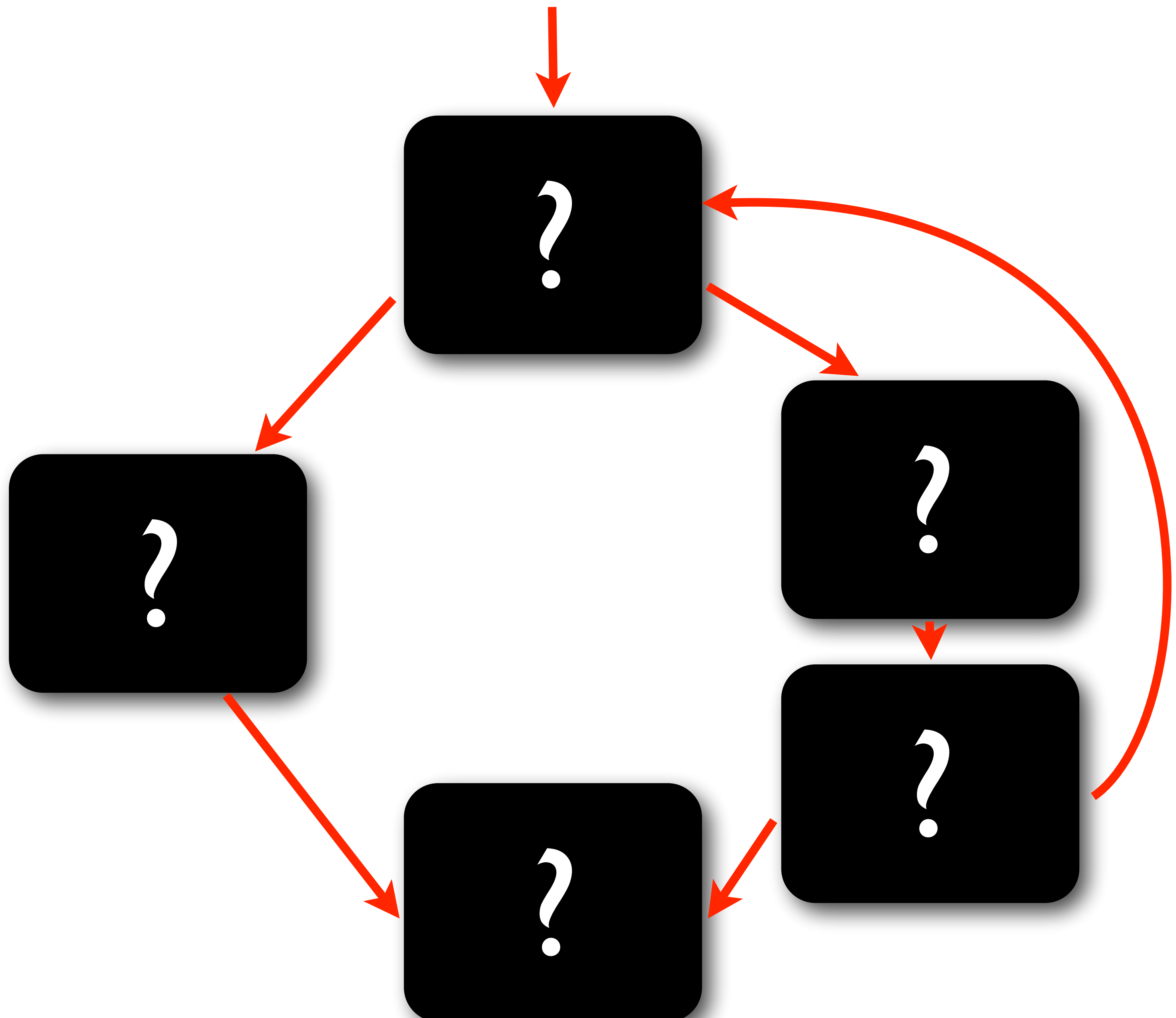
Lecture 2
Unreachable-code & -procedure elimination

Control-flow analysis



Discovering information about how *control* (e.g. the program counter) may move through a program.

Intra-procedural analysis

An *intra-procedural* analysis collects information about the code inside a single procedure.

We may repeat it many times (i.e. once per procedure), but information is only propagated within the boundaries of each procedure, not between procedures.

One example of an intra-procedural control-flow optimisation (an analysis and an accompanying transformation) is *unreachable-code elimination*.

Dead vs. unreachable code

```
int f(int x, int y) {
  int z = x * y;   DEAD
  return x + y;
}
```

Dead code computes unused values.
(Waste of time.)

Dead vs. unreachable code

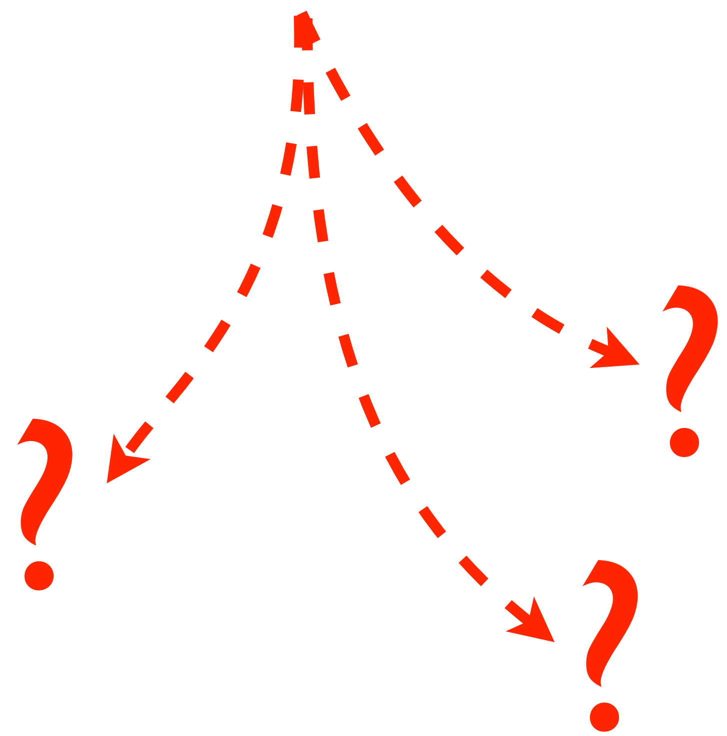
```
int f(int x, int y) {
  return x + y;
  int z = x * y;    UNREACHABLE
}
```

Unreachable code cannot possibly be executed.
(Waste of space.)

Dead vs. unreachable code

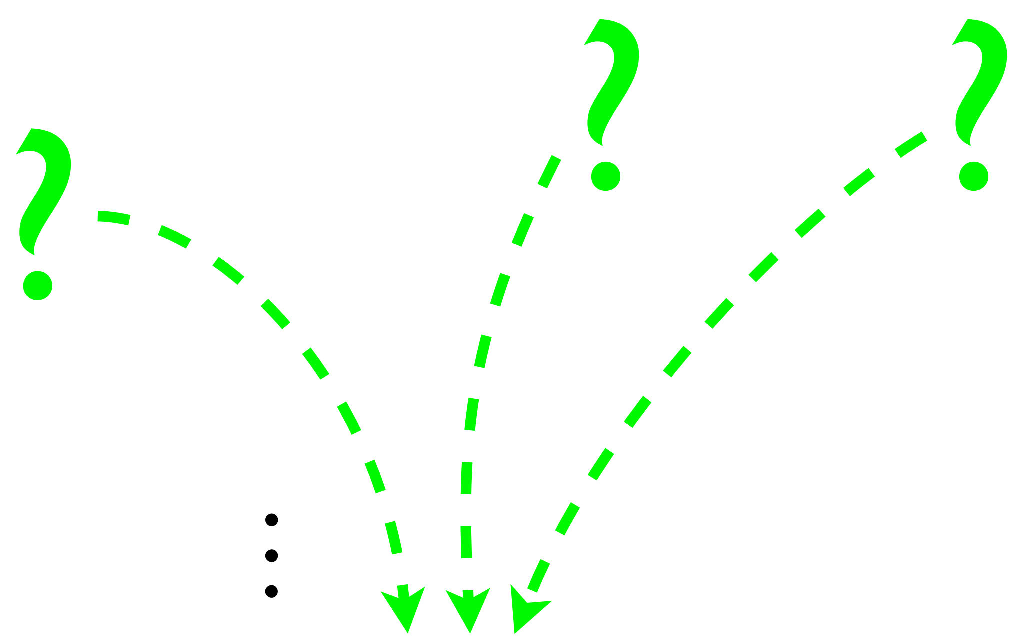
Deadness is a *data-flow* property:
“May this data ever arrive anywhere?”

```
int f(int x, int y) {
   int z = x * y;
   ⋮
```



Dead vs. unreachable code

Unreachability is a *control-flow* property:
"May control ever arrive here?"




```
    ⋮
    int z = x * y;
}
```

Safety of analysis

```
int f(int x, int y) {
   if (g(x)) {
     int z = x * y;
   }
   return x + y;
}
```

UNREACHABLE?

```
bool g(int x) {
   return false;
}
```

✓

Safety of analysis

```
int f(int x, int y) {
   if (g(x)) {
      int z = x * y;
   }
   return x + y;
}
```

UNREACHABLE?

```
bool g(int x) {
   return ...x...;
}
```

?

