Null Dereference Verification Via Over-approximated Weakest Precondition analysis

Ravichandhran Madhavan
Microsoft Research, India

Joint work with

Raghavan Komondoor,
Indian Institute of Science

Problem Definition

- Verify absence of Null dereferences
  - Demand-driven
  - Analyze a dereference in almost real-time
  - Sound (i.e, no false negatives)
  - No programmer annotations
  - Reasonably precise
  - Should work on real-world Java programs

Weakest (atleast once) Precondtion

- *WP(p,C)*
  - Constraint on the initial state that ensures that *C* holds **whenever** control reaches *p*
  - p may never be reached when *WP(p,C)* holds

- $WP_1(p,C)$
  - Constraint on the initial state that ensures that *p* is reached **atleast once** in a state satisfying *C.*

- $WP(p,C) = \neg WP_1(p,\neg C)$

Null deref verification using WP_1

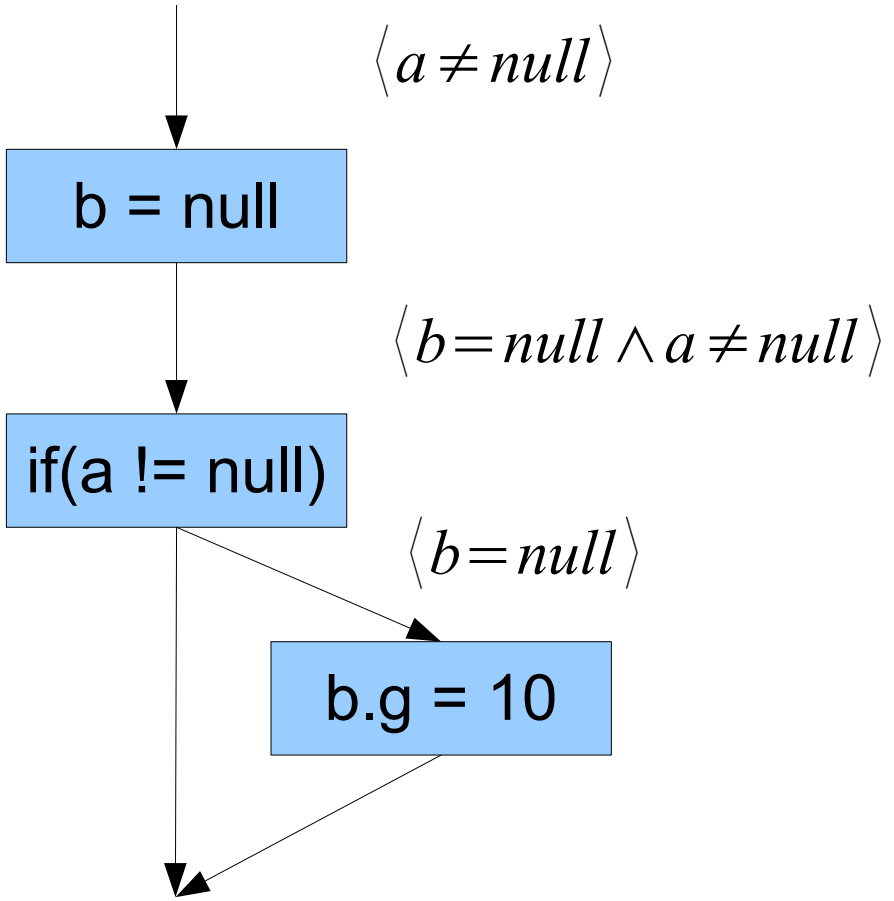
```
1: foo(a) {
2:     b = null;
3:     if (a != null)
4:         b.g = 10;
5: }
```

- We need to show that *WP(S4, b != null) = true*
- Equivalently, WP_1*(S4, b = null) = false*

Null deref verification using WP_1

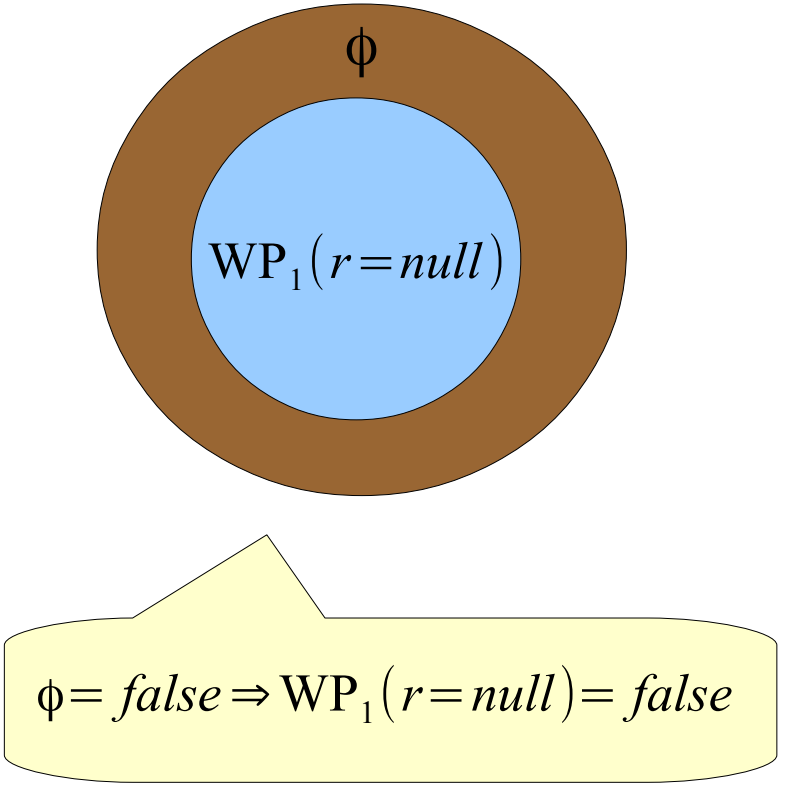
```
1: foo(a) {
2:     b = null;
3:     if (a != null)
4:         b.g = 10;
5: }
```

Dereference of b is not safe



Our approach

- Computing *WP* and WP_1 is undecidable
- **Our approach**: compute a condition that is weaker than WP_1



Design Goals

- Perform **strong updates** even in the presence of aliasing
- Incorporate **path-sensitivity:** track relevant branches
- Perform **context-sensitive inter-procedural** analysis

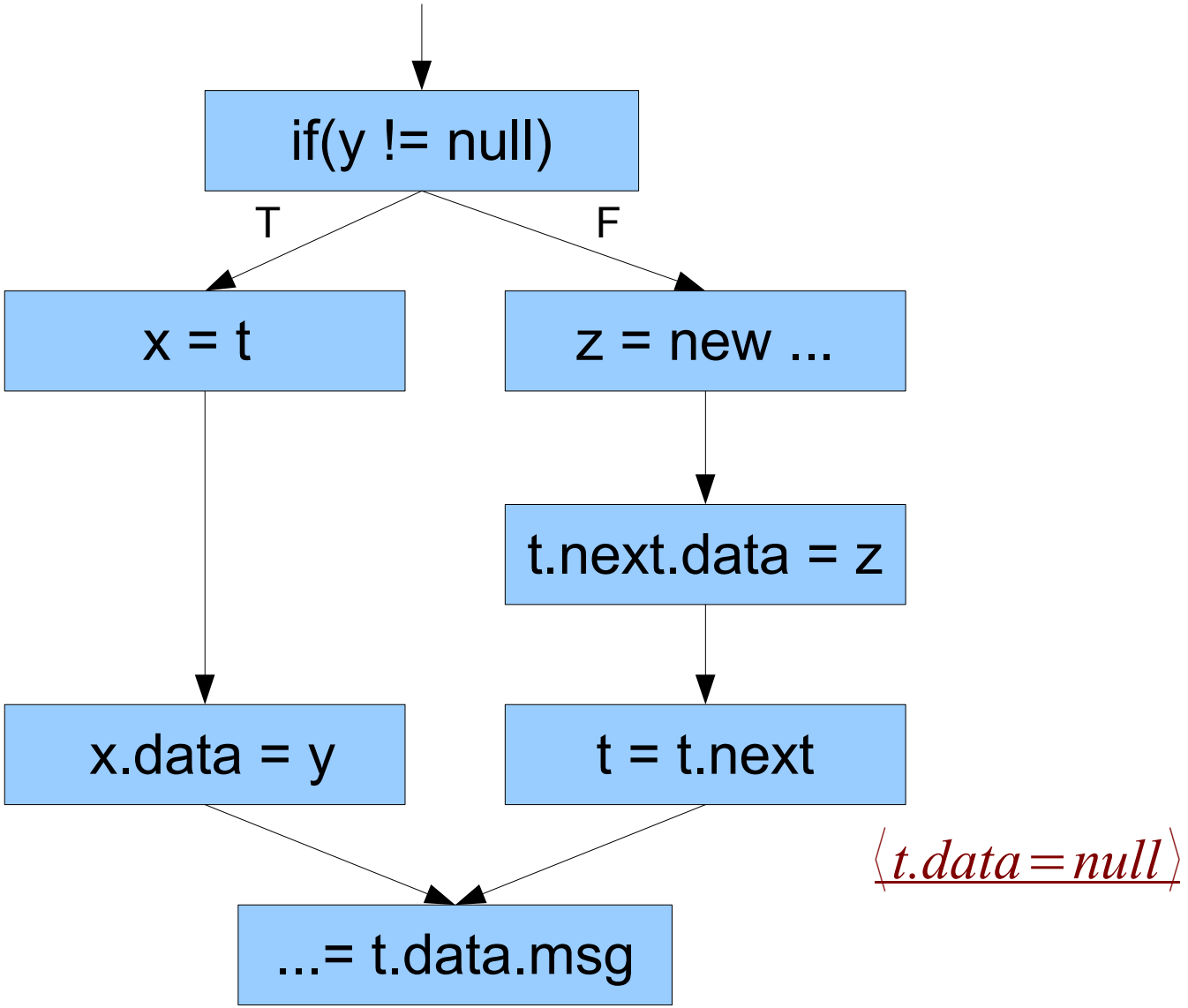
Abstract Domain

$$
\begin{array}{lcl}
AccessPath(AP) & \rightarrow & Variable.Fields \mid Variable \\
Fields & \rightarrow & field.Fields \mid \epsilon \\
Atom & \rightarrow & AP \mid null \\
Predicate & \rightarrow & Atom \;\; op \;\; Atom \mid true \mid false \\
op & \rightarrow & = \mid \neq \\
Disjunct & \rightarrow & 2^{Predicate} \\
Domain & \rightarrow & 2^{Disjunct}
\end{array}
$$

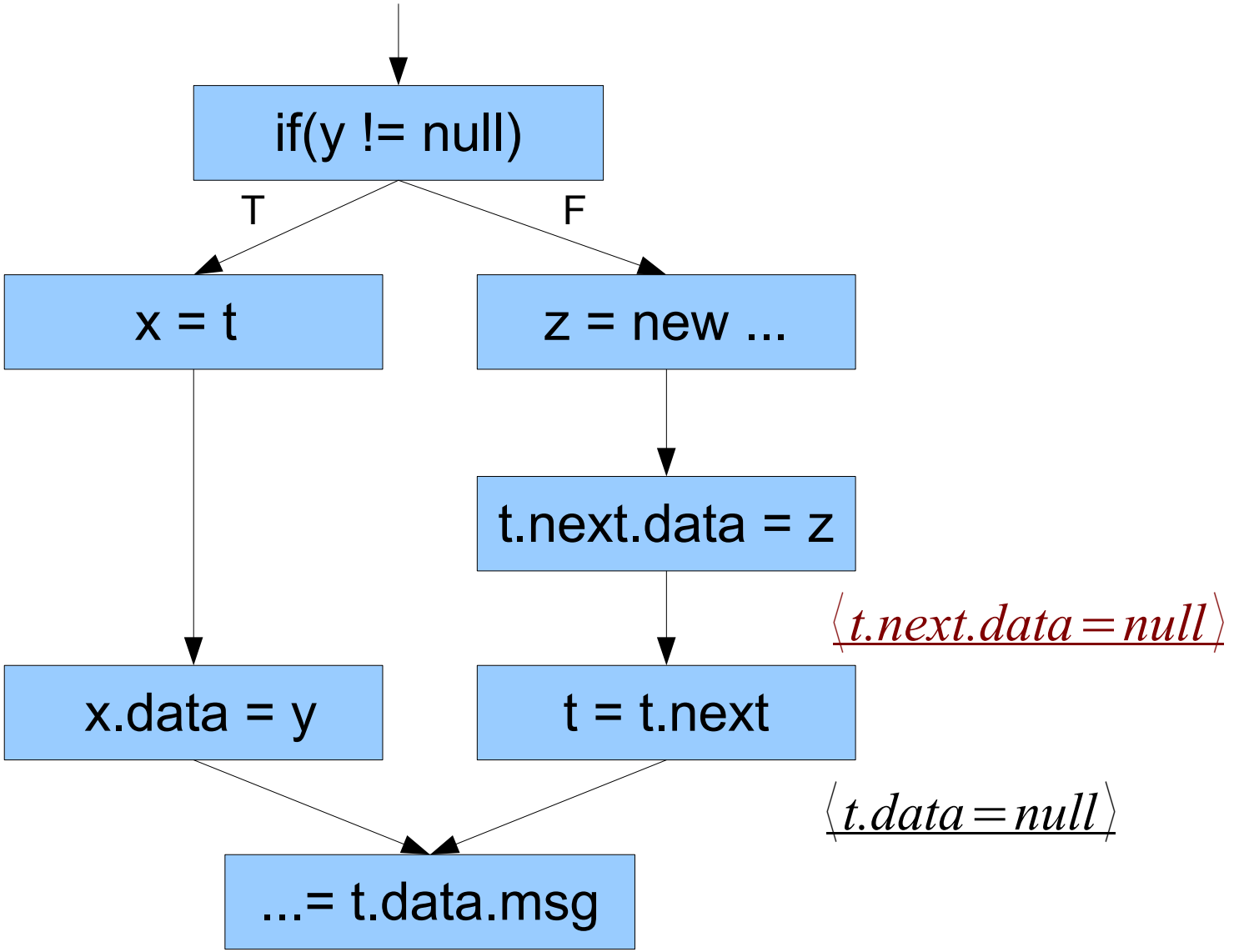
- The domain excludes access-paths in which a field repeats.
- Eg: $\langle a.f.g.h = null\,, b = null \rangle, \langle b.h = null \rangle \in Domain$

