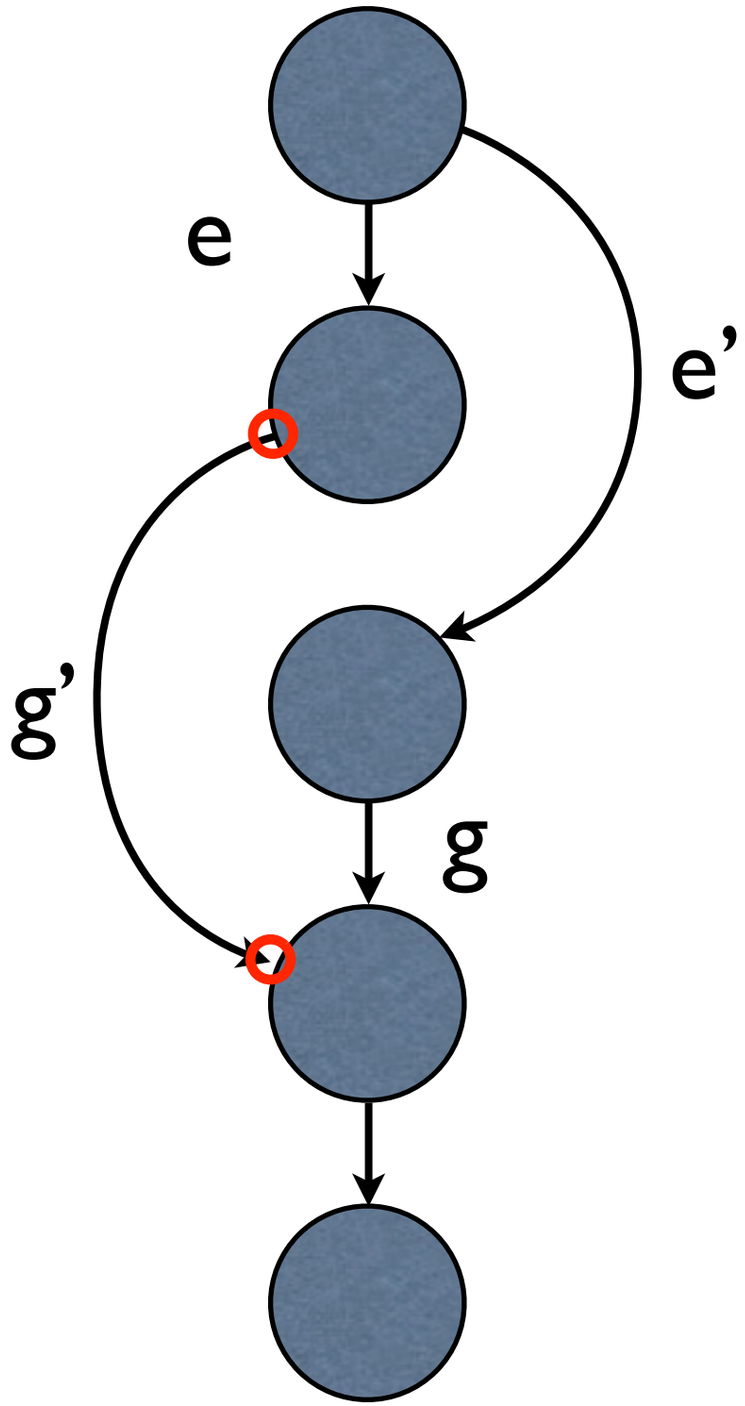
⟨fib(k)⟩

⟨fib(k-1)⟩

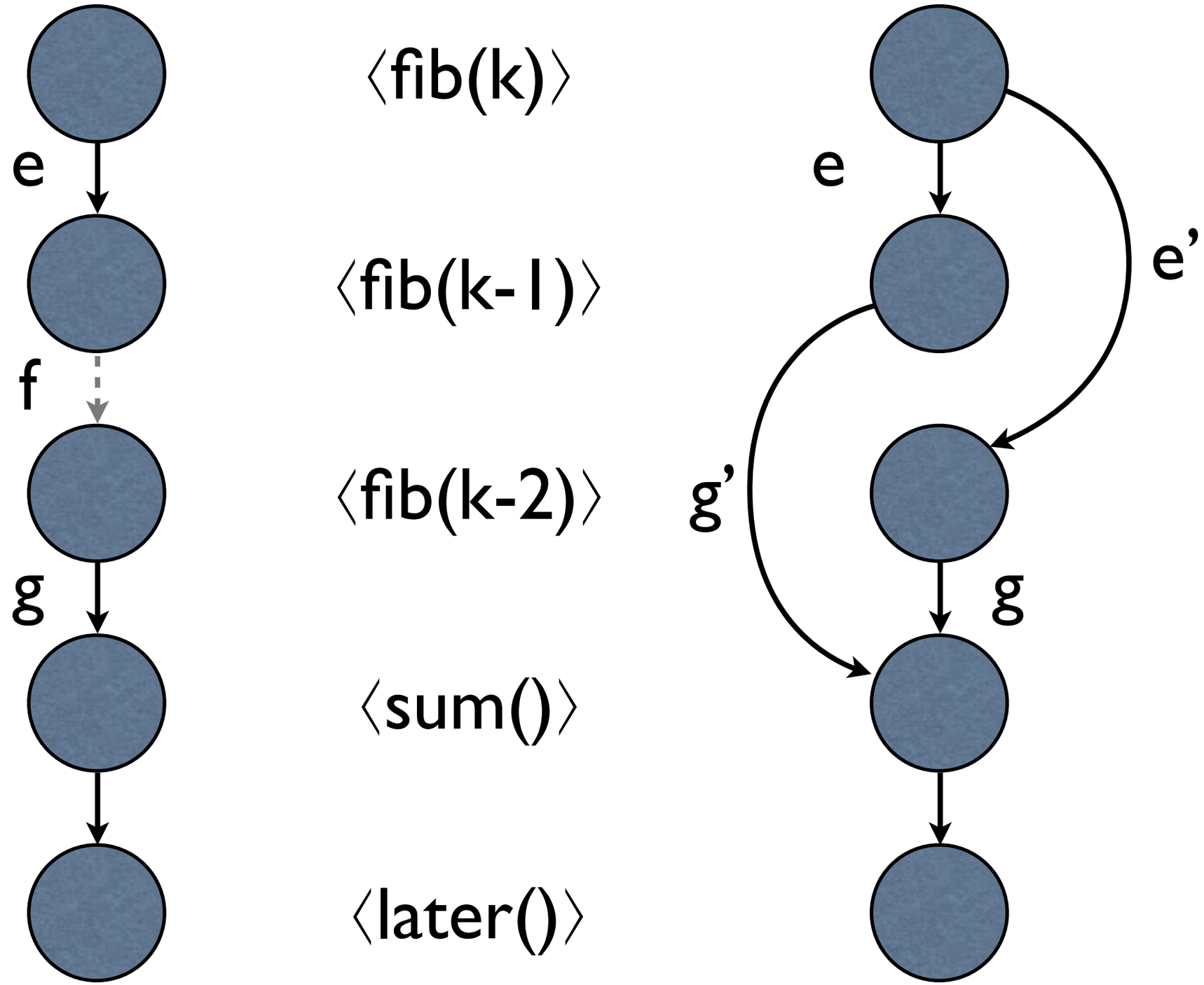
⟨fib(k-2)⟩

⟨sum()⟩

⟨later()⟩



Fix Transitive Ordering



Related Work

- SSA for Parallel Programs
  - J. Lee, S. Midkiff, and D. A. Padua. Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs
  - H. Srinivasan, J. Hook, and M. Wolfe. Static Single Assignment Form for Explicitly Parallel Programs
- OpenMP and Cilk
  - K. Randall. Cilk: Efficient Multithreaded Computing.
- Erbium
  - C. Miranda, P. Dumont, A. Cohen, M. Duranton, and A. Pop. Erbium: A Deterministic, Concurrent Intermediate Representation for Portable and Scalable Performance

Concluding Remarks

- Current compiler representations lack support for parallel constructs
- pCPS allows a compiler to incrementally increase (and decrease) parallelism
  - Starting from a sequential program
  - By adding/removing edges
  - By combining/splitting tasks
- Different independent optimizations can be integrated into a single optimizing compiler

Questions?