Safety of analysis

```
int f(int x, int y) {
  if (g(x)) {
    int z = x * y;
  }
  return x + y;
}
```

UNREACHABLE?

In general, this is undecidable.
(Arithmetic is undecidable; cf. halting problem.)

Safety of analysis

- Many interesting properties of programs are undecidable and cannot be computed precisely...
- ...so they must be *approximated*.
- A broken program is much worse than an inefficient one...
- ...so we must err on the side of *safety*.

Safety of analysis

- If we decide that code is unreachable then we may do something dangerous (e.g. remove it!)...
- ...so the safe strategy is to *overestimate* reachability.
- If we can't easily tell whether code is reachable, we just assume that it is. (This is conservative.)
- For example, we assume
  - both branches of a conditional are reachable
  - and that loops always terminate.

Safety of analysis

Naïvely,

```
if (false) {
  int z = x * y;
}
```

this instruction is reachable,

```
while (true) {
  // Code without 'break'
}
int z = x * y;
```

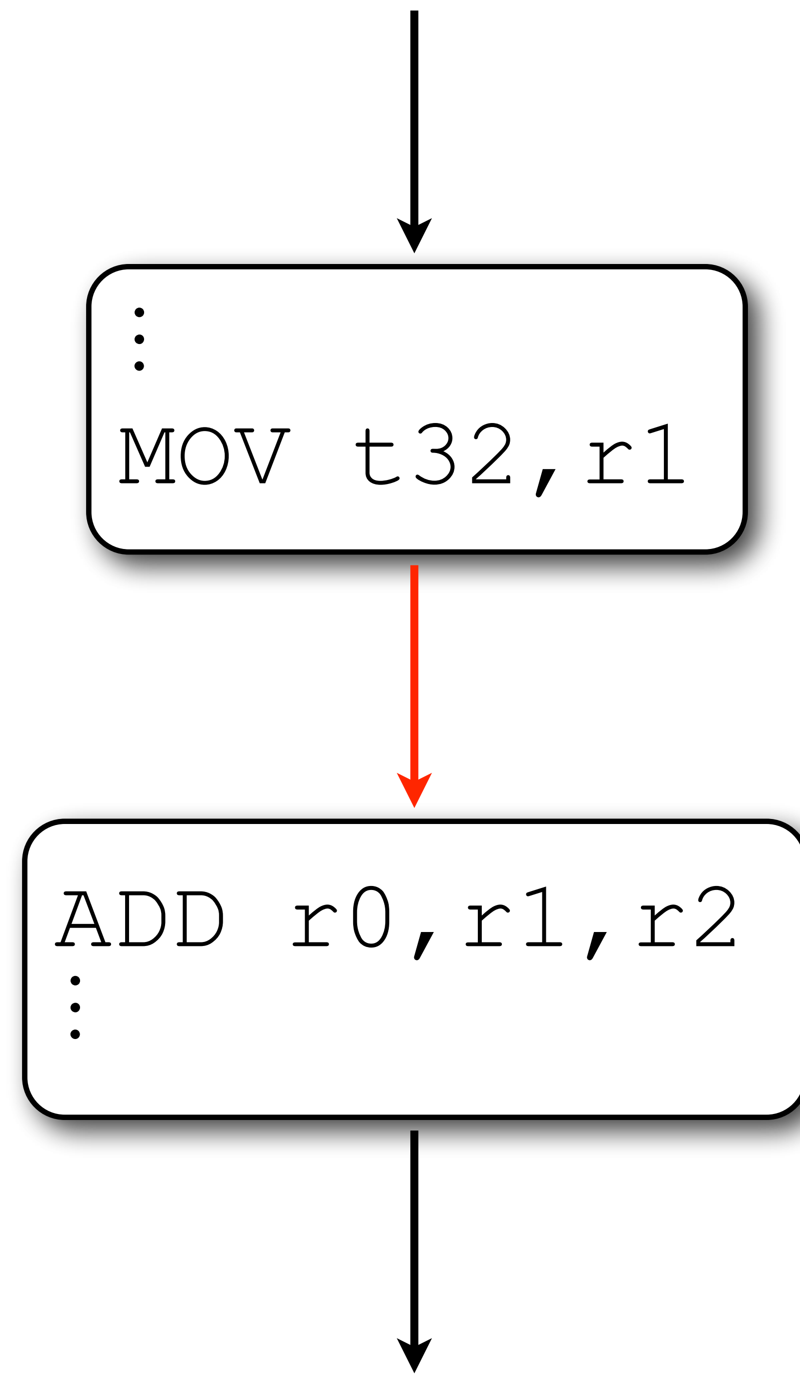
and so is this one.

Safety of analysis

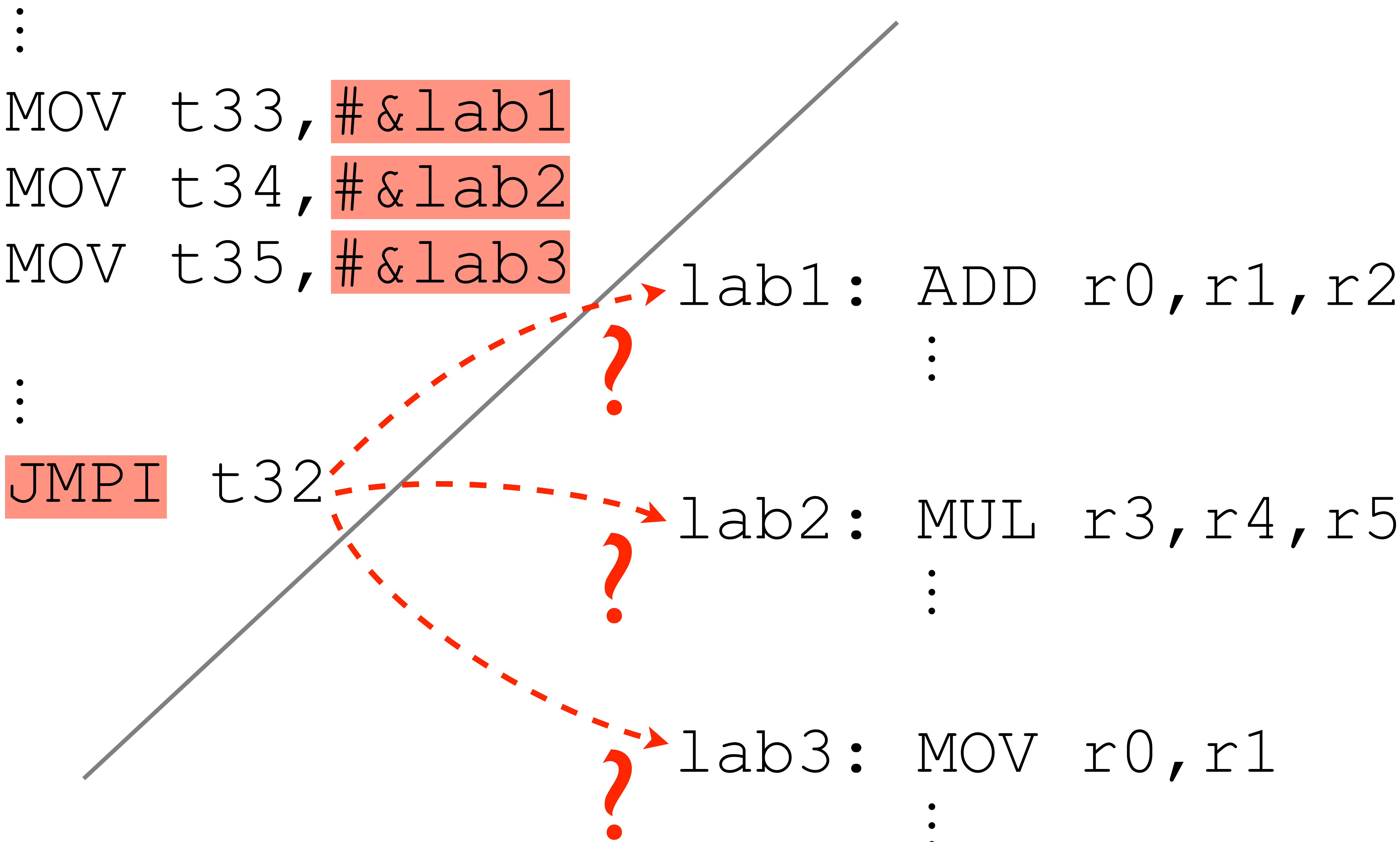
Another source of uncertainty is encountered when constructing the original flowgraph: the presence of indirect branches (also known as "computed jumps").