  $\langle a.f.g.h.f = null \rangle \notin Domain$

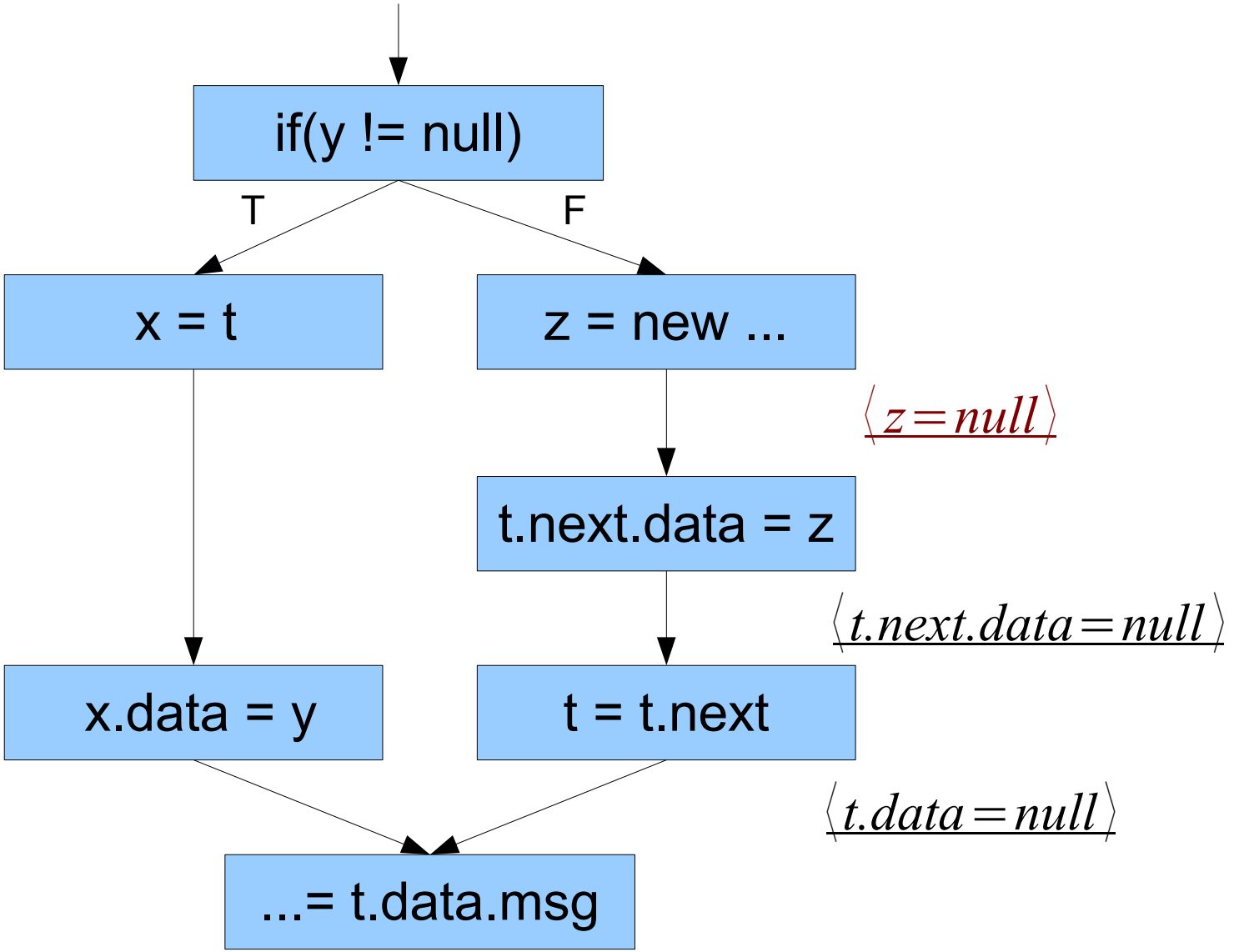
Illustration



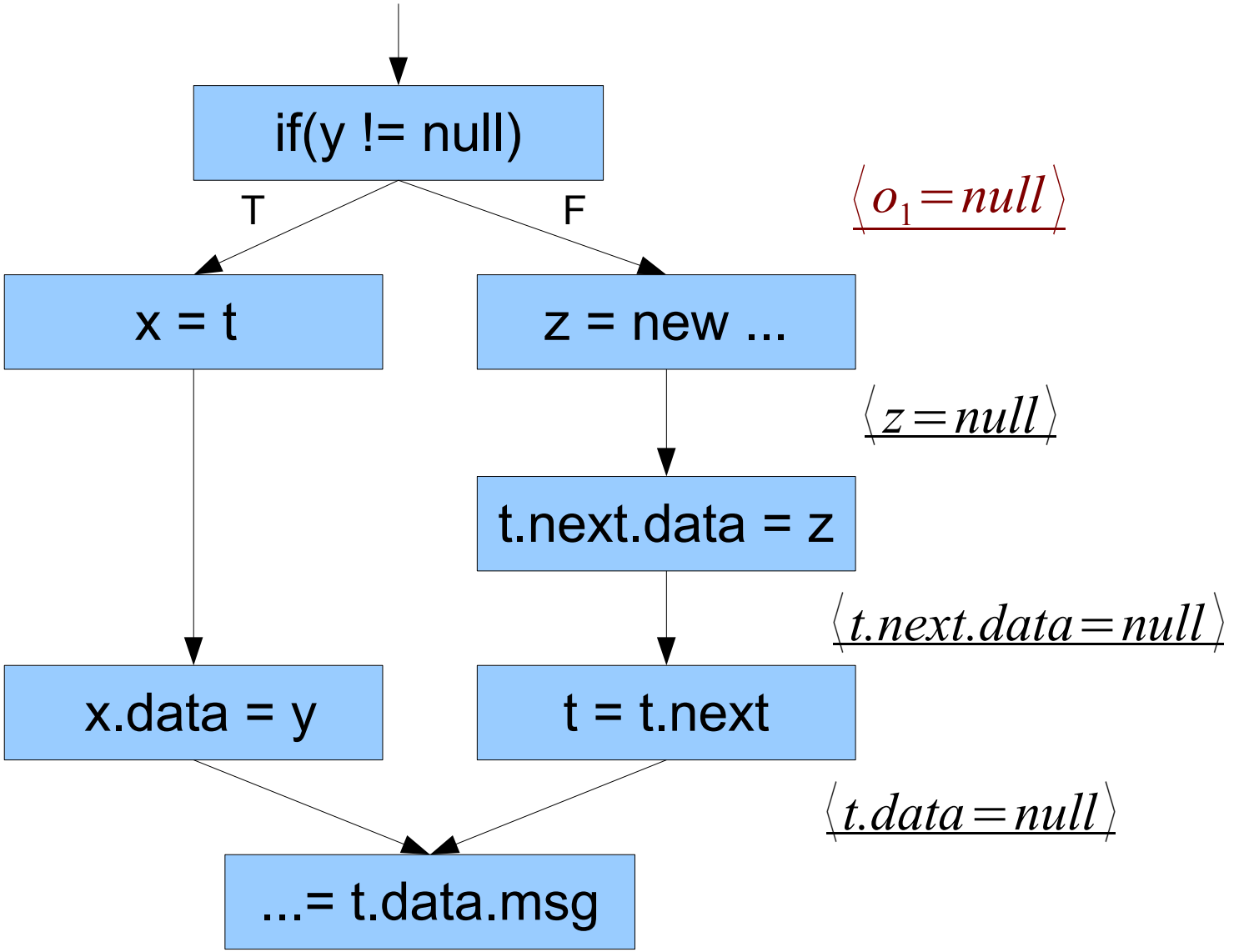
Illustration



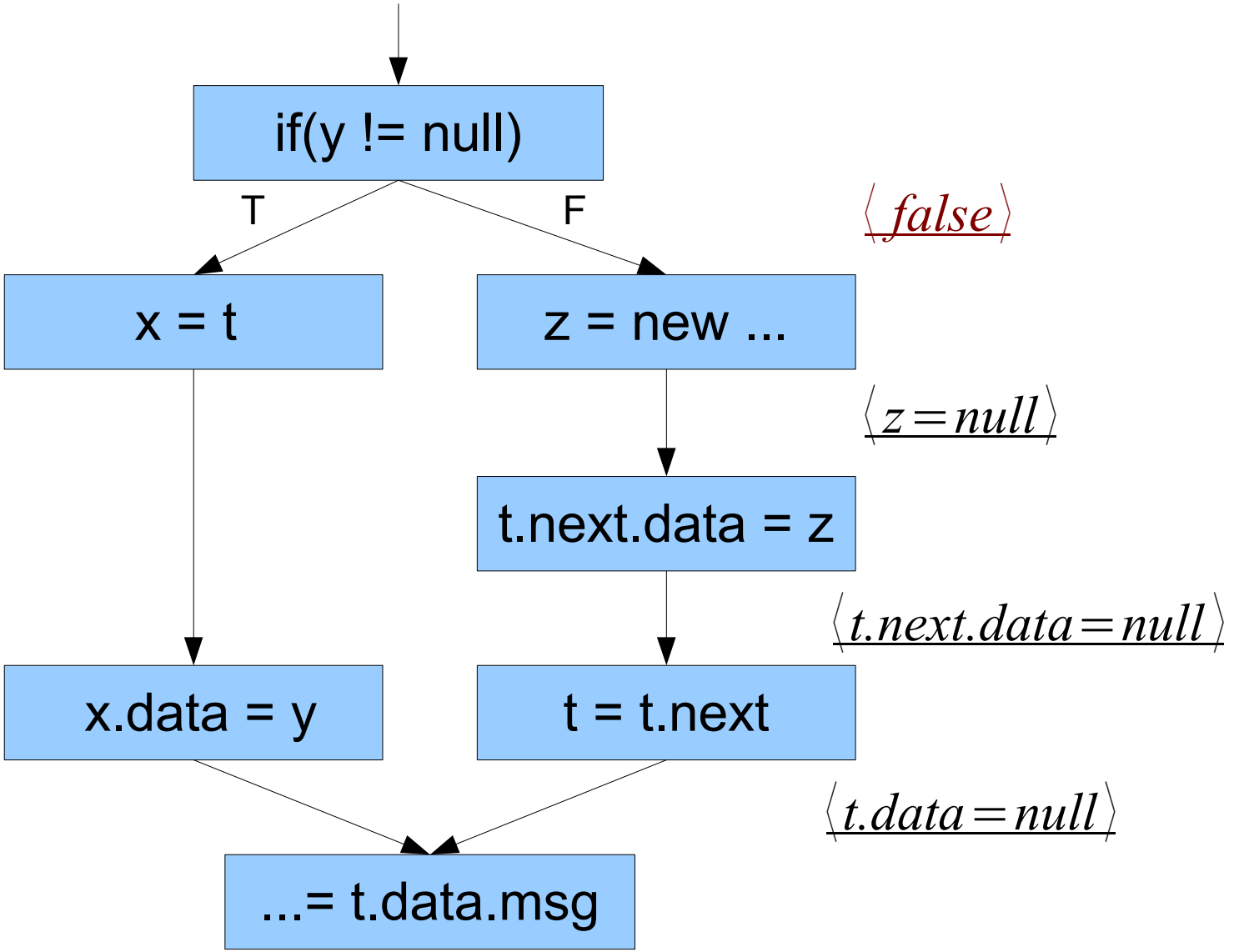
Illustration



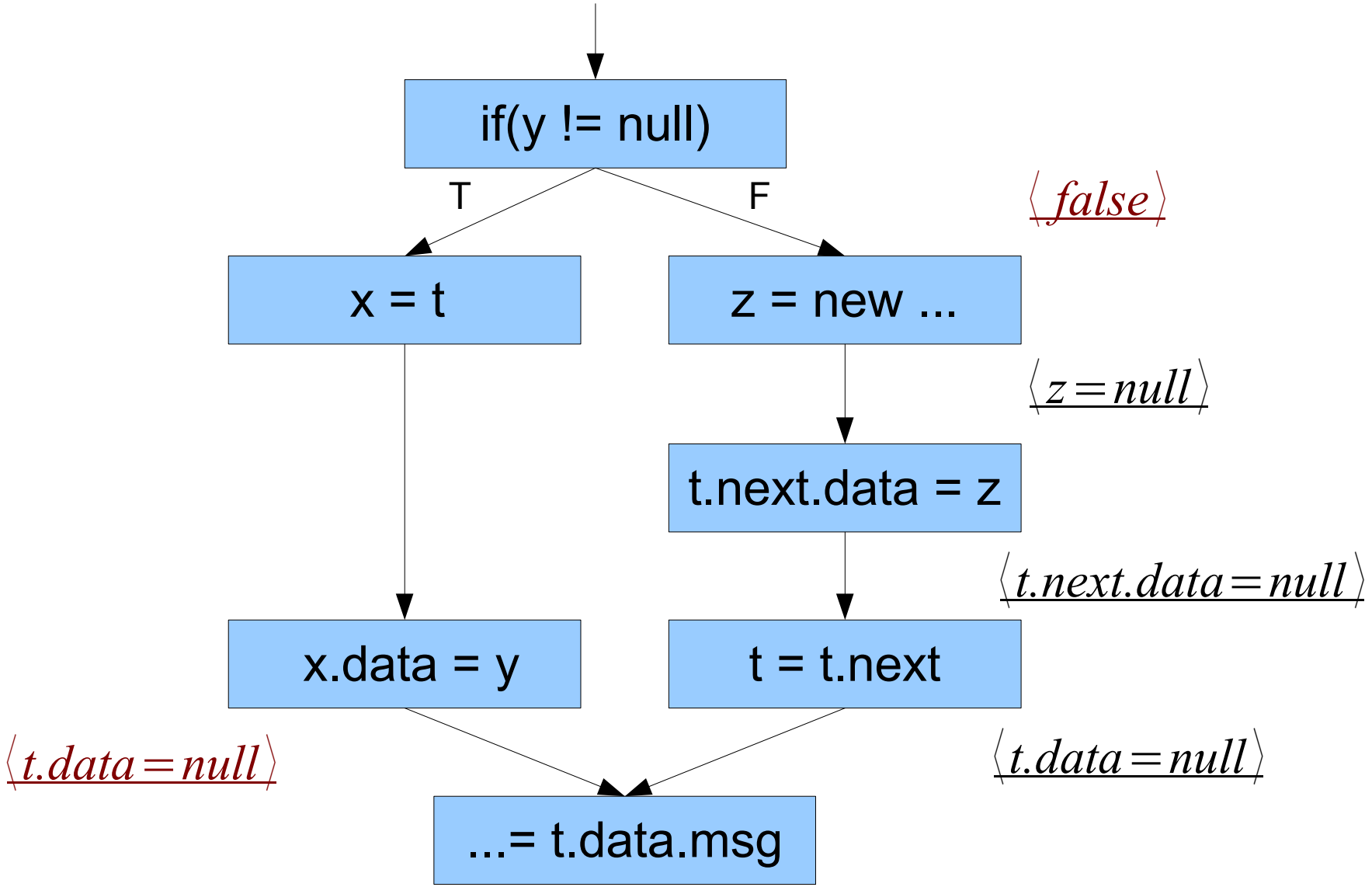
Illustration



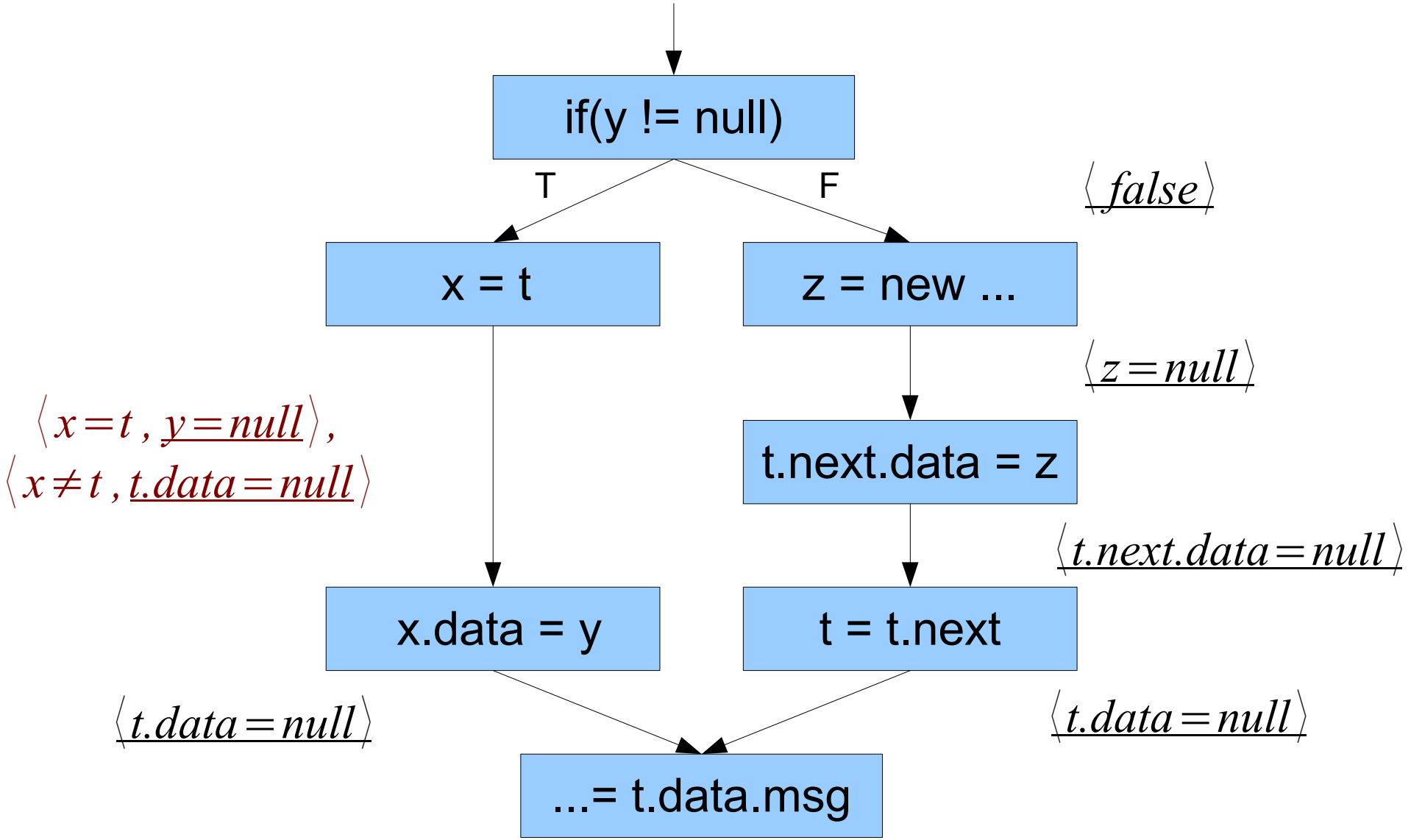
Illustration



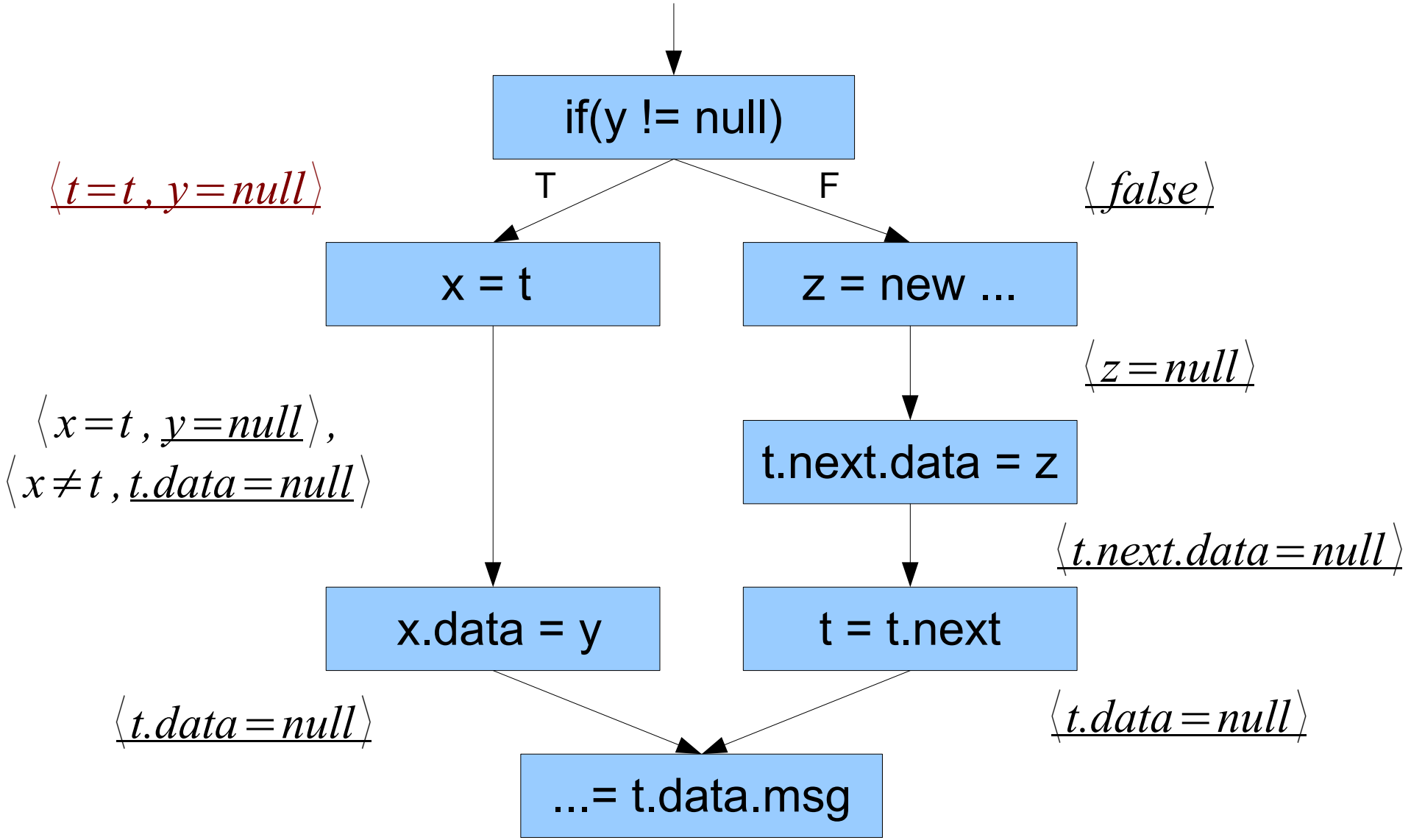
Illustration



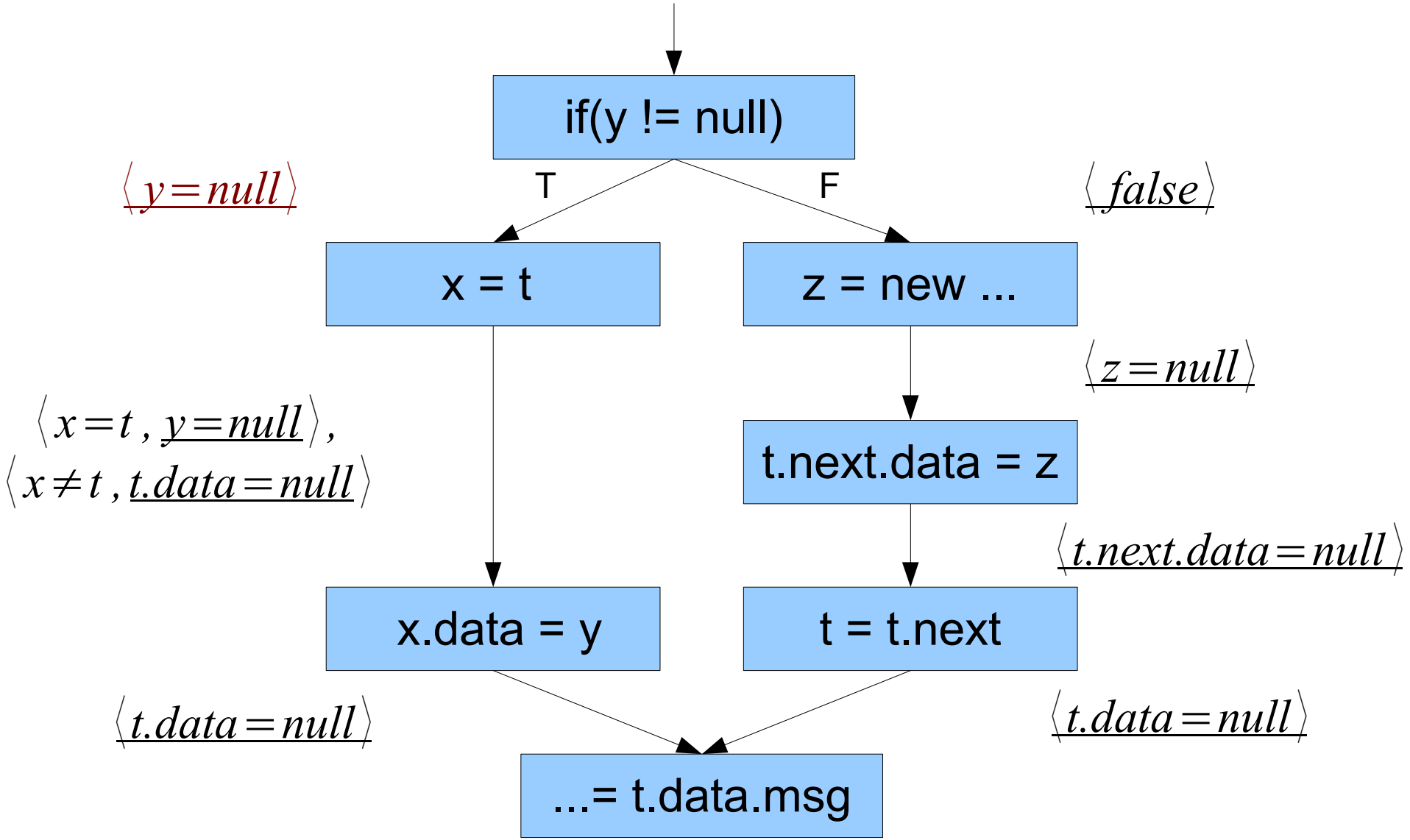
Illustration



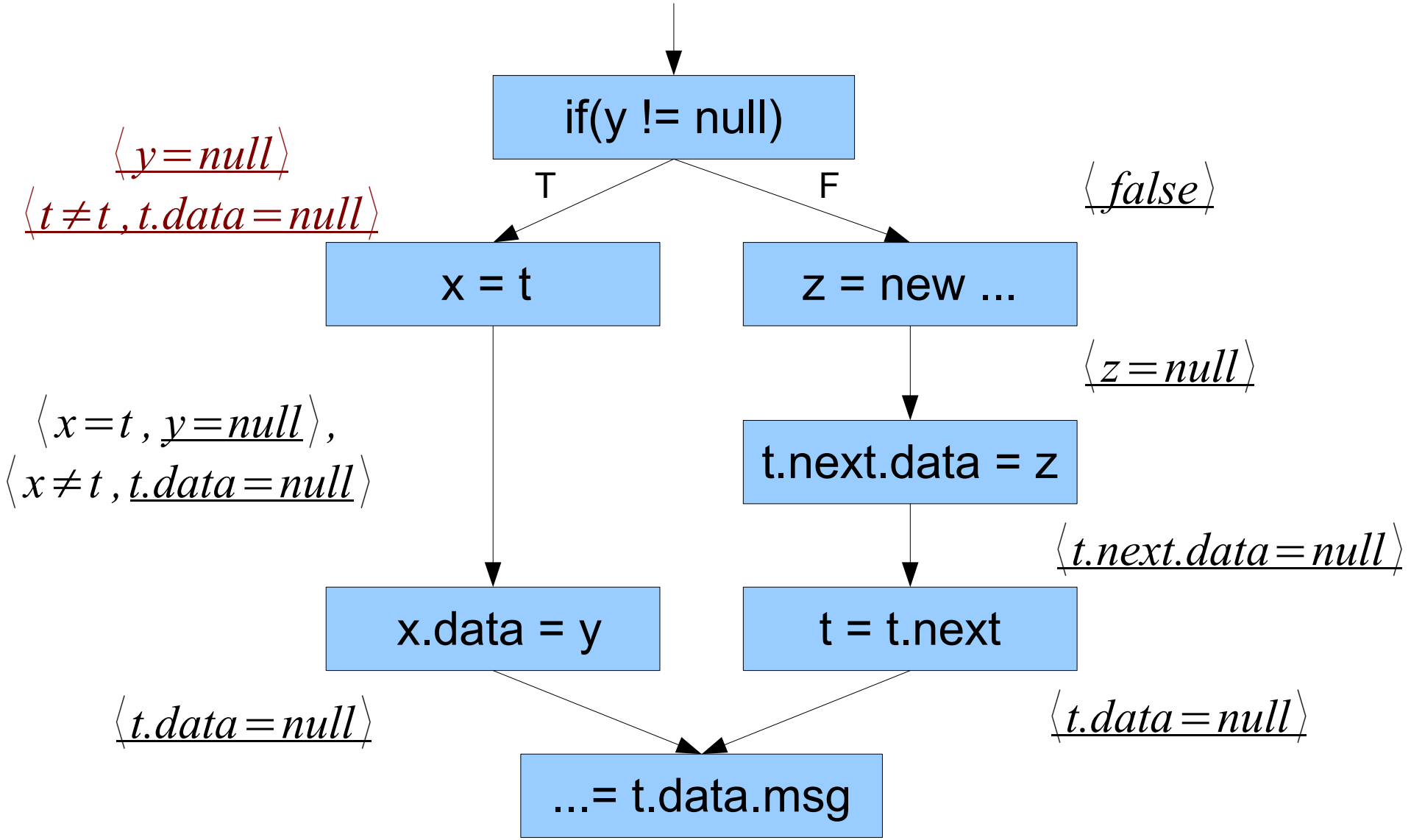
Illustration



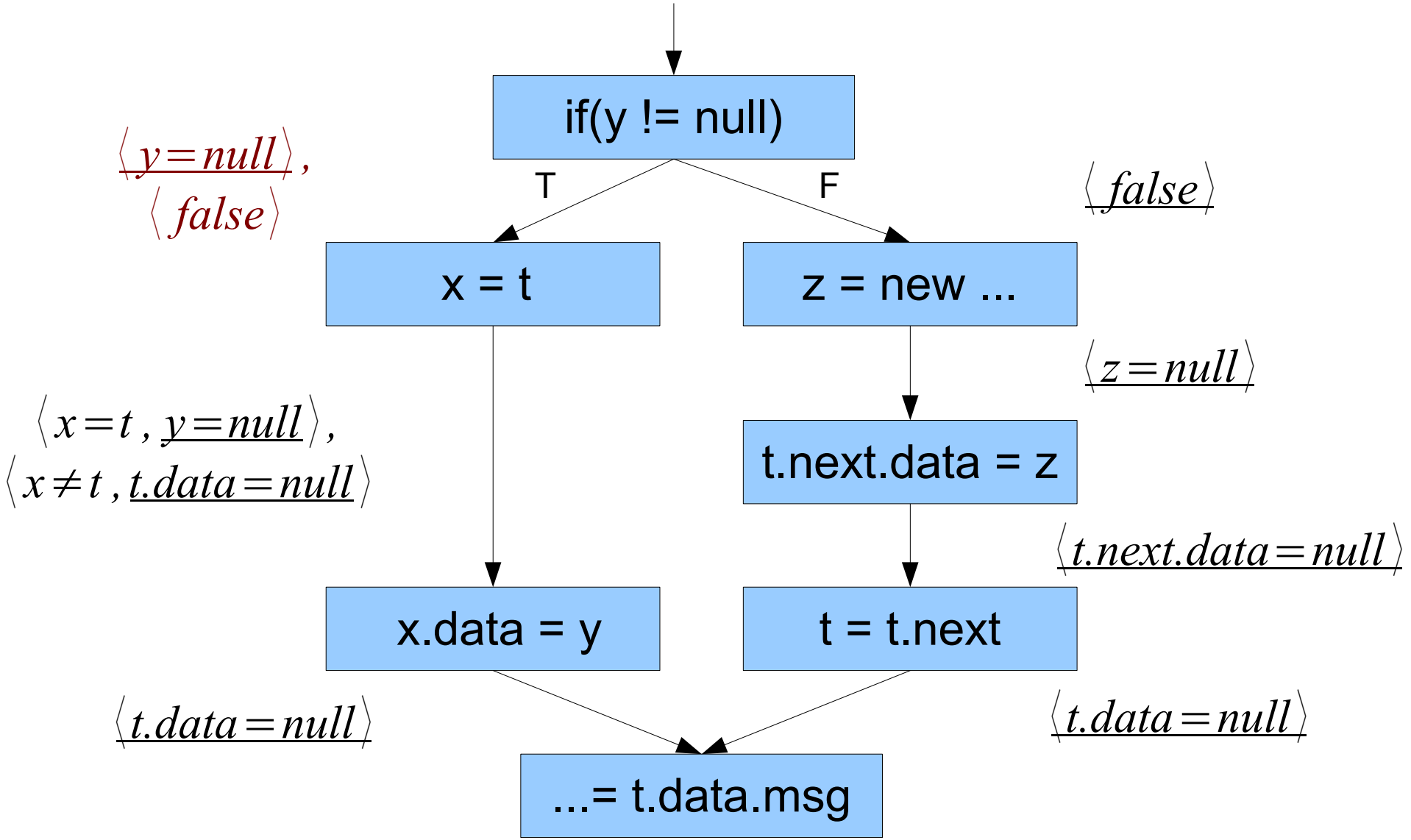
Illustration



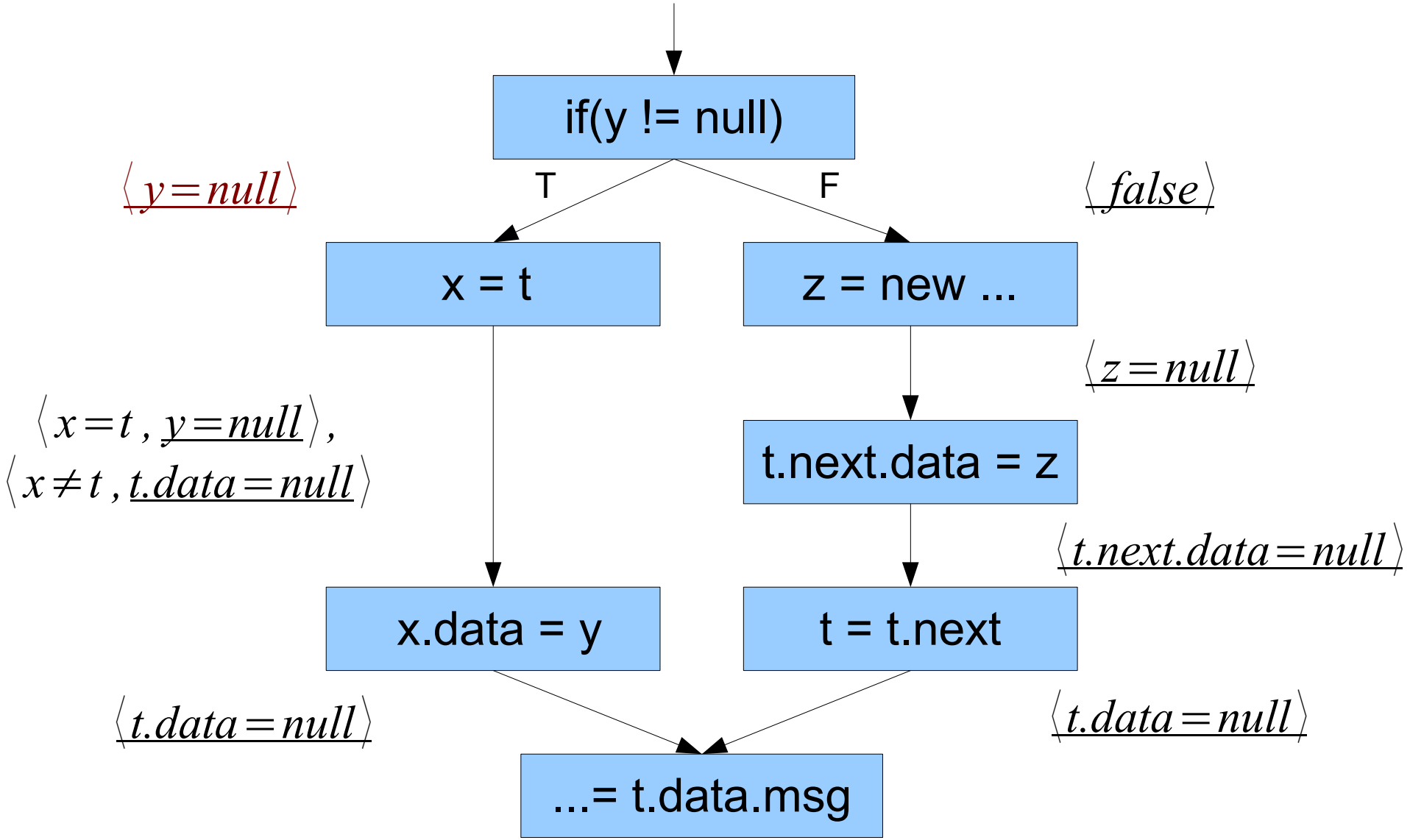
Illustration



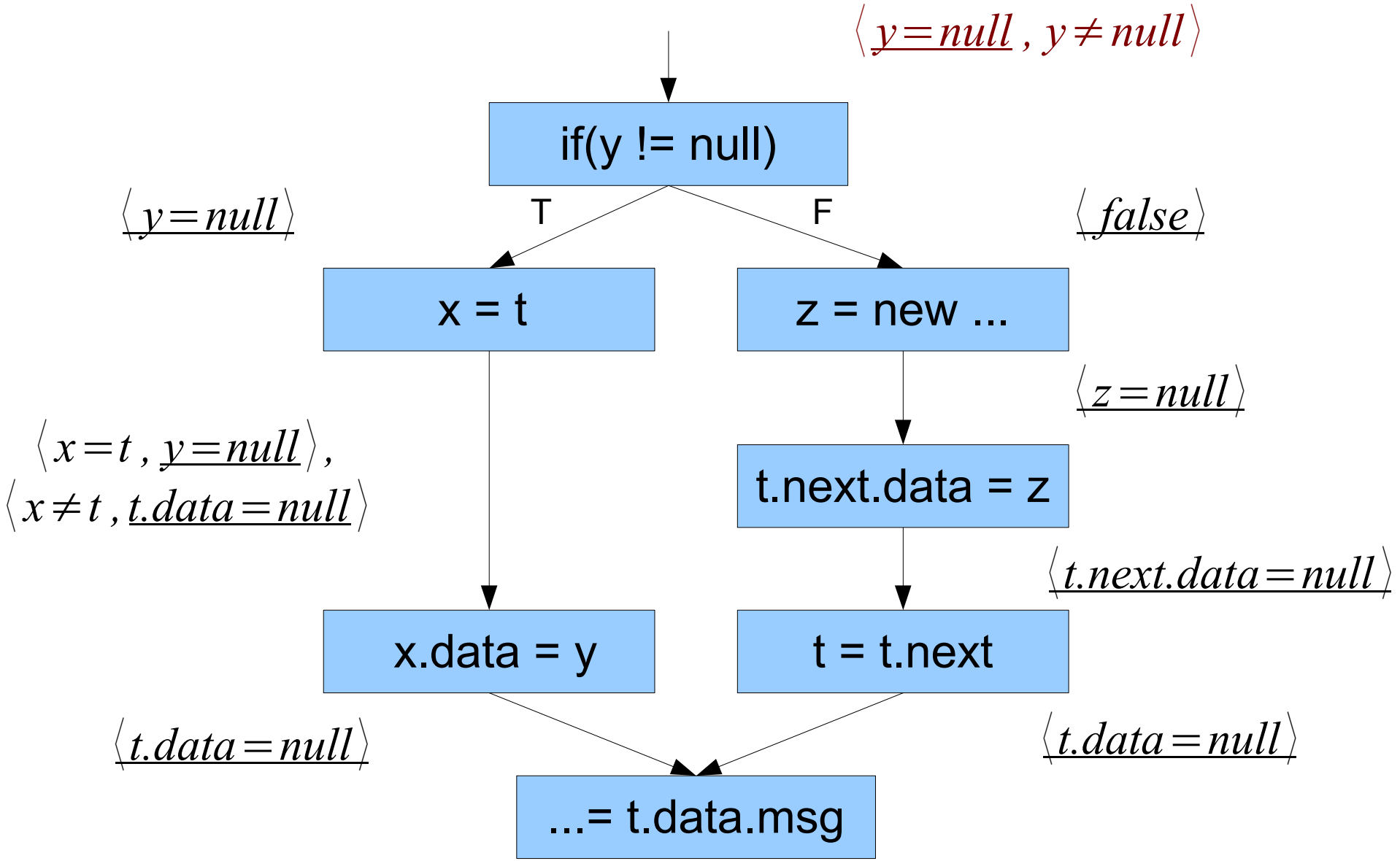
Illustration