Safety of analysis

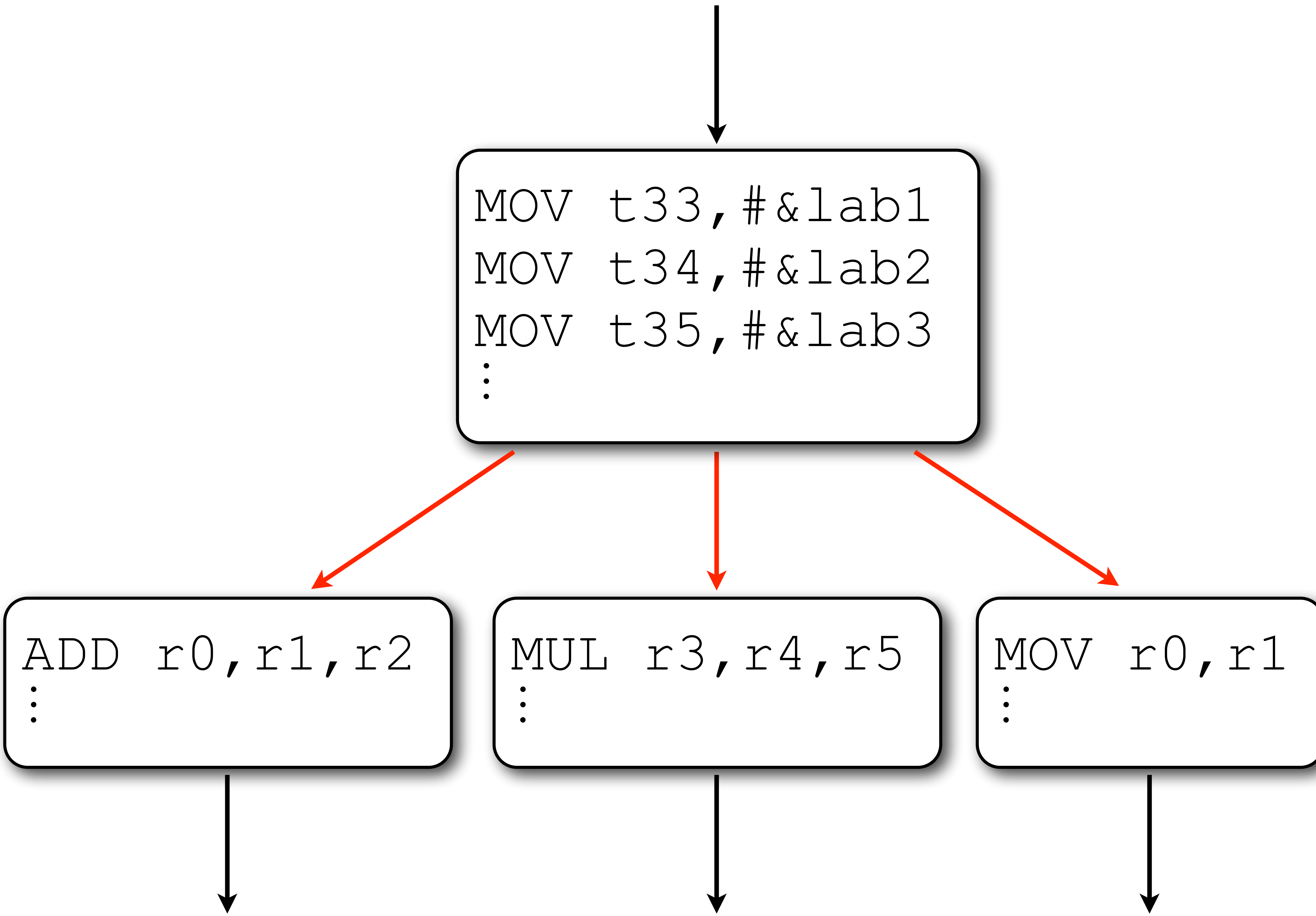
```
      ⋮
      MOV t32,r1
      JMP lab1

      ⋮
lab1: ADD r0,r1,r2
      ⋮
```



Safety of analysis



Safety of analysis

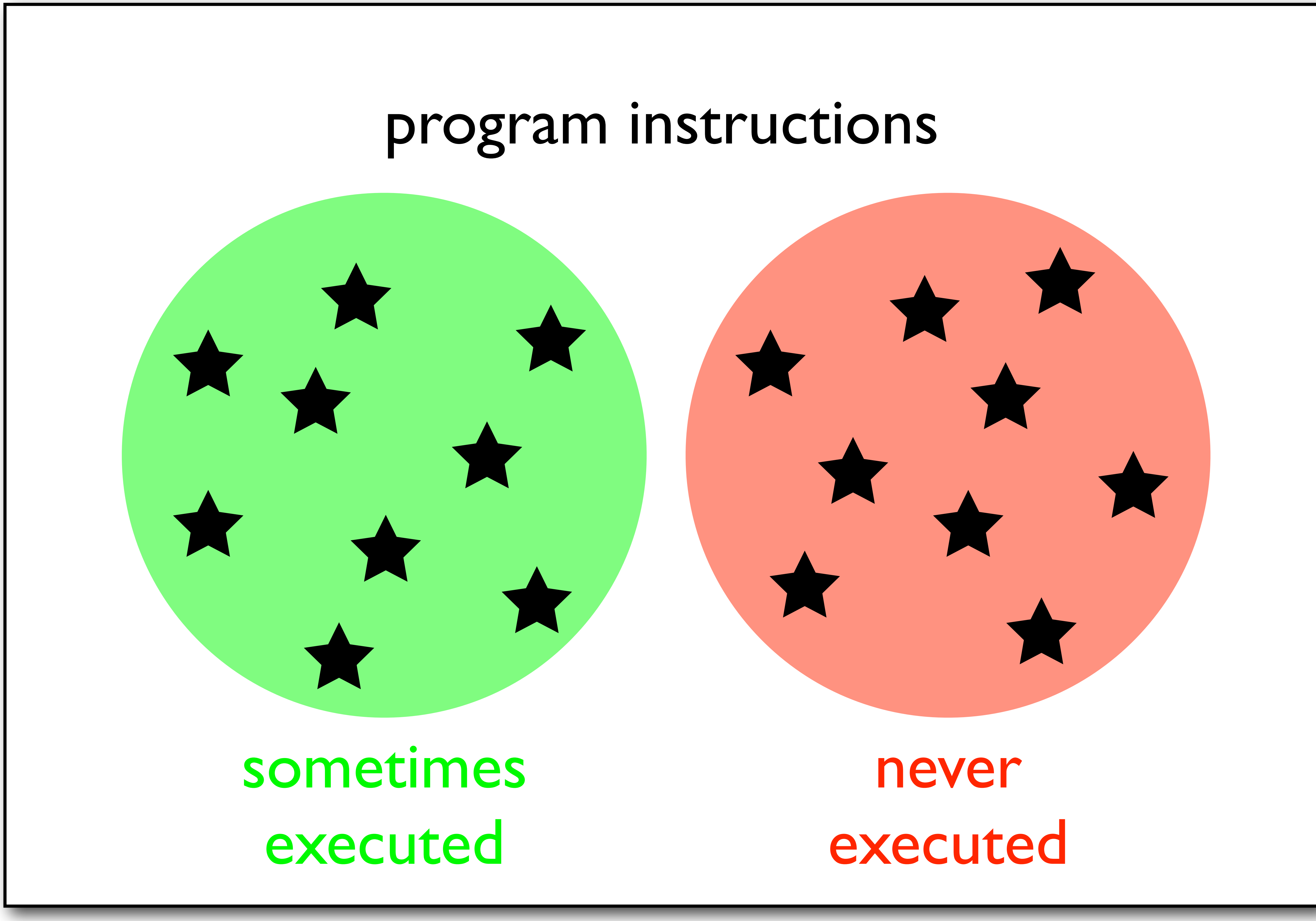


Safety of analysis

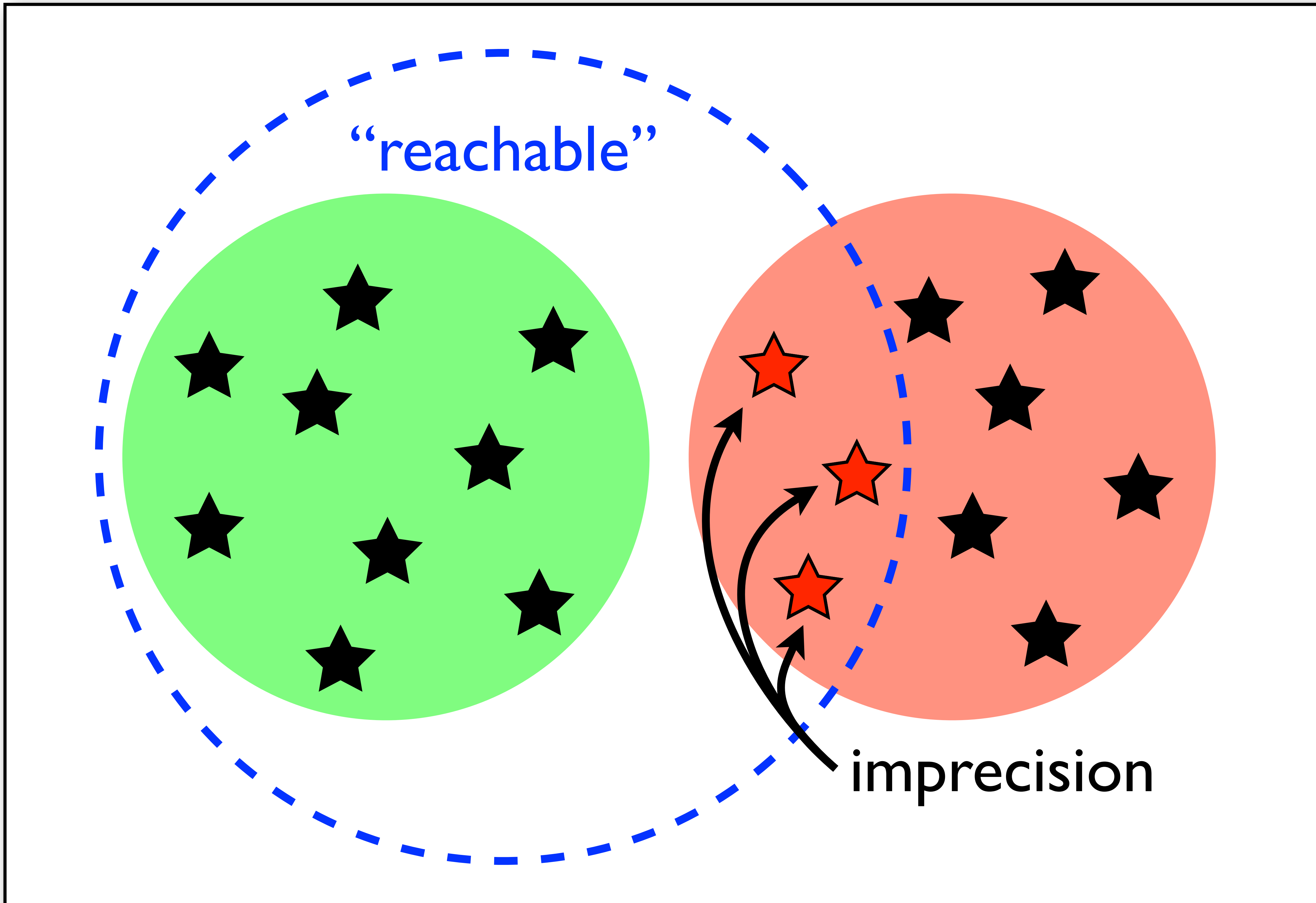
Again, this is a conservative *overestimation* of reachability.

In the worst-case scenario in which branch-address computations are completely unrestricted (i.e. the target of a jump could be absolutely anywhere), the presence of an indirect branch forces us to assume that *all* instructions are potentially reachable in order to guarantee safety.