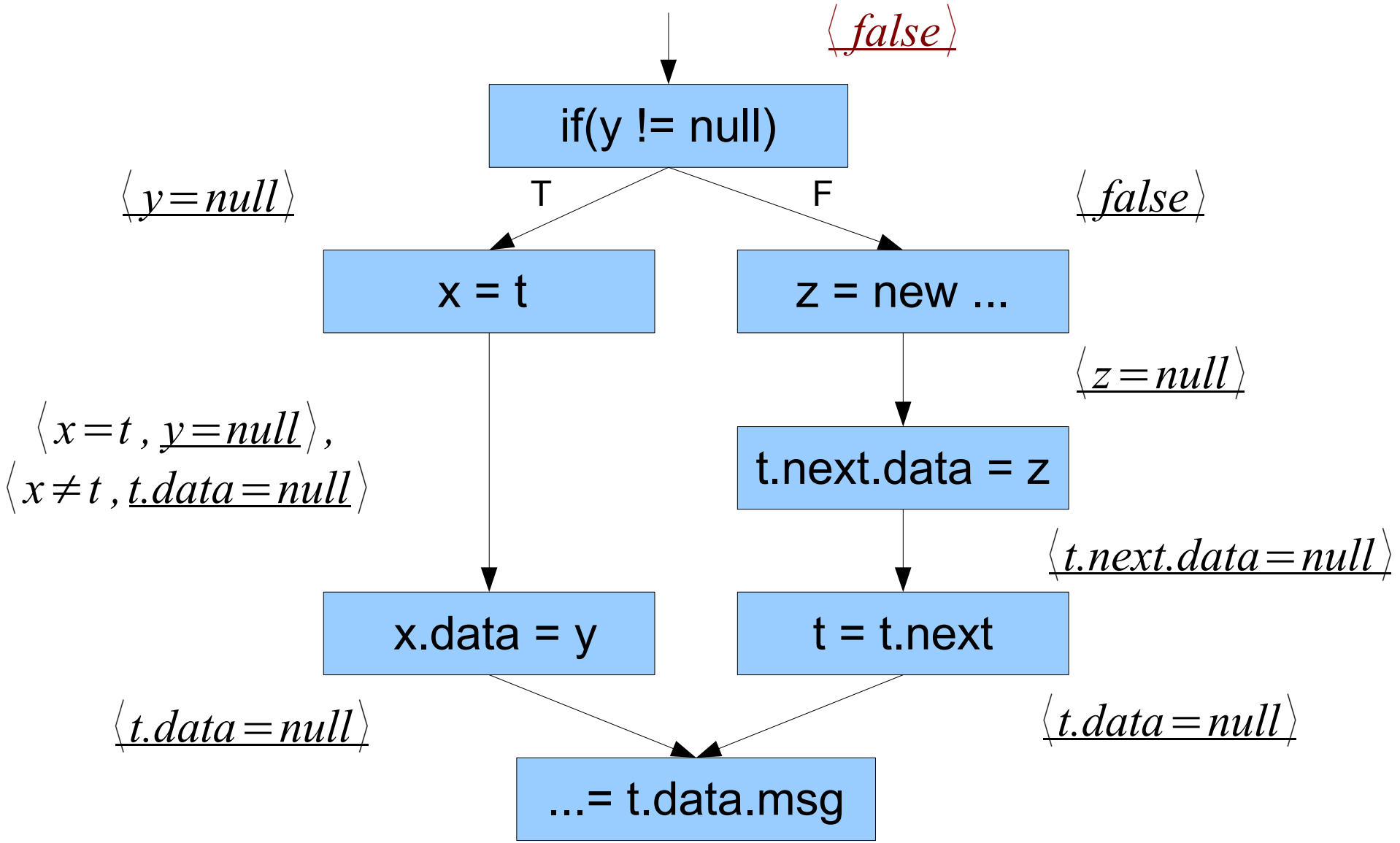
Illustration



Illustration



Illustration



Simplification rules

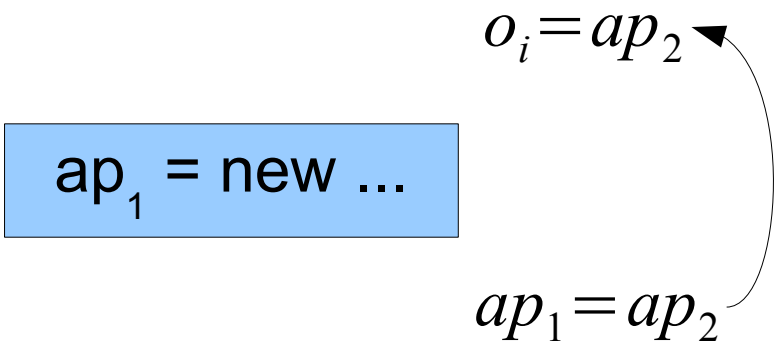
$$ap = ap \rightarrow true$$

$$\{ap_1 = ap_2, ap_1 \neq ap_2\} \rightarrow false$$

$$o_i = o_j \rightarrow false$$

$$o_i = null \rightarrow false$$

$$o_i = o_i \rightarrow true \ (\text{over-approximation})$$



Simplification rules

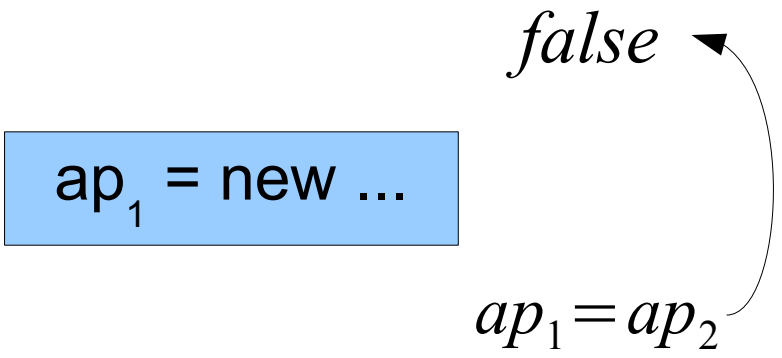
$$ap = ap \rightarrow true$$

$$\{ap_1 = ap_2, ap_1 \neq ap_2\} \rightarrow false$$

$$o_i = o_j \rightarrow false$$

$$o_i = null \rightarrow false$$

$$o_i = o_i \rightarrow true \ (\text{over-approximation})$$



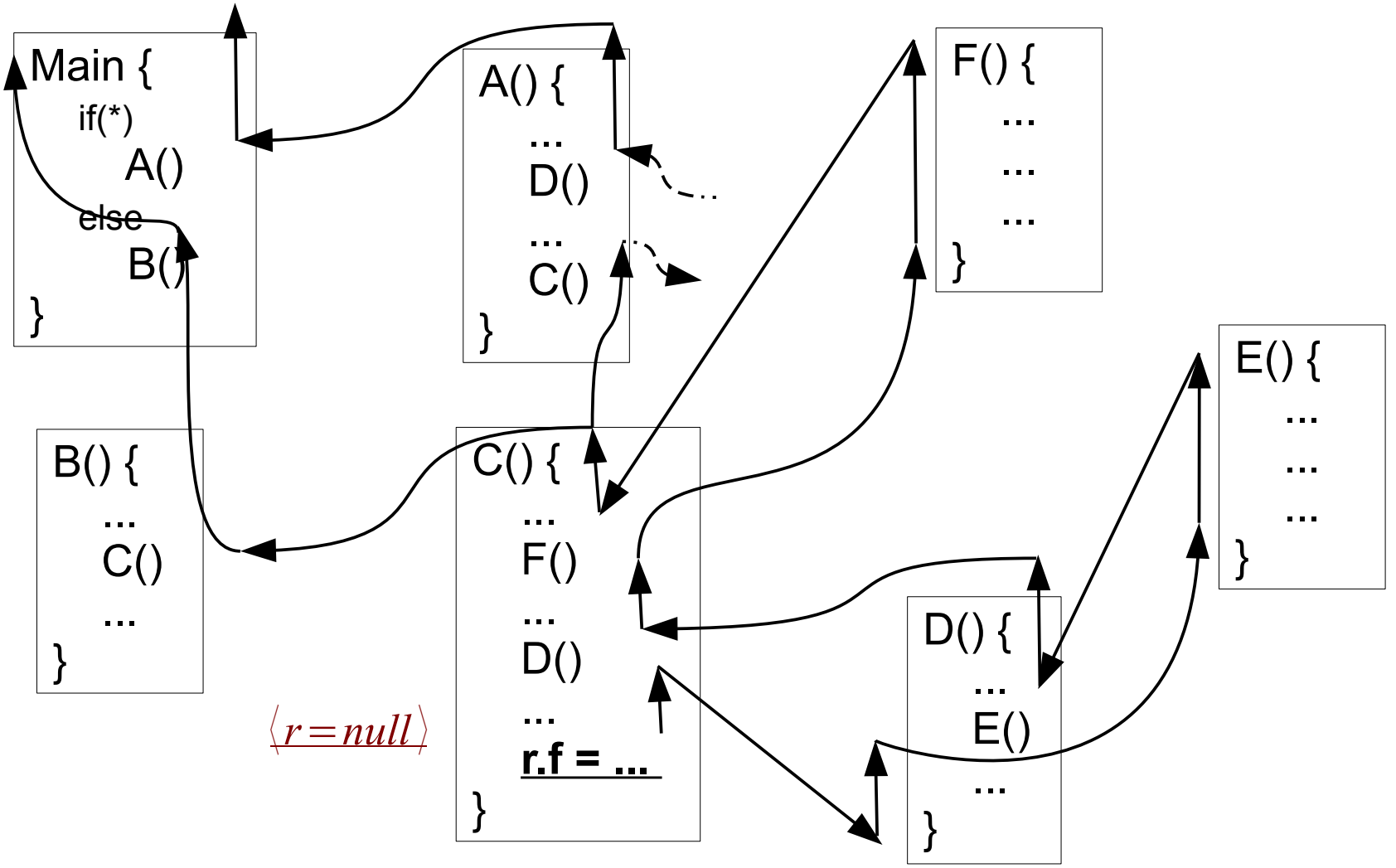