Safety of analysis



Safety of analysis

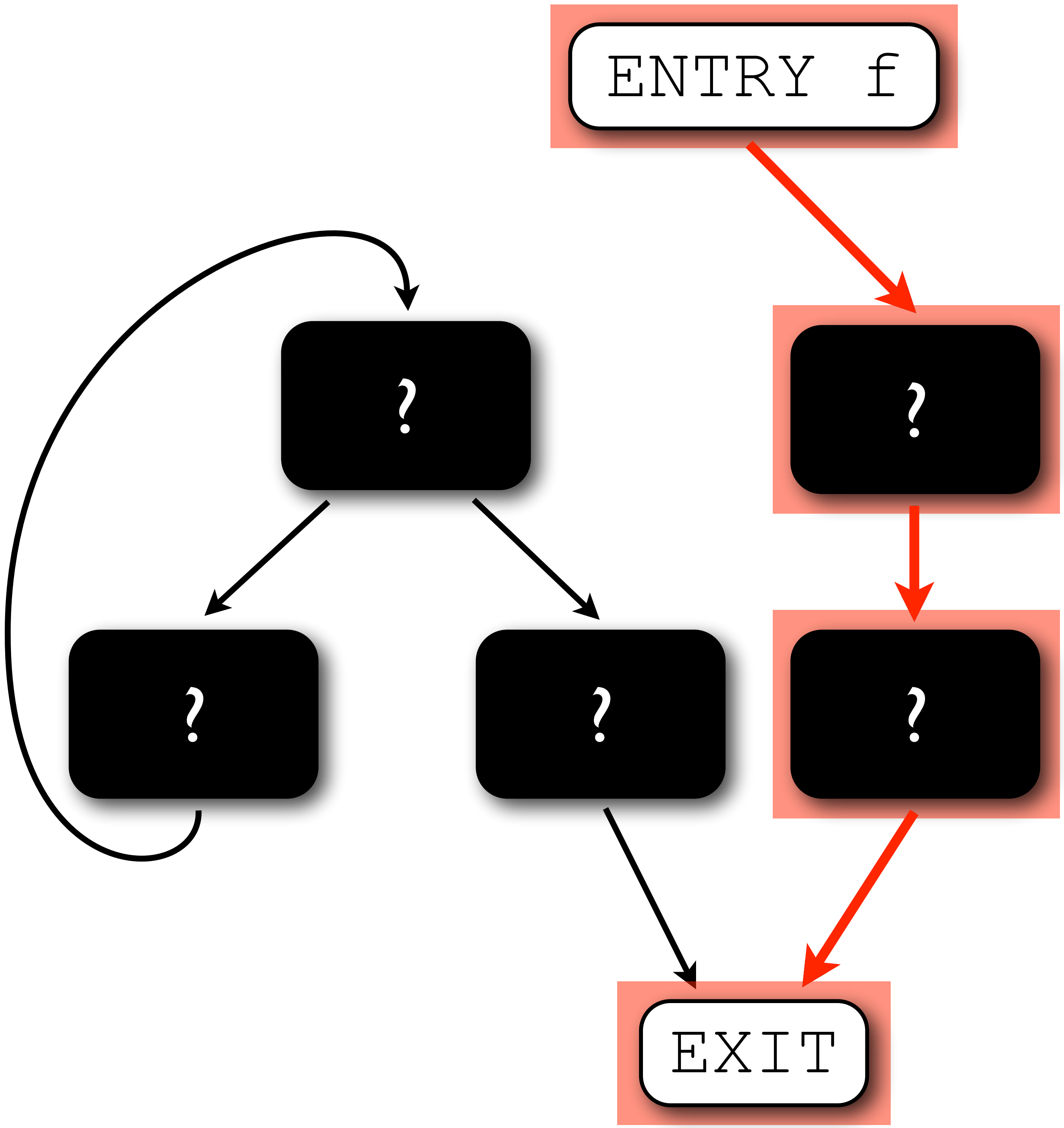


Unreachable code

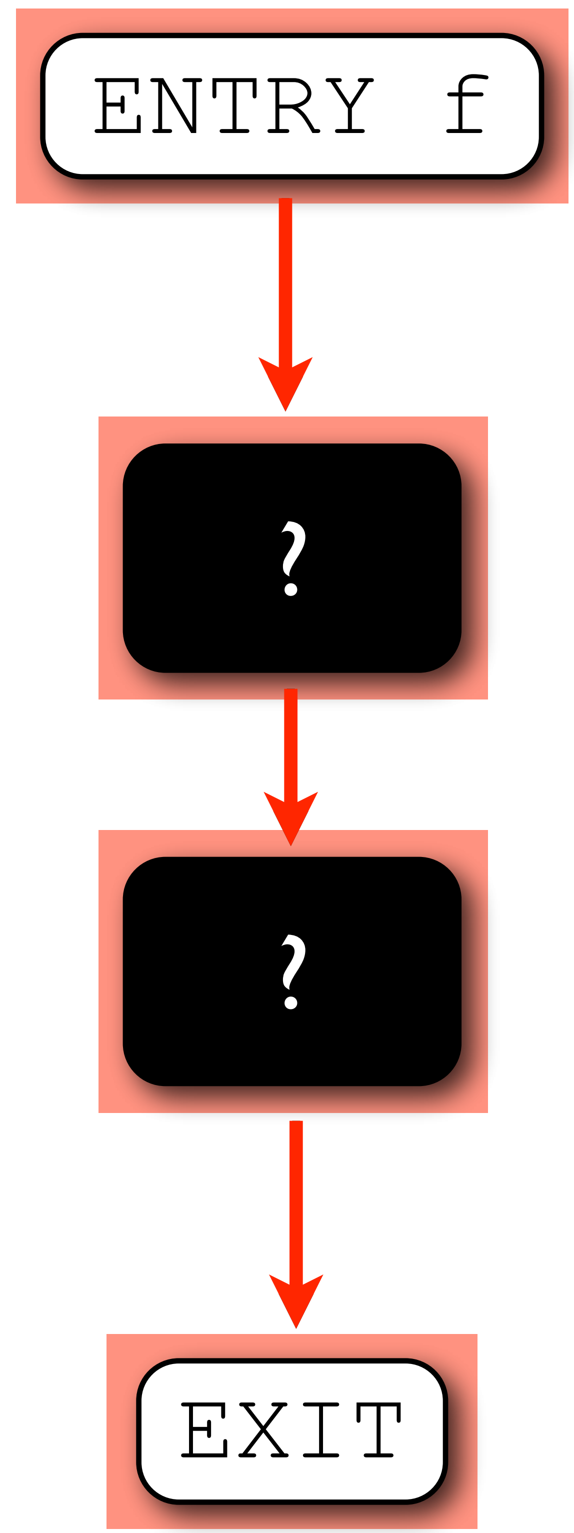
This naïve reachability analysis is simplistic, but has the advantage of corresponding to a very straightforward operation on the flowgraph of a procedure:

1. mark the procedure's entry node as reachable;
2. mark every successor of a marked node as reachable and repeat until no further marking is required.

Unreachable code



Unreachable code



Unreachable code

Programmers rarely write code which is completely unreachable in this naïve sense. Why bother with this analysis?

- Naïvely unreachable code may be introduced as a result of other optimising transformations.
- With a little more effort, we can do a better job.

Unreachable code

Obviously, if the conditional expression in an `if` statement is literally the constant "`false`", it's safe to assume that the statements within are unreachable.

```
if (false) {
    int z = x * y;
}
```

UNREACHABLE

But programmers never write code like that either.

Unreachable code

However, other optimisations might produce such code.
For example, *copy propagation*:

```
bool debug = false;
⋮
if (debug) {
   int z = x * y;
}
```

Unreachable code

However, other optimisations might produce such code.
For example, *copy propagation*:

```
⋮
if (false) {
  int z = x * y;
}
```

UNREACHABLE

Unreachable code

We can try to spot (slightly) more subtle things too.

- `if (!true) {... }`
- `if (false && ...) {... }`
- `if (x != x) {... }`
- `while (true) {... } ...`
- ...

Unreachable code

Note, however, that the reachability analysis no longer consists simply of checking whether any paths to an instruction *exist* in the flowgraph, but whether any of the paths to an instruction are actually *executable*.

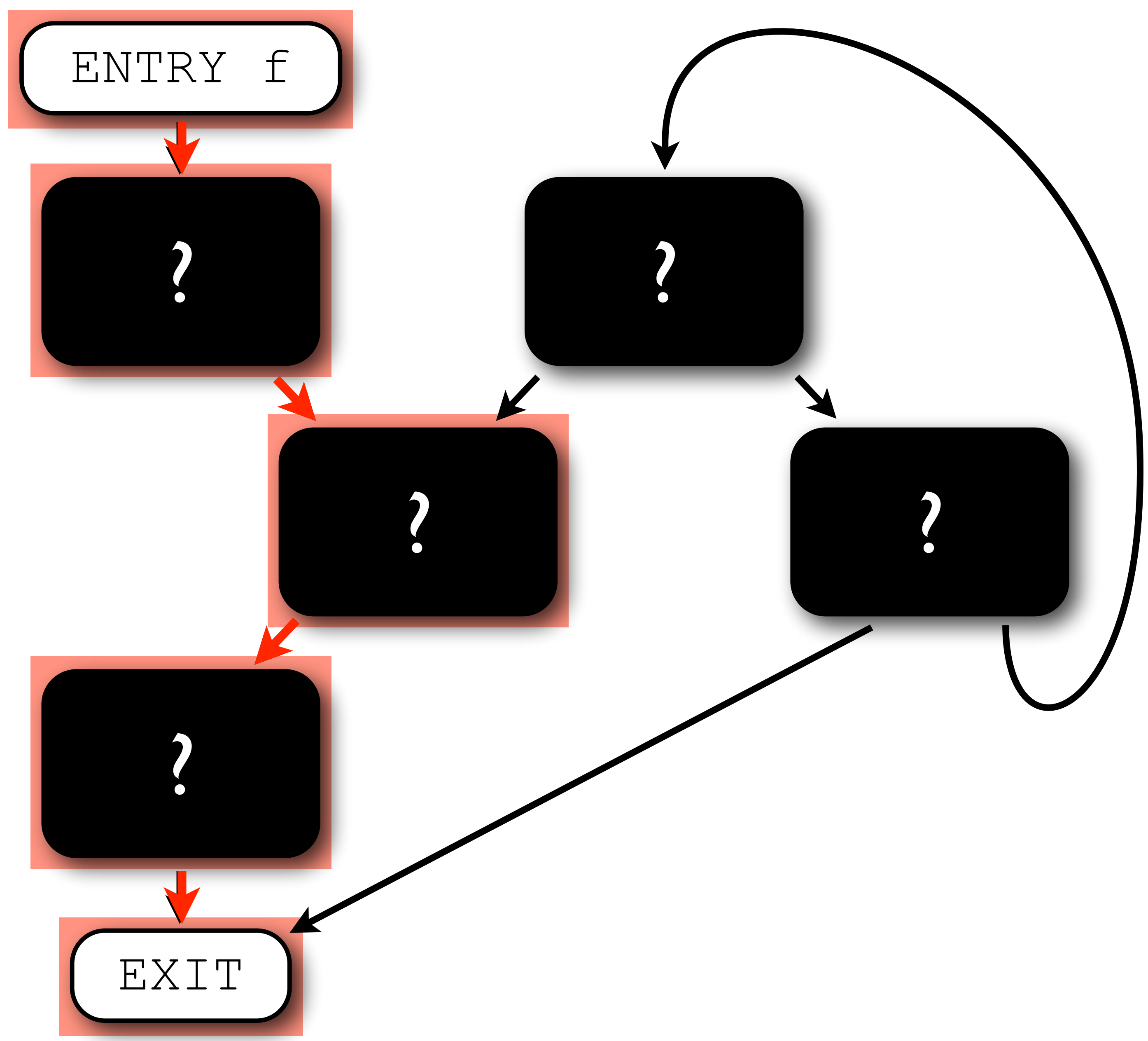
With more effort we may get arbitrarily clever at spotting non-executable paths in particular cases, but in general the undecidability of arithmetic means that we cannot always spot them all.

Unreachable code

Although unreachable-code elimination can only make a program *smaller*, it may enable other optimisations which make the program *faster*.

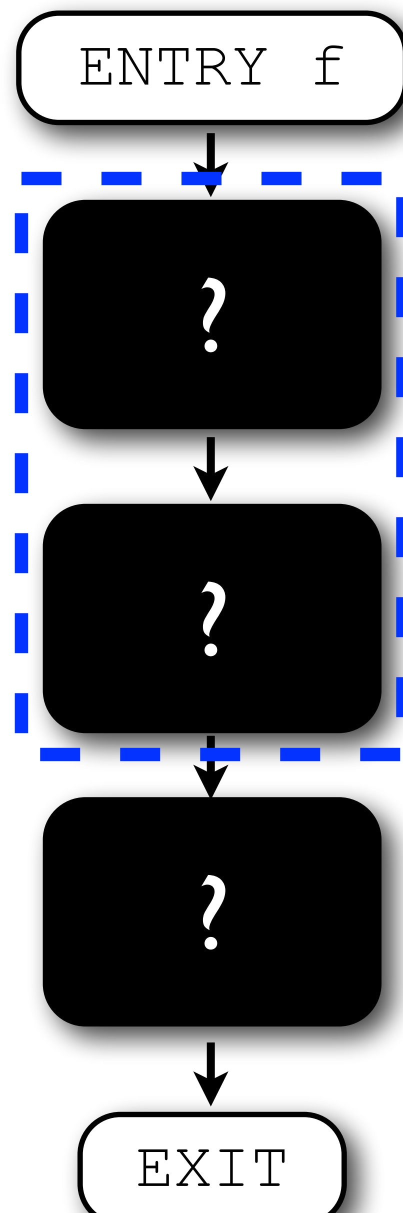
Unreachable code

For example, *straightening* is an optimisation which can eliminate jumps between basic blocks by coalescing them:



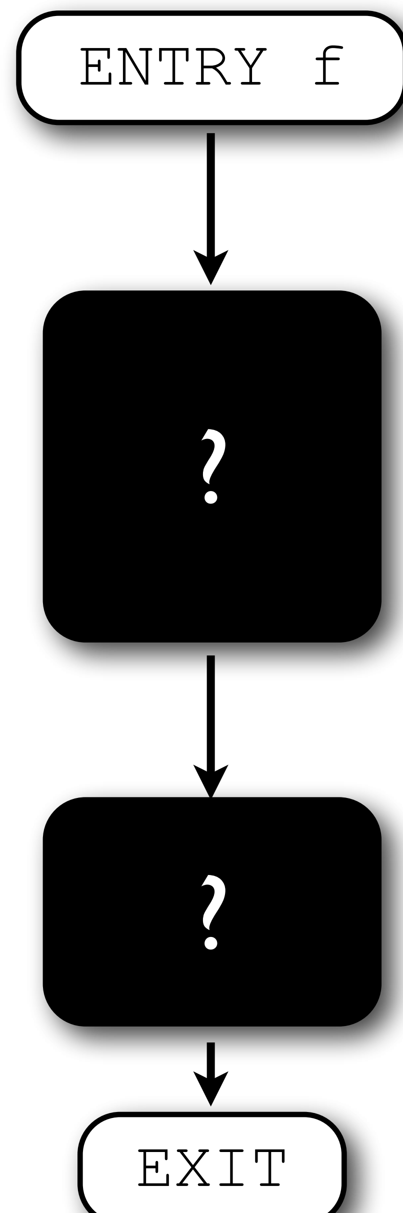
Unreachable code

For example, *straightening* is an optimisation which can eliminate jumps between basic blocks by coalescing them:



Unreachable code

For example, *straightening* is an optimisation which can eliminate jumps between basic blocks by coalescing them:



Straightening has removed a branch instruction, so the new program will execute faster.

Inter-procedural analysis

An *inter-procedural* analysis collects information about an entire program.

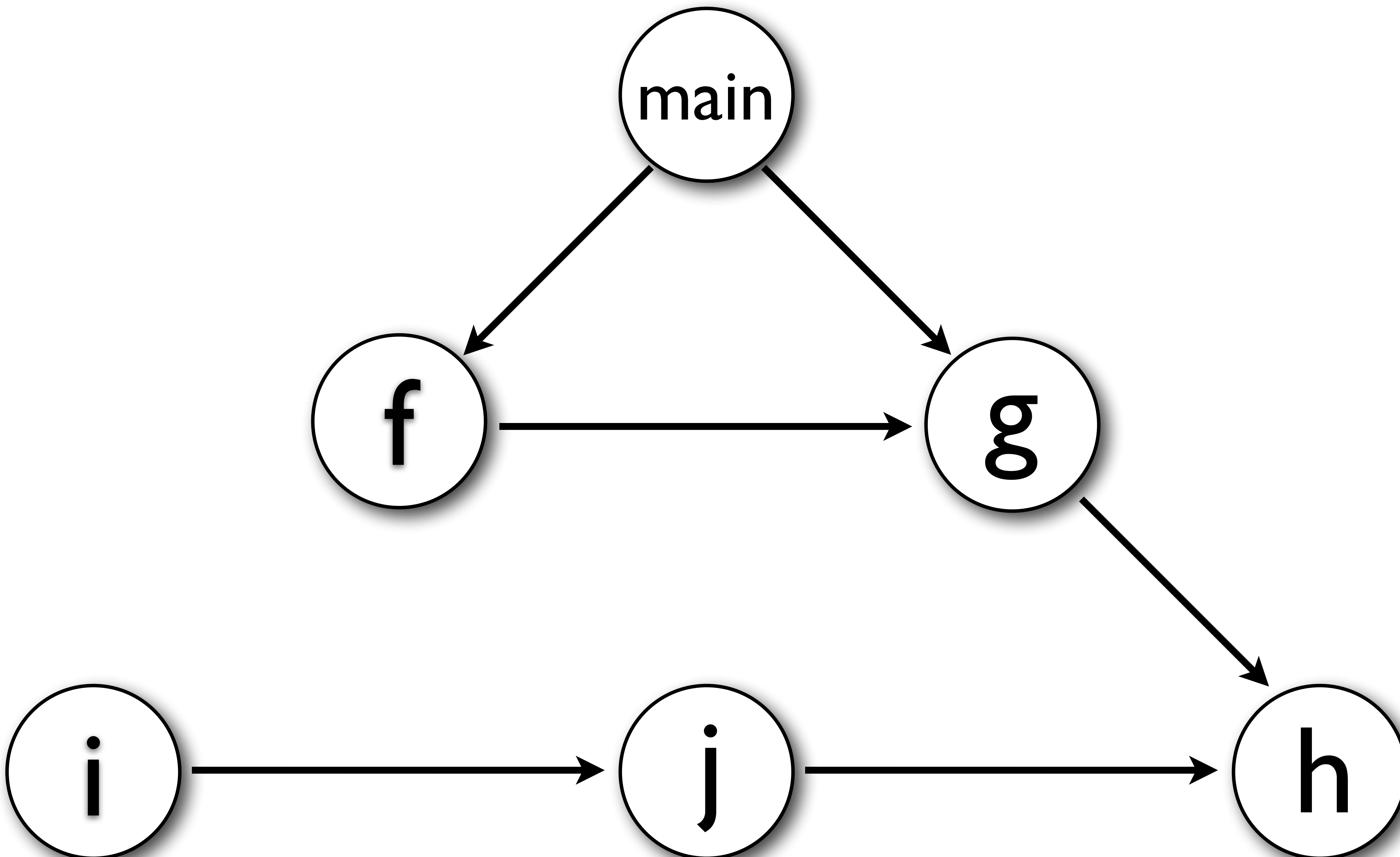
Information is collected from the instructions of each procedure and then propagated between procedures.

One example of an inter-procedural control-flow optimisation (an analysis and an accompanying transformation) is *unreachable-procedure elimination*.