Abstract Interpretation Formulation

- **Concrete semantics**
  - **Domain:** sets of concrete stores ordered by set inclusion
  - Backward collecting semantics
    - set union is the join operator
  - $\gamma(\phi) = \{ s \mid s \; satisfies \; \phi \}$

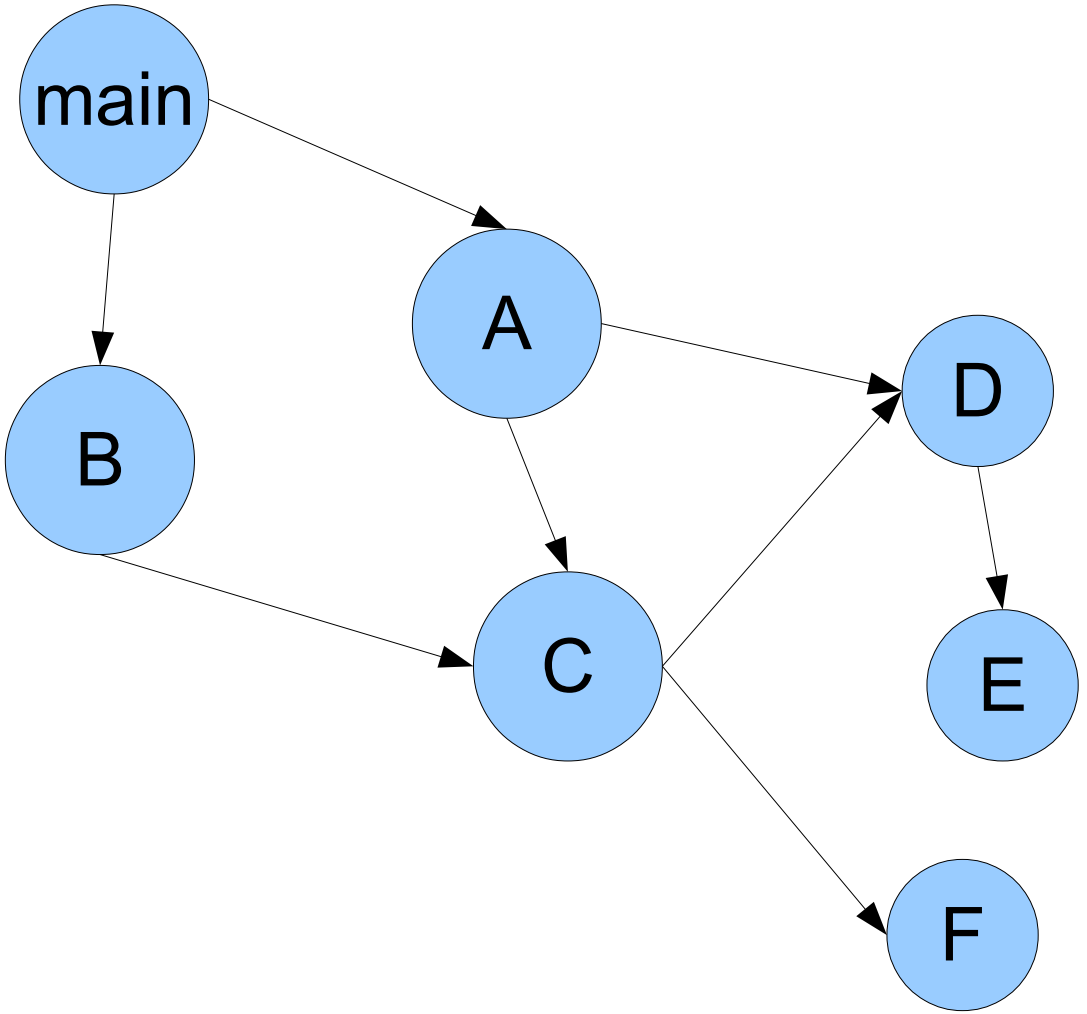
Abstract Interpretation Formulation

- **Abstract semantics:**
  - $\phi_1 \leq \phi_2 \;\; iff \;\; \phi_1 \subseteq \phi_2$
  - $\phi_1 \subseteq \phi_2 \Rightarrow \phi_1 \Rightarrow \phi_2$, but the converse does not hold
  - Transfer functions and simplification rules are monotonic
  - Abstract transfer functions over-approximate the concrete transfer functions

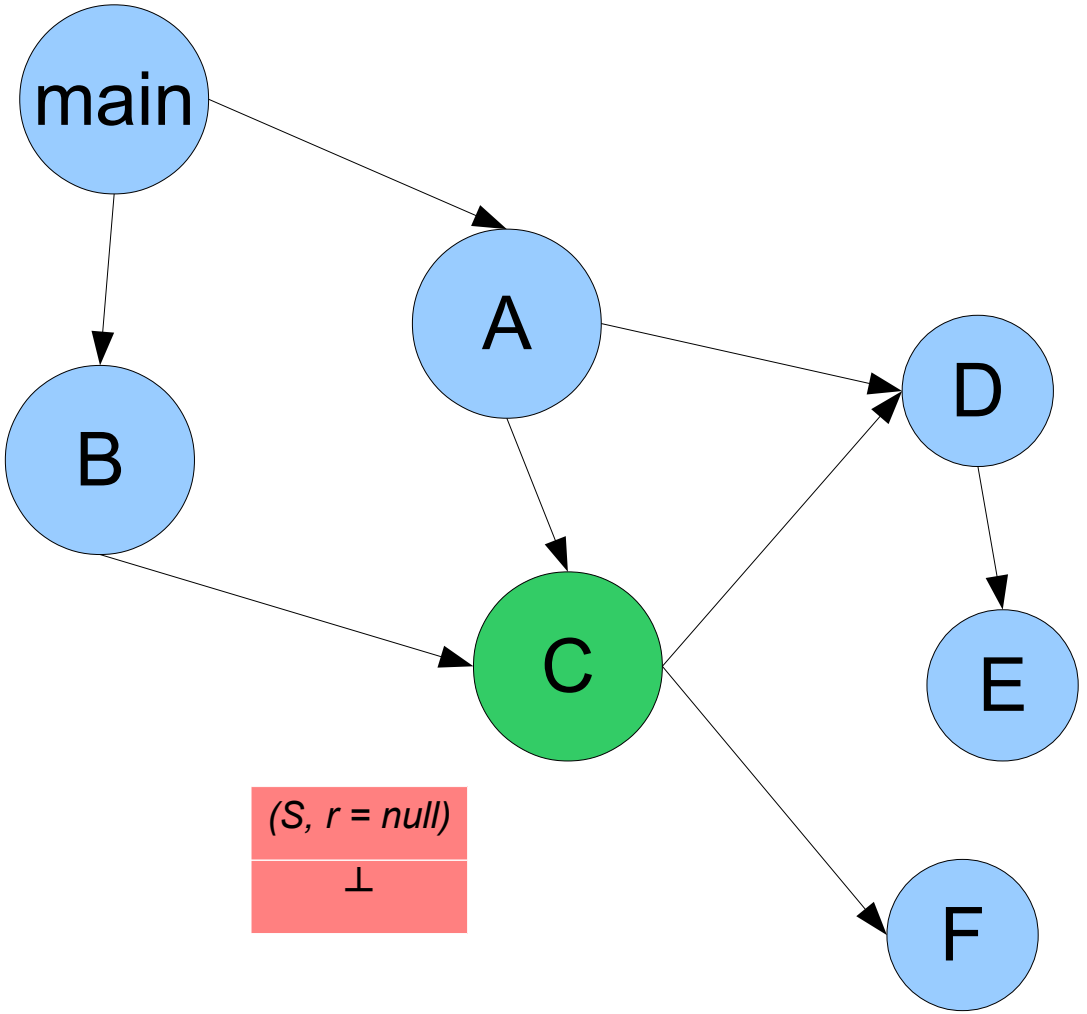
Inter-procedural analysis



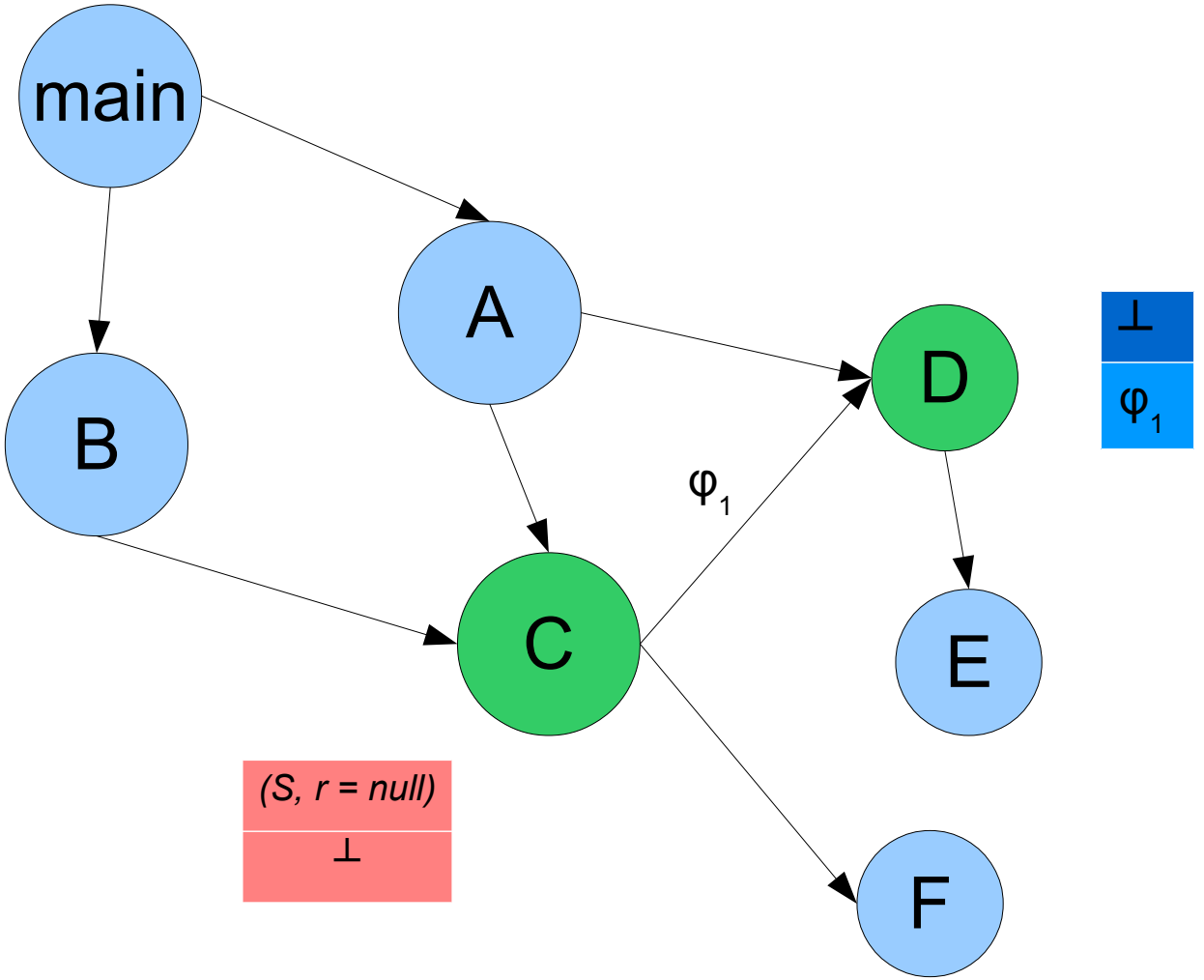
Inter-procedural analysis



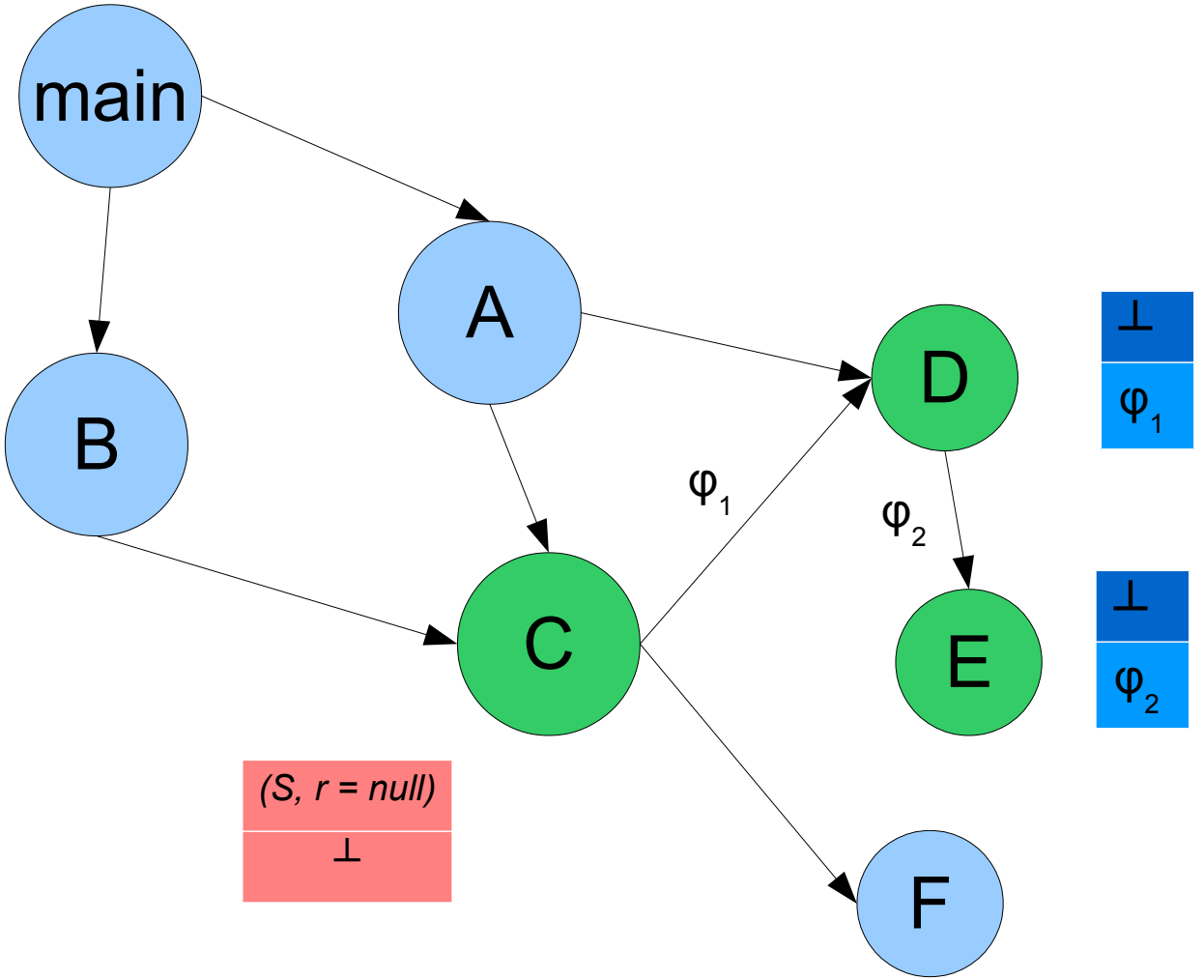
Inter-procedural analysis



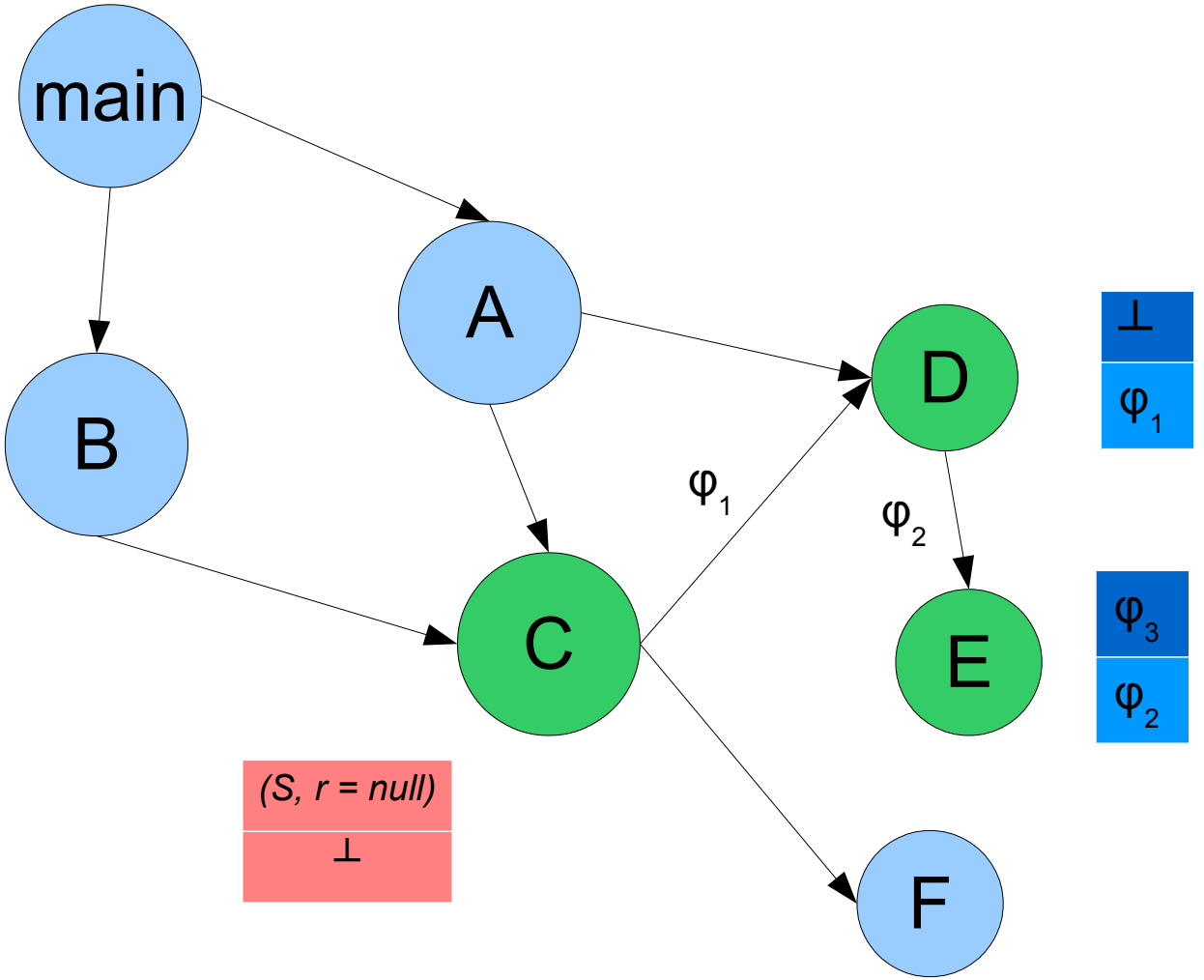
Inter-procedural analysis



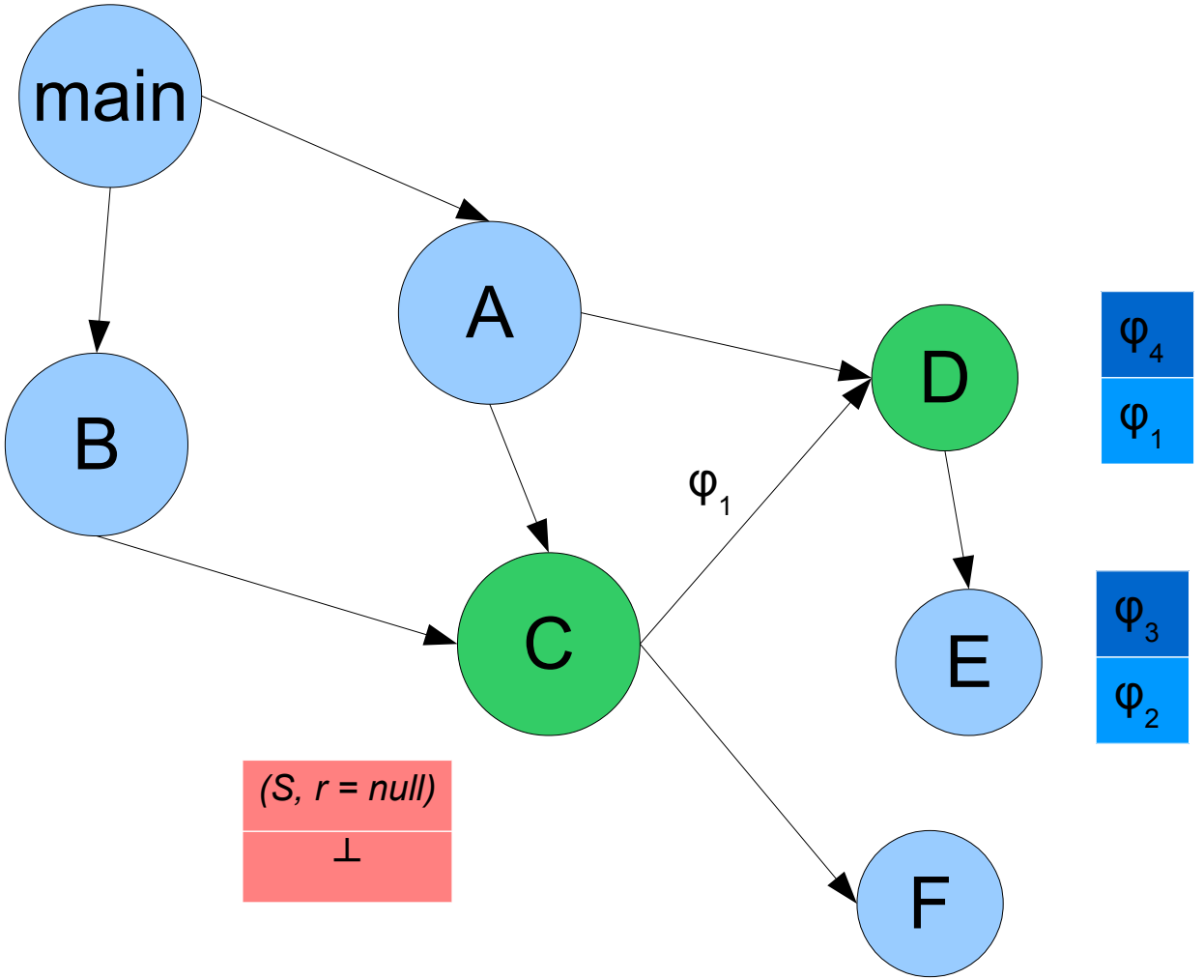
Inter-procedural analysis



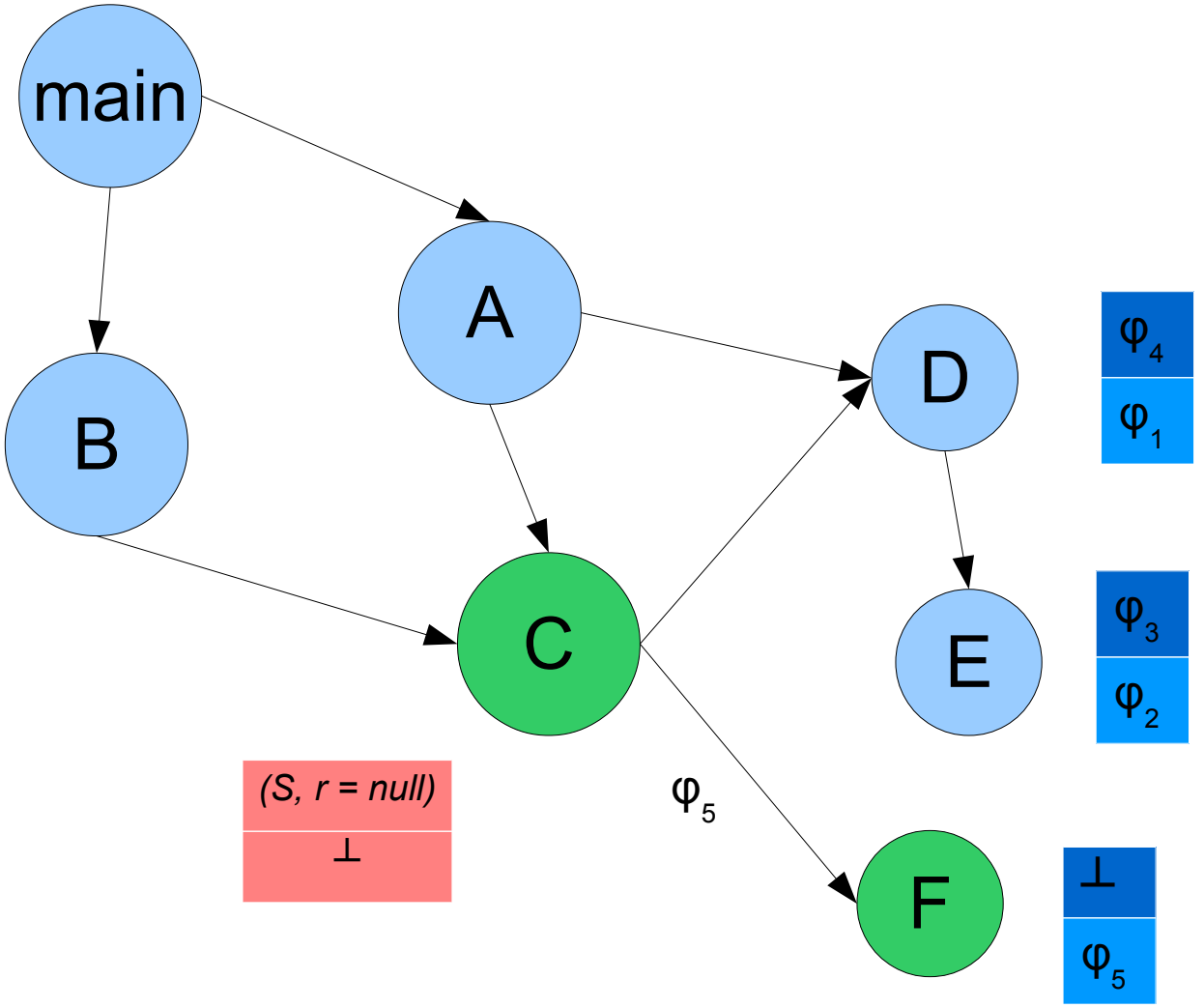