Unreachable procedures

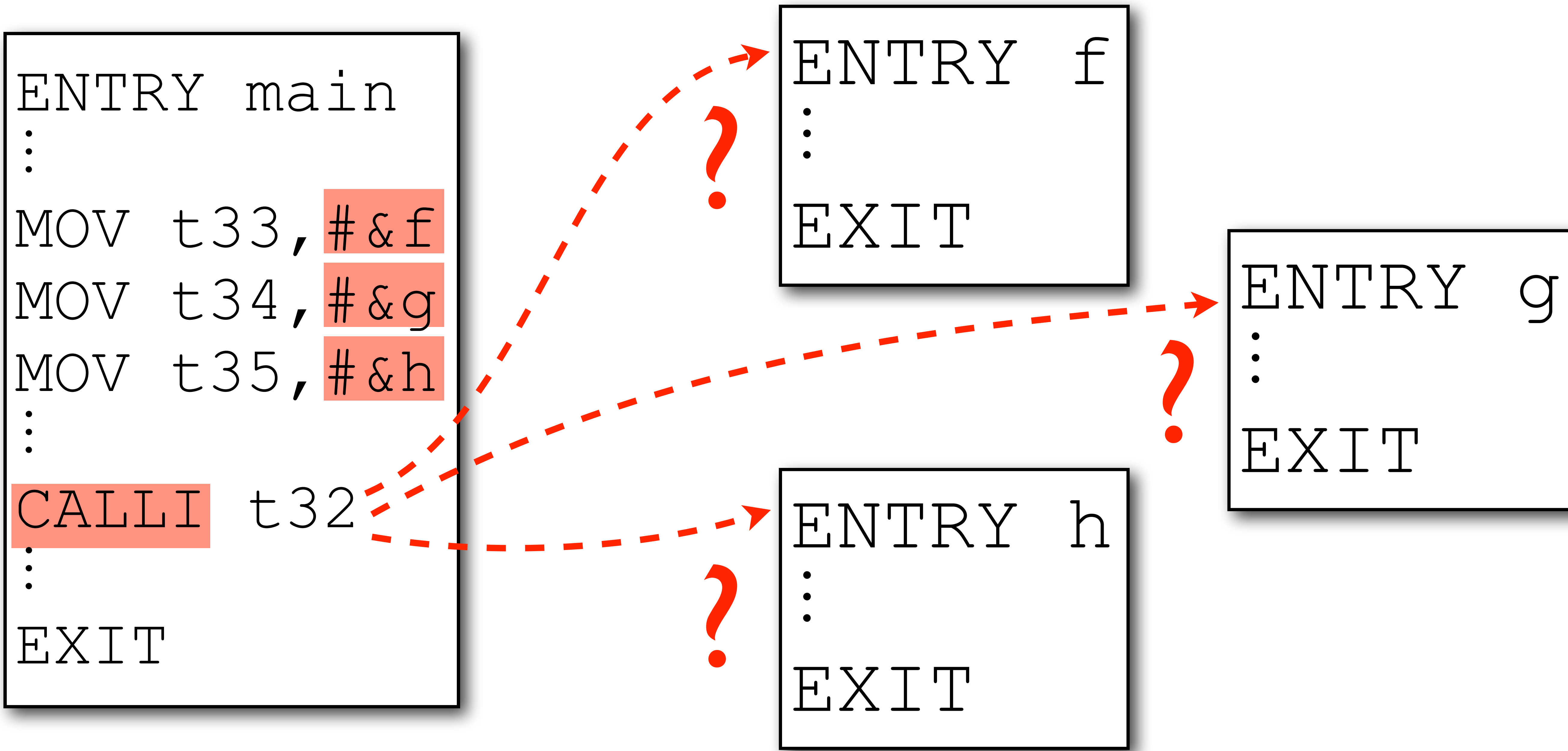
Unreachable-*procedure* elimination is very similar in spirit to unreachable-*code* elimination, but relies on a different data structure known as a *call graph*.

Call graphs



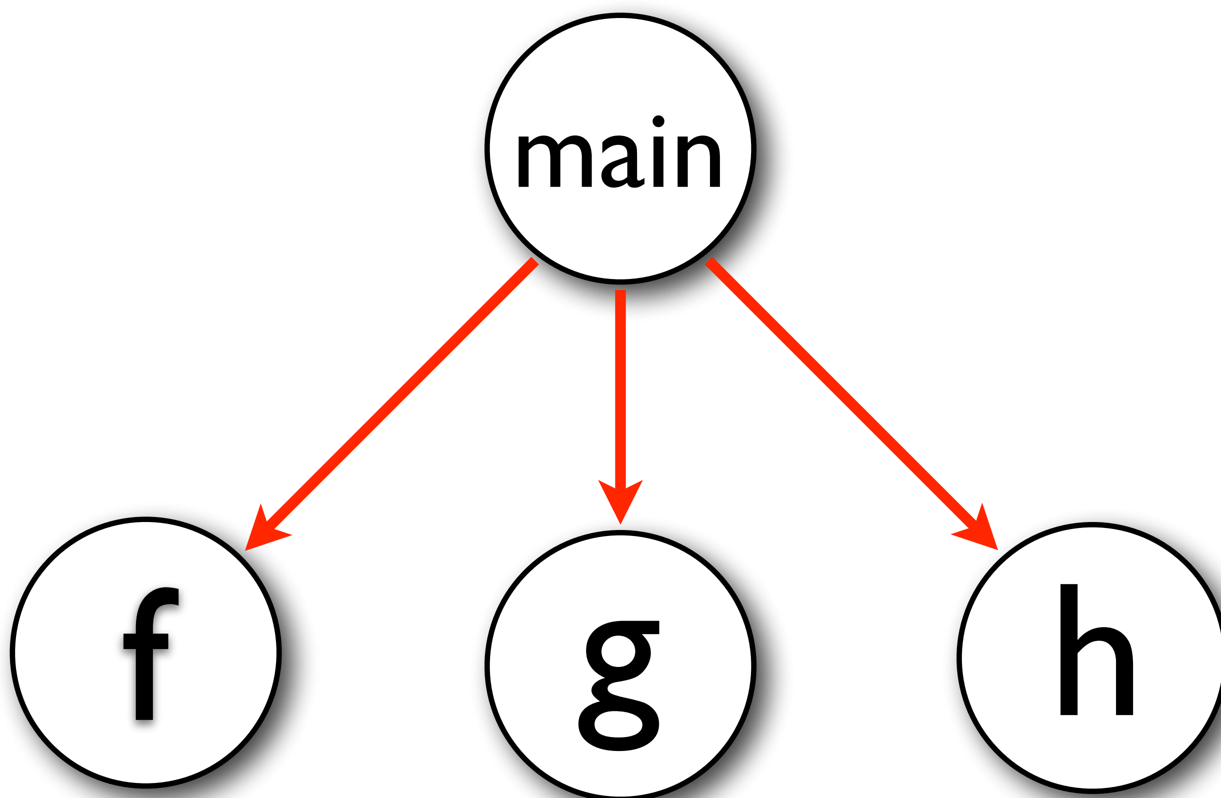
Call graphs

Again, the precision of the graph is compromised in the presence of *indirect calls*.



Call graphs

Again, the precision of the graph is compromised in the presence of *indirect calls*.



And as before, this is a *safe* overestimation of reachability.

Call graphs

In general, we assume that a procedure containing an indirect call has *all* address-taken procedures as successors in the call graph — i.e., it could call any of them.

This is obviously *safe*; it is also obviously *imprecise*.