Inter-procedural analysis



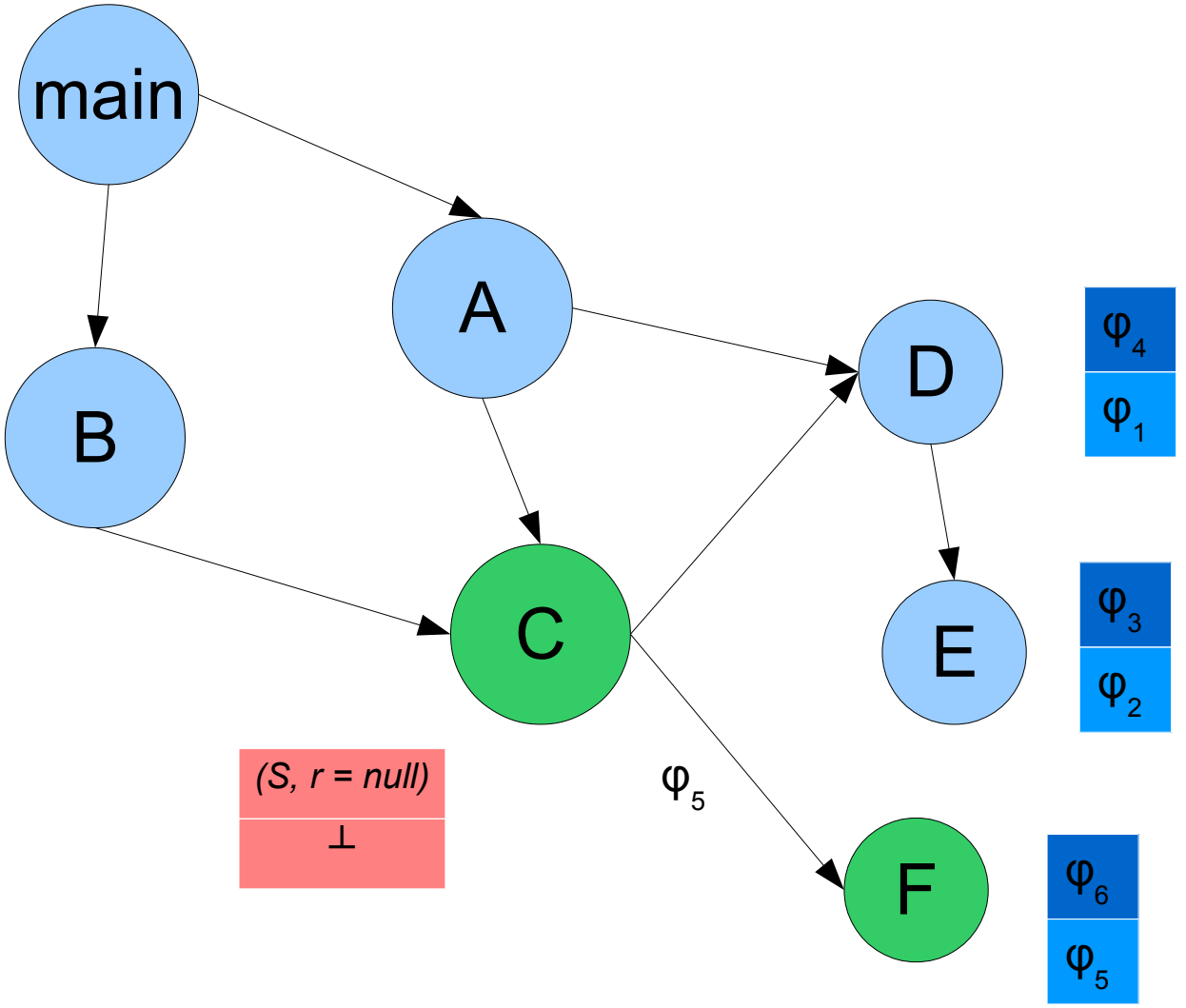
Inter-procedural analysis



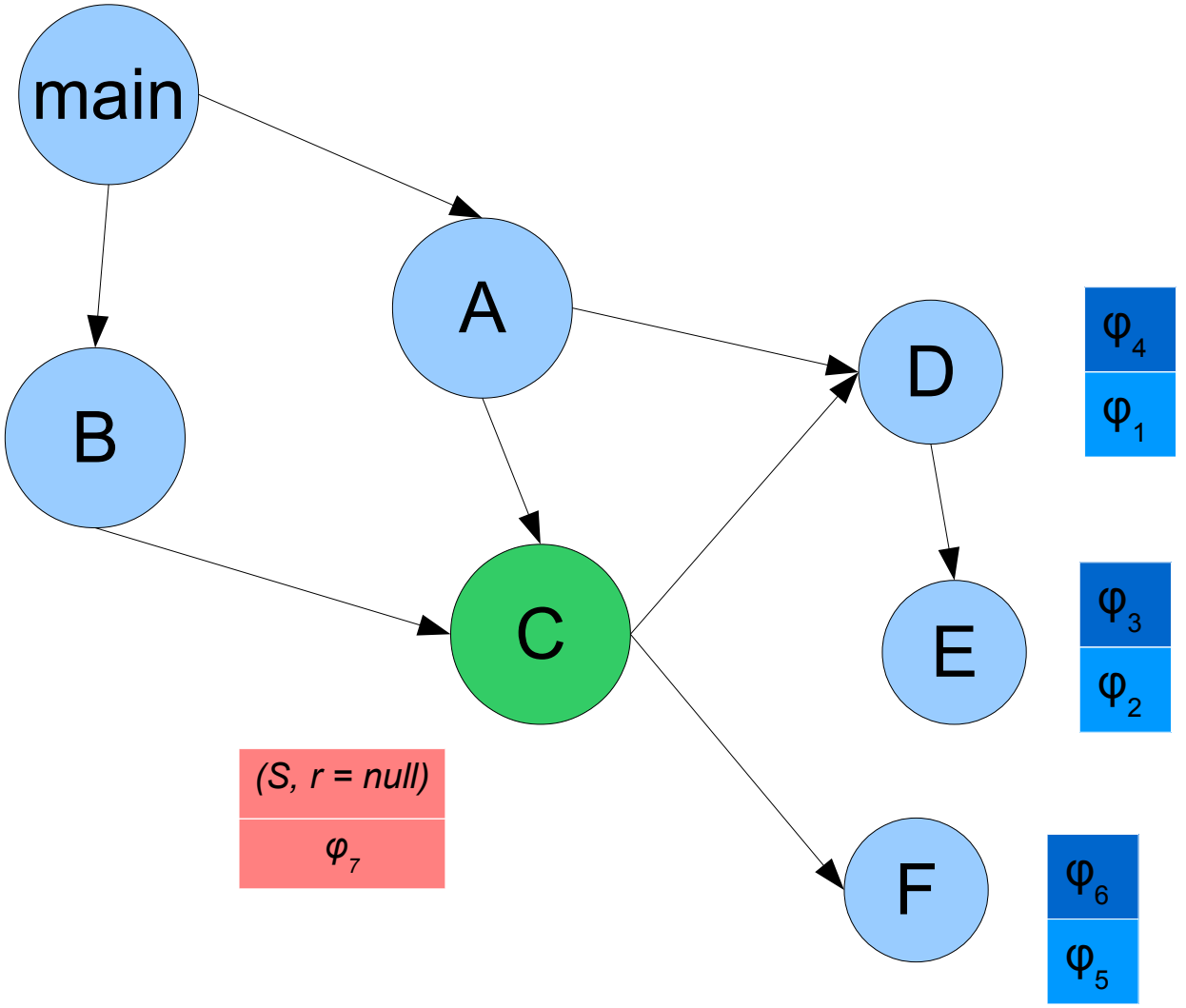
Inter-procedural analysis



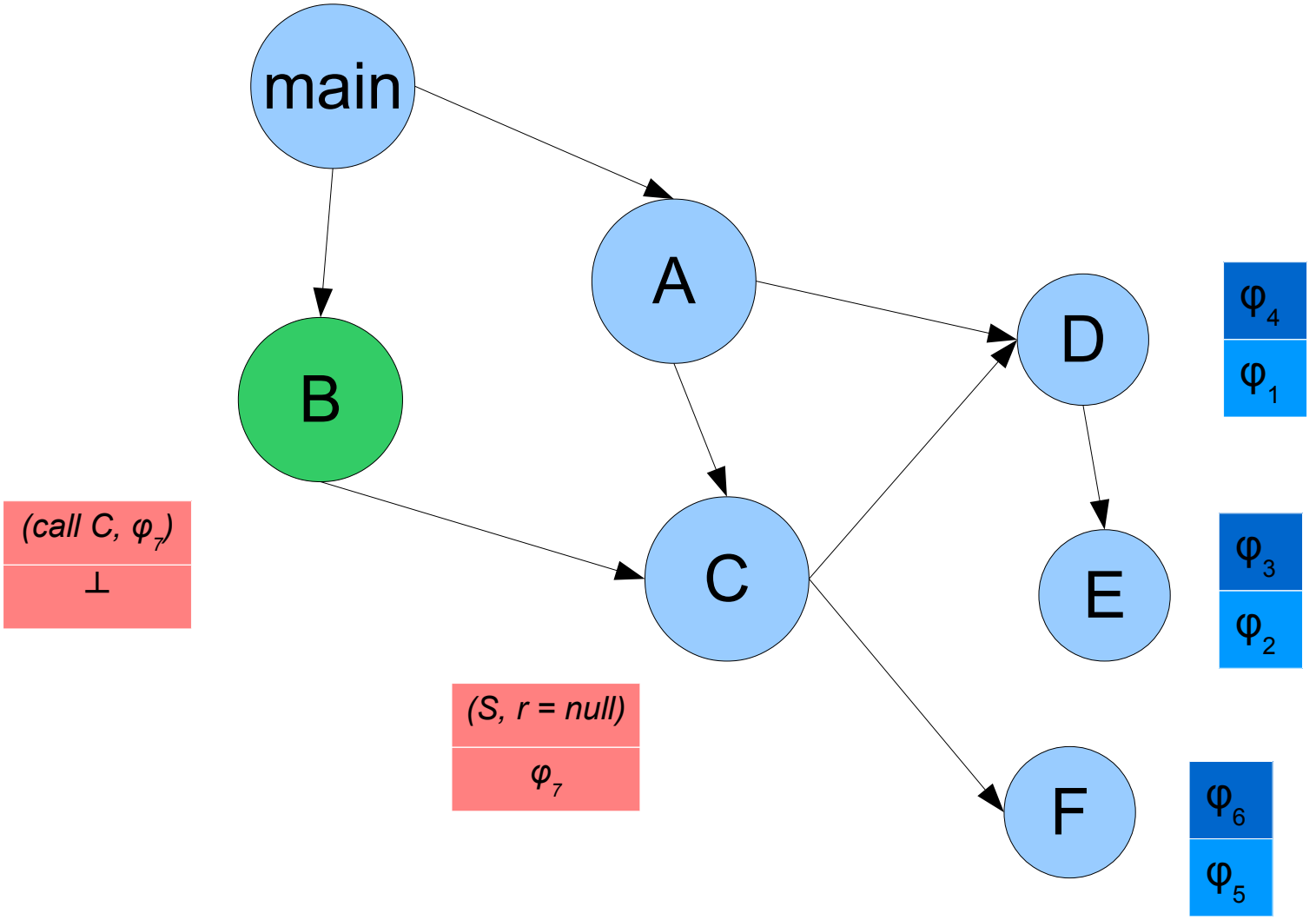
Inter-procedural analysis



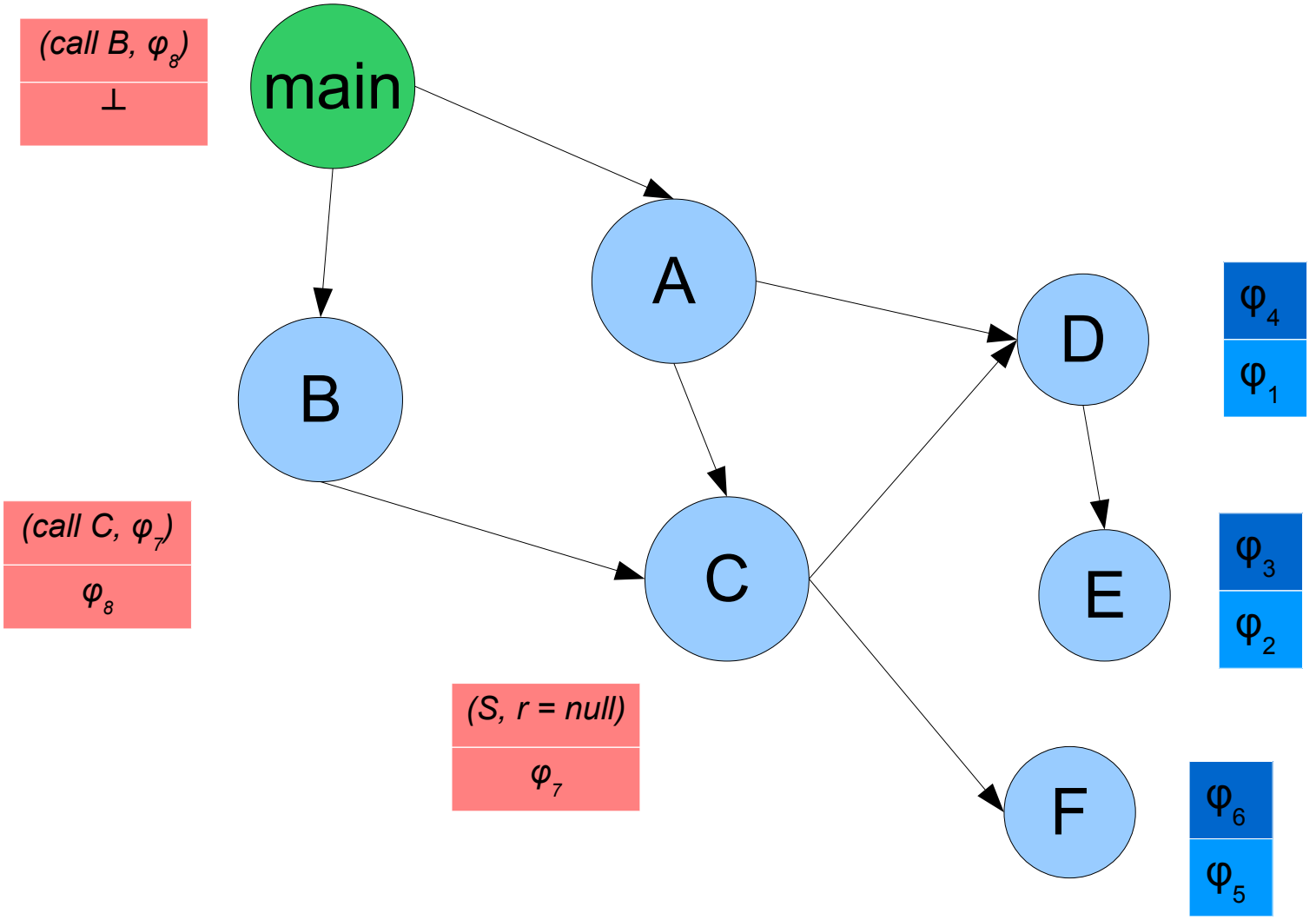
Inter-procedural analysis



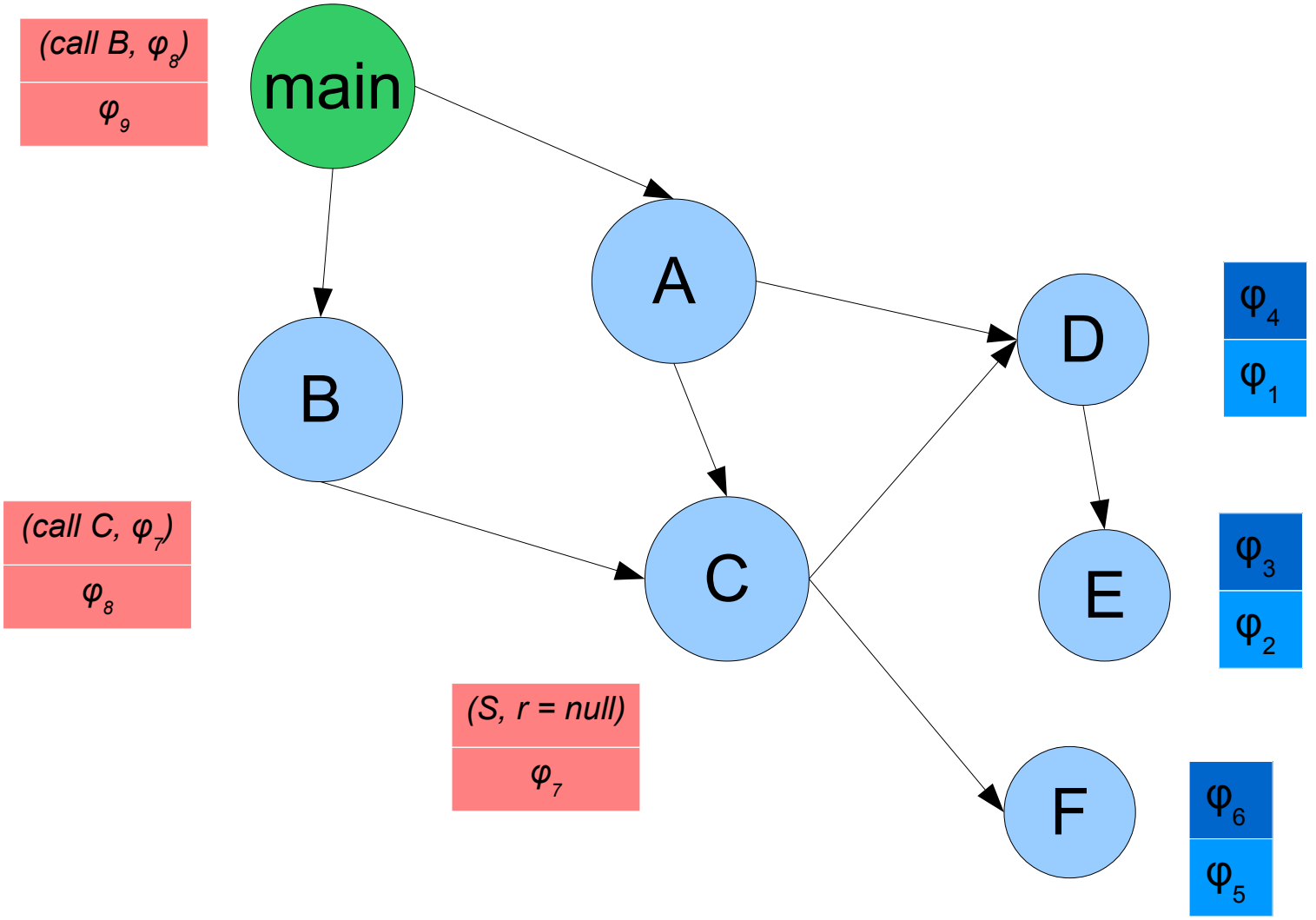
Inter-procedural analysis



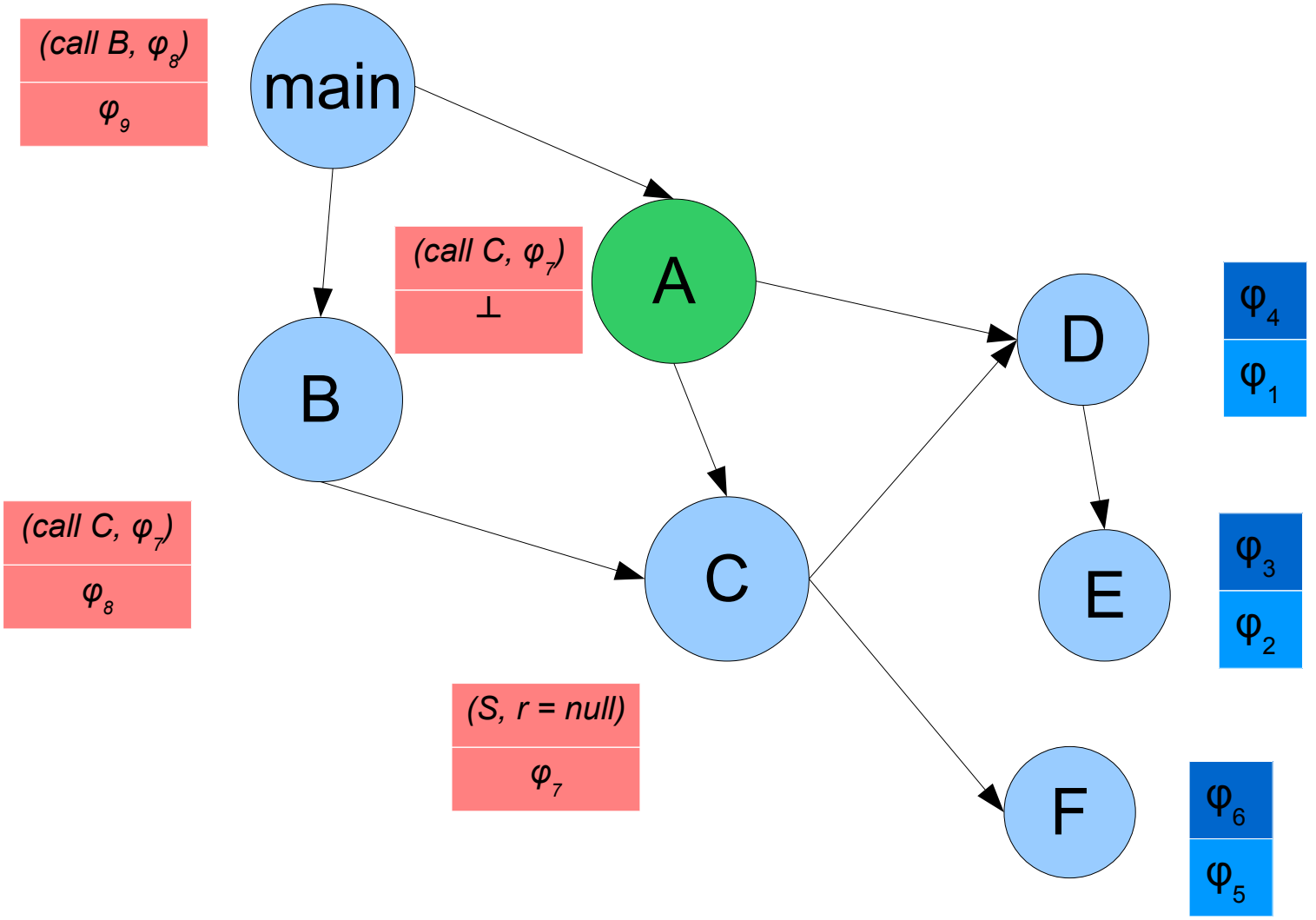
Inter-procedural analysis



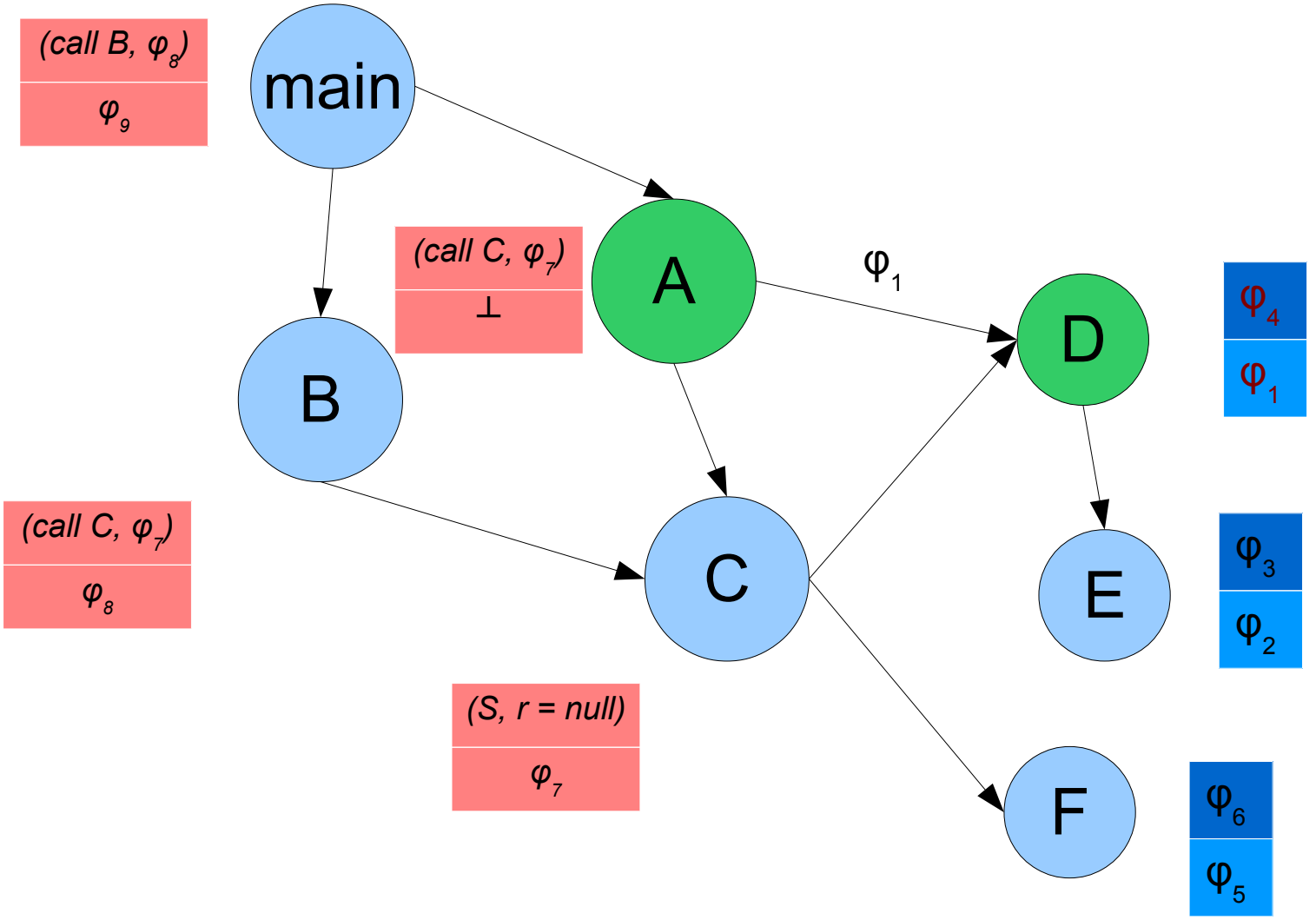
Inter-procedural analysis



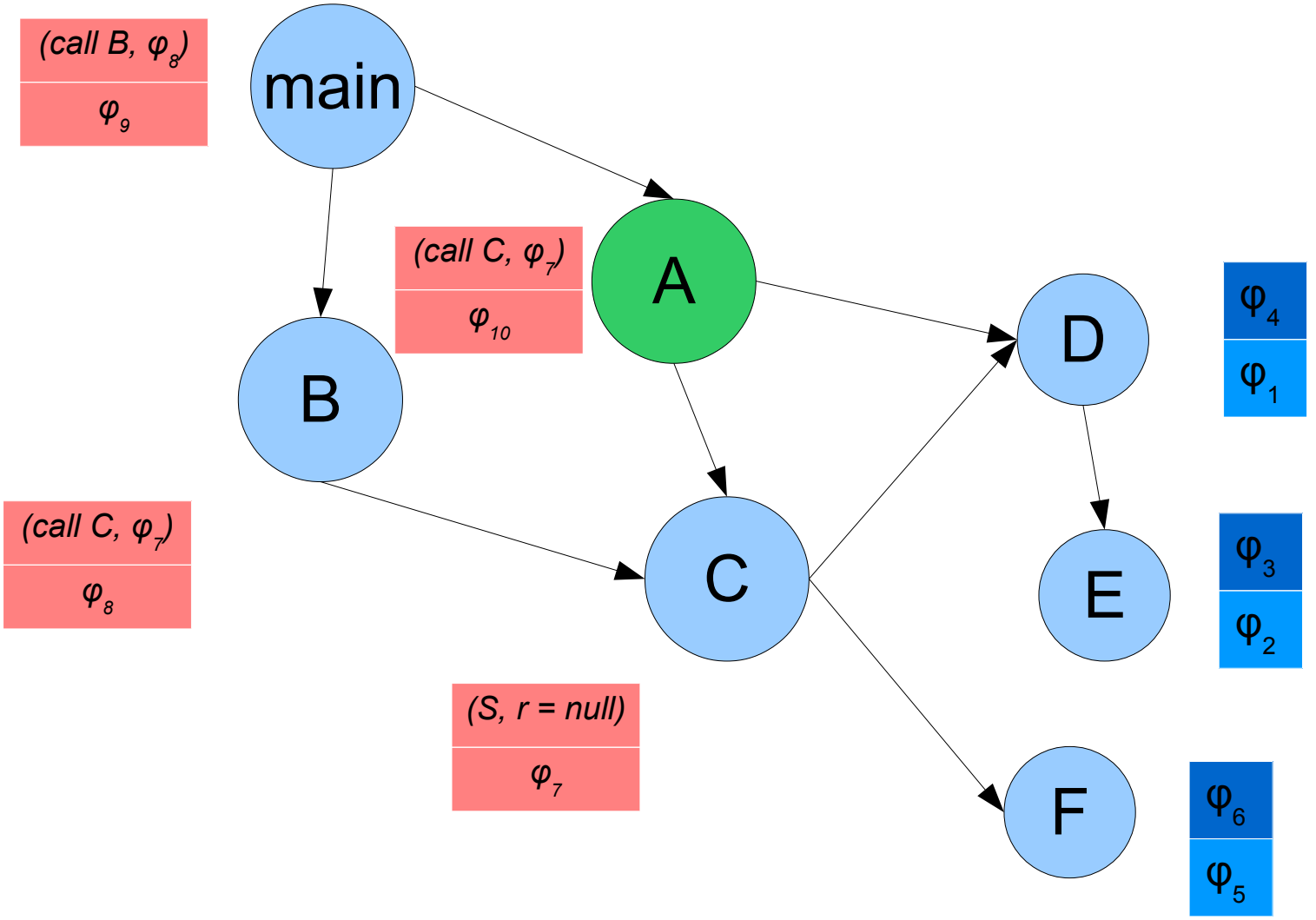
Inter-procedural analysis



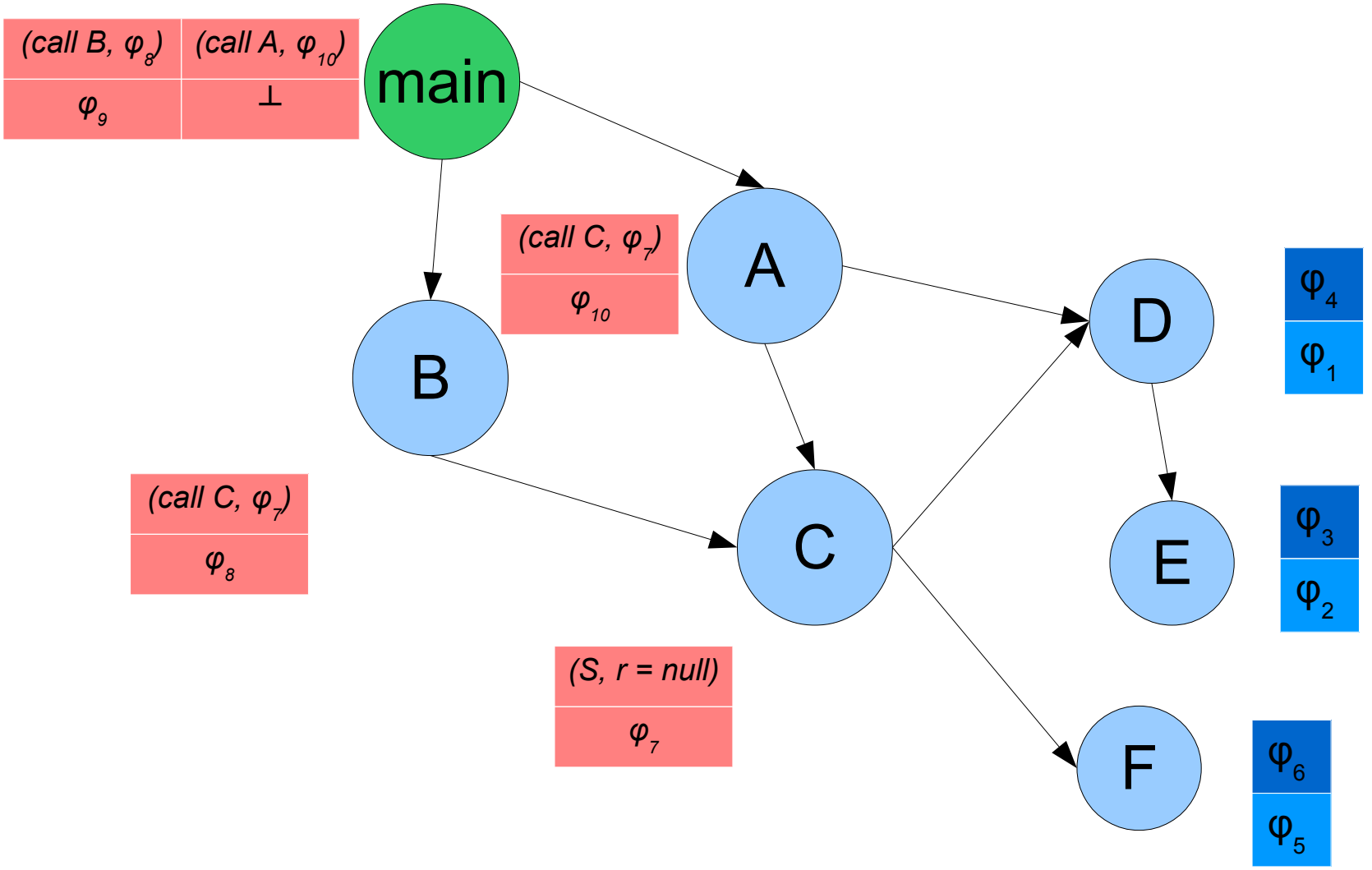
Inter-procedural analysis



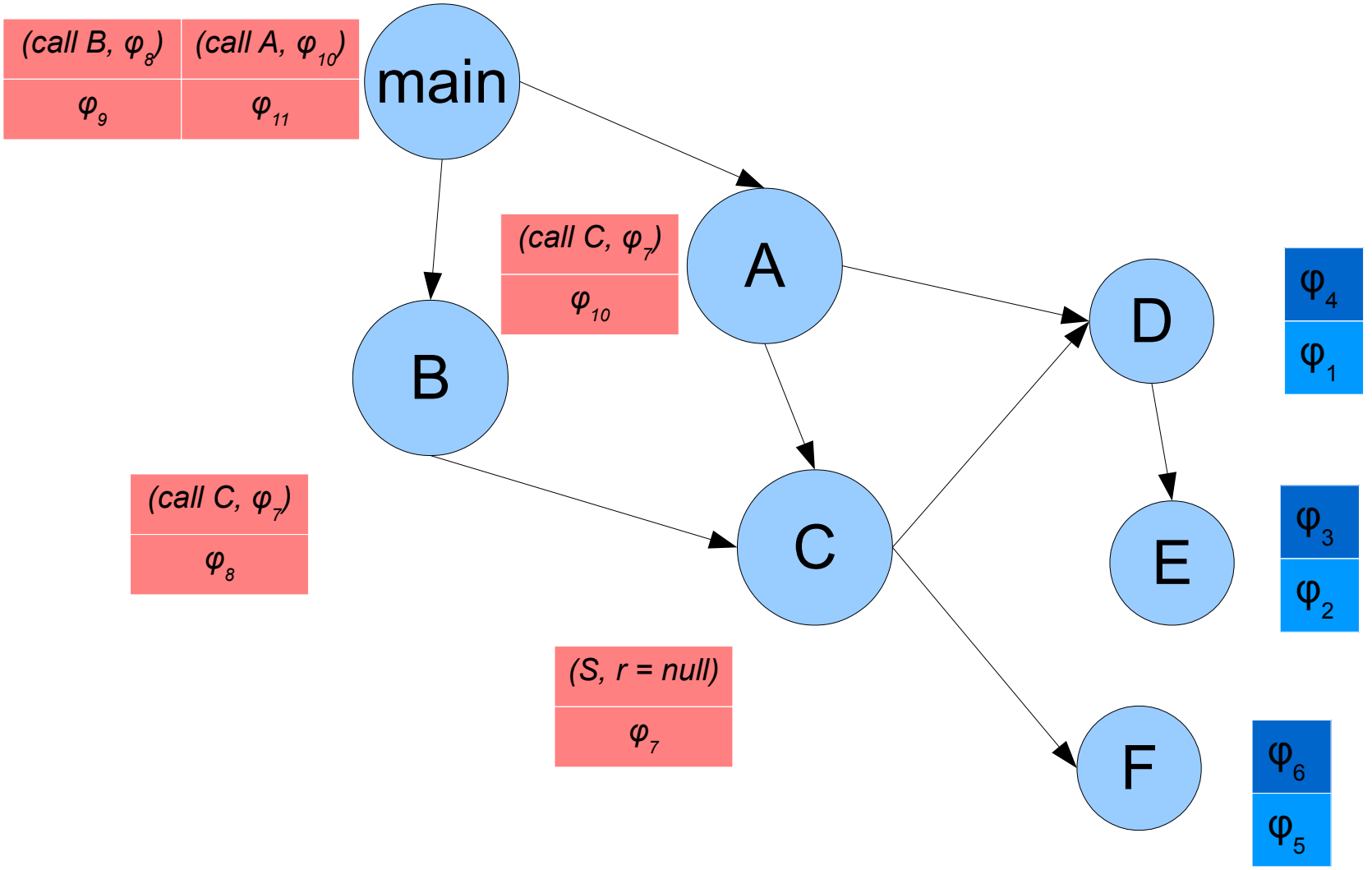
Inter-procedural analysis



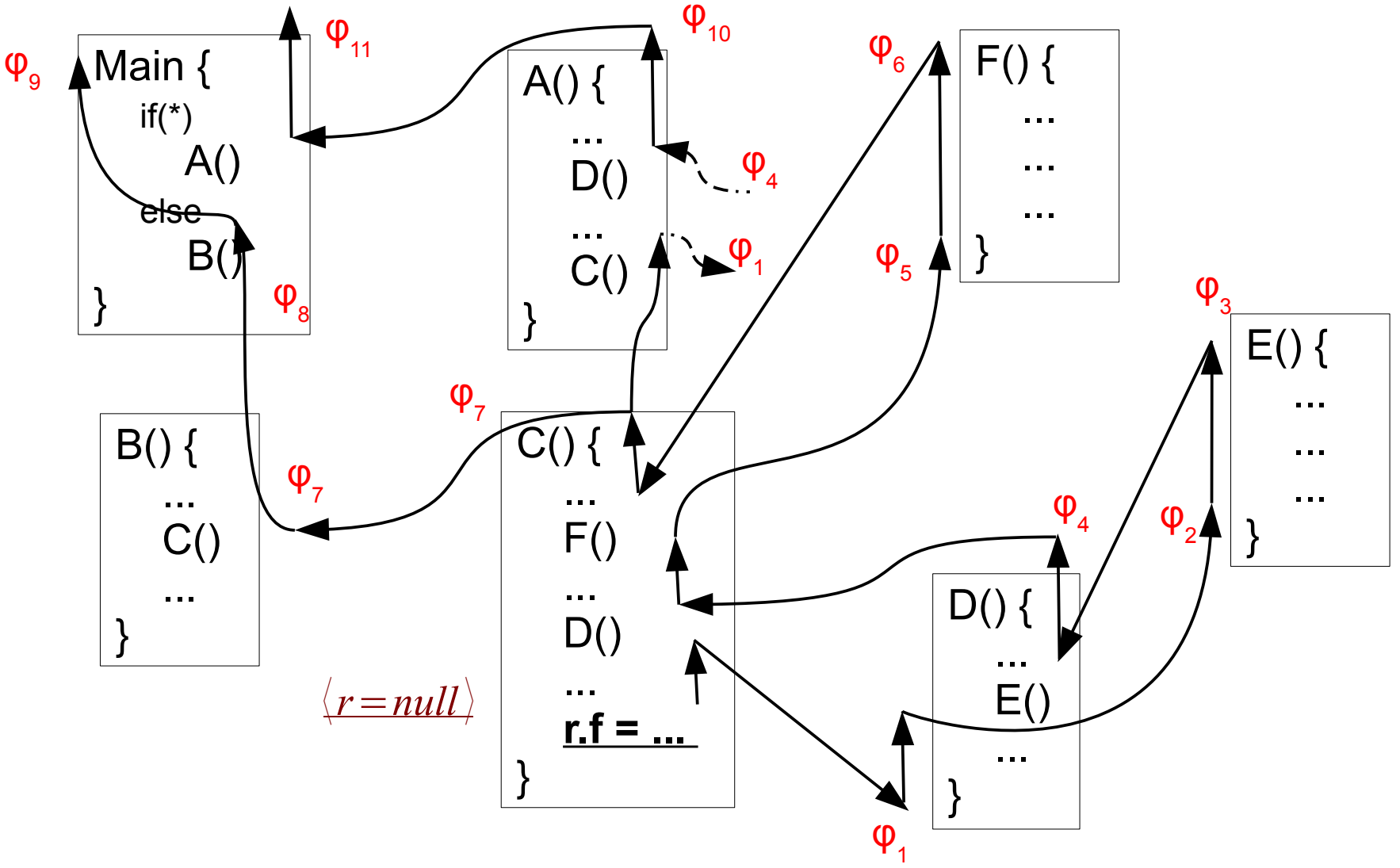