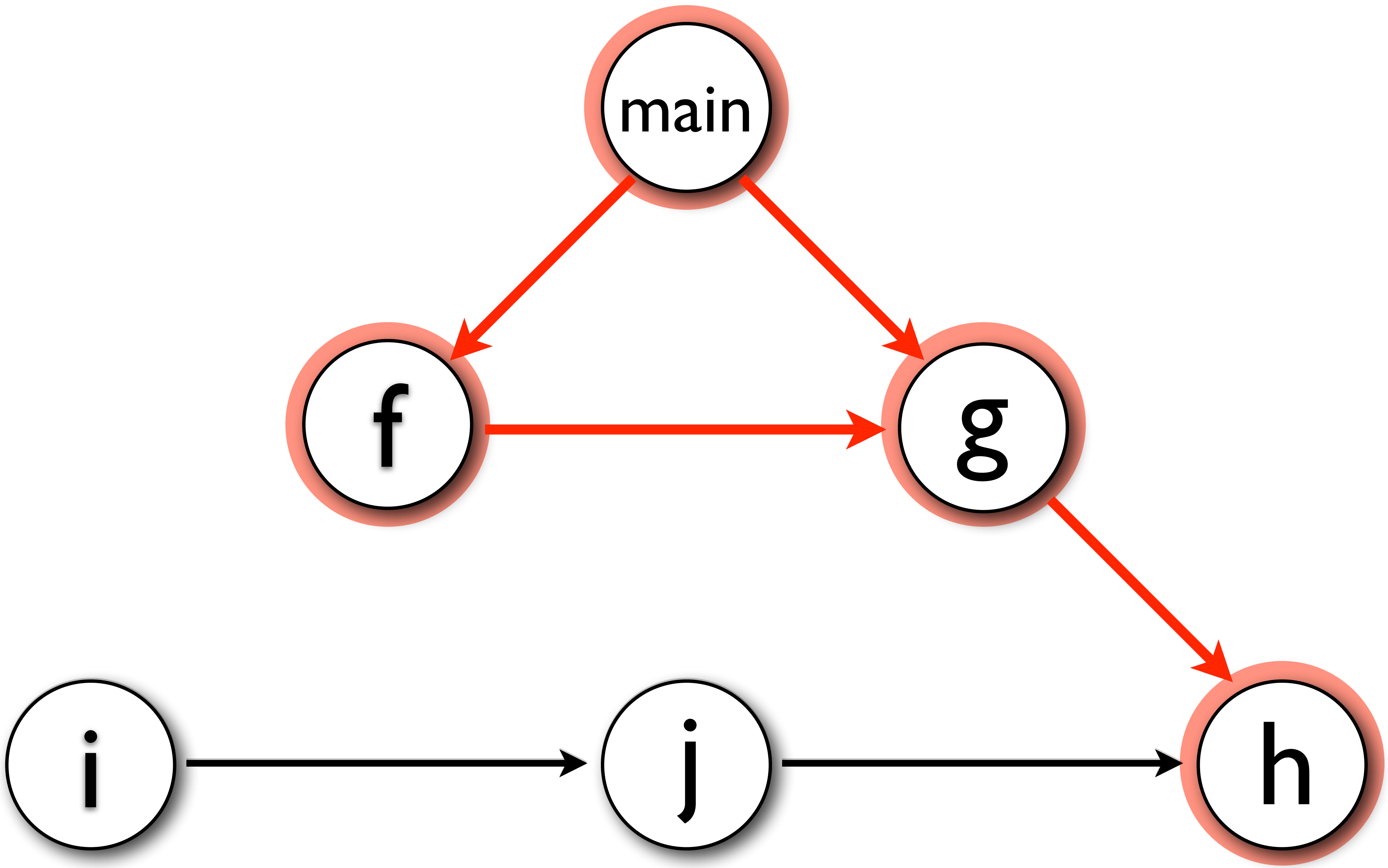
As before, it might be possible to do better by application of more careful methods (e.g. tracking data-flow of procedure variables).

Unreachable procedures

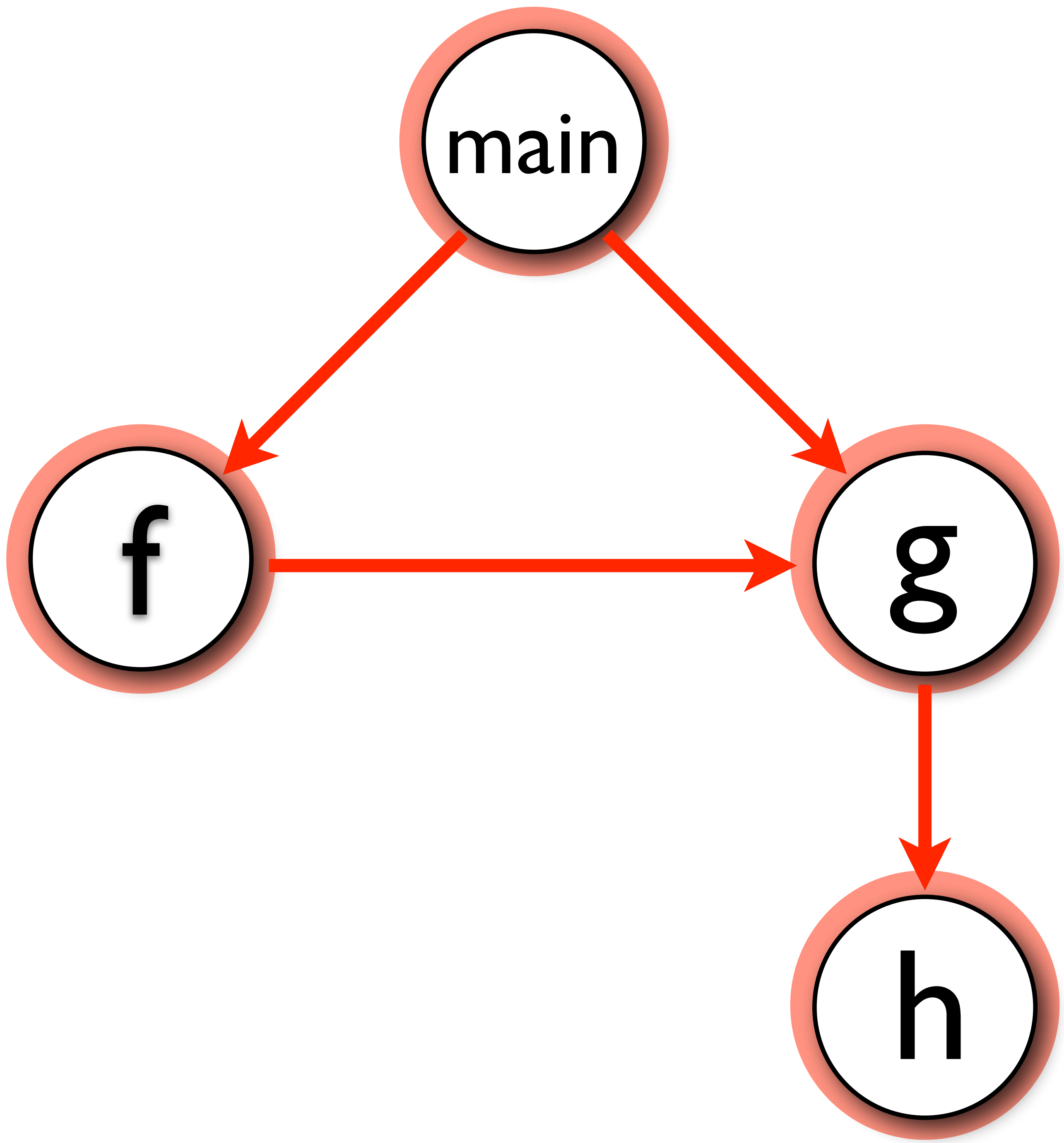
The reachability analysis is virtually identical to that used in unreachable-*code* elimination, but this time operates on the call graph of the entire program (vs. the flowgraph of a single procedure):

1. mark procedure `main` as callable;
2. mark every successor of a marked node as callable and repeat until no further marking is required.

Unreachable procedures



Unreachable procedures



Safety of transformations

- All instructions/procedures to which control may flow at execution time will *definitely* be marked by the reachability analyses...
- ...but not vice versa, since some marked nodes might never be executed.
- Both transformations will *definitely* not delete any instructions/procedures which are needed to execute the program...
- ...but they might leave others alone too.

If simplification

- Let's look at another set of basic control-flow transformations that can be carried out with only small amounts of analysis
- In this case, *if simplification*, which alters the structure of if statements (or removes them altogether) when possible

If simplication

Empty `then` in `if-then`

```
if (f(x)) {
}
```

(Assuming that `f` has no side effects.)

If simplication

Empty `else` in `if-then-else`

```
if (f(x)) {
   z = x * y;
} else {
}
```

If simplication

Empty `then` in `if-then-else`

```
if (!f(x)) {
} else {
  z = x * y;
}
```

If simplication

Empty `then` and `else` in `if`-`then`-`else`

```
if (f(x)) {
} else {
}
```

If simplication

Constant condition

```
if (true) {
   z = x * y;
}
```

If simplication

Nested `if` with common subexpression

```
if (x > 3 && t) {
   ⋮
   if (x > 3) {
      z = x * y;
   } else {
      z = y - x;
   }
}
```

Loop simplification

```
int x = 0;
int i = 0;
while (i < 4) {
  i = i + 1;
  x = x + i;
}
```

Loop simplification

```
int x = 10;
int i = 4;
```

Summary

- Control-flow analysis operates on the control structure of a program (flowgraphs and call graphs)
- Unreachable-*code* elimination is an *intra*-procedural optimisation which reduces code size
- Unreachable-*procedur*e elimination is a similar, *inter*-procedural optimisation making use of the program's call graph
- Analyses for both optimisations must be imprecise in order to guarantee safety