Inter-procedural analysis



Inter-procedural analysis



Inter-procedural analysis

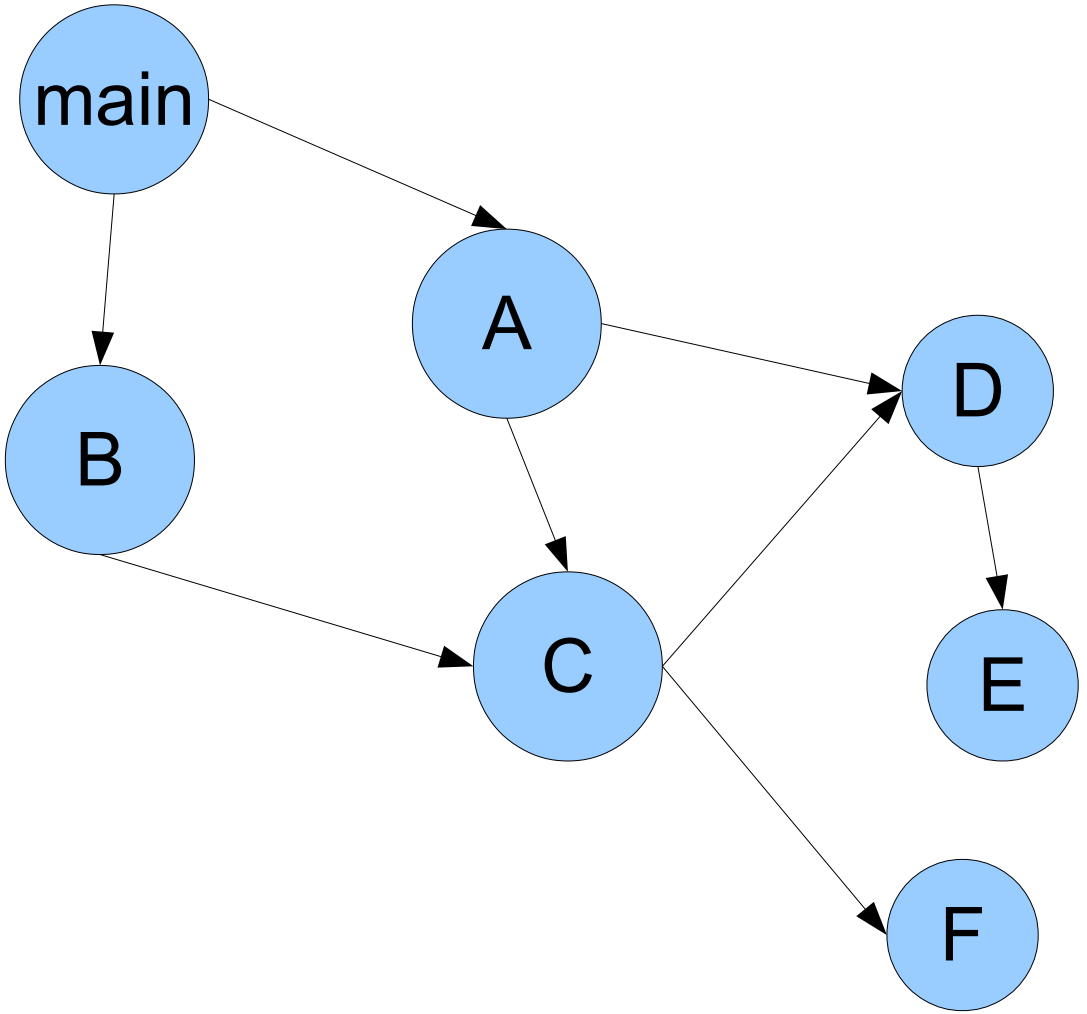


Inter-procedural analysis

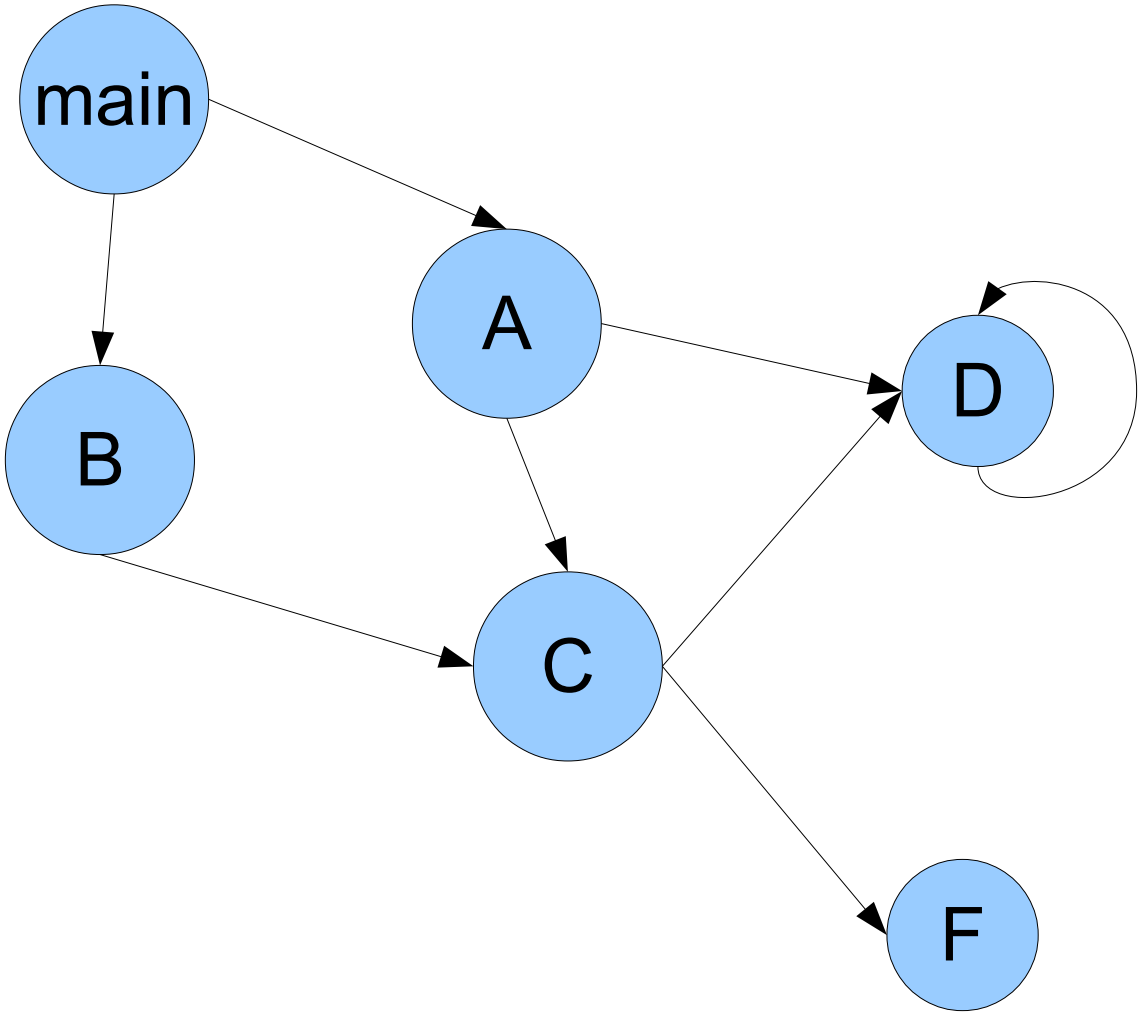
- Uses depth-first strategy instead of conventional chaotic iteration
  - Analyzes callees before callers

- **Pros:**
  - Uses less memory
  - Can abort search on discovering a satisfiable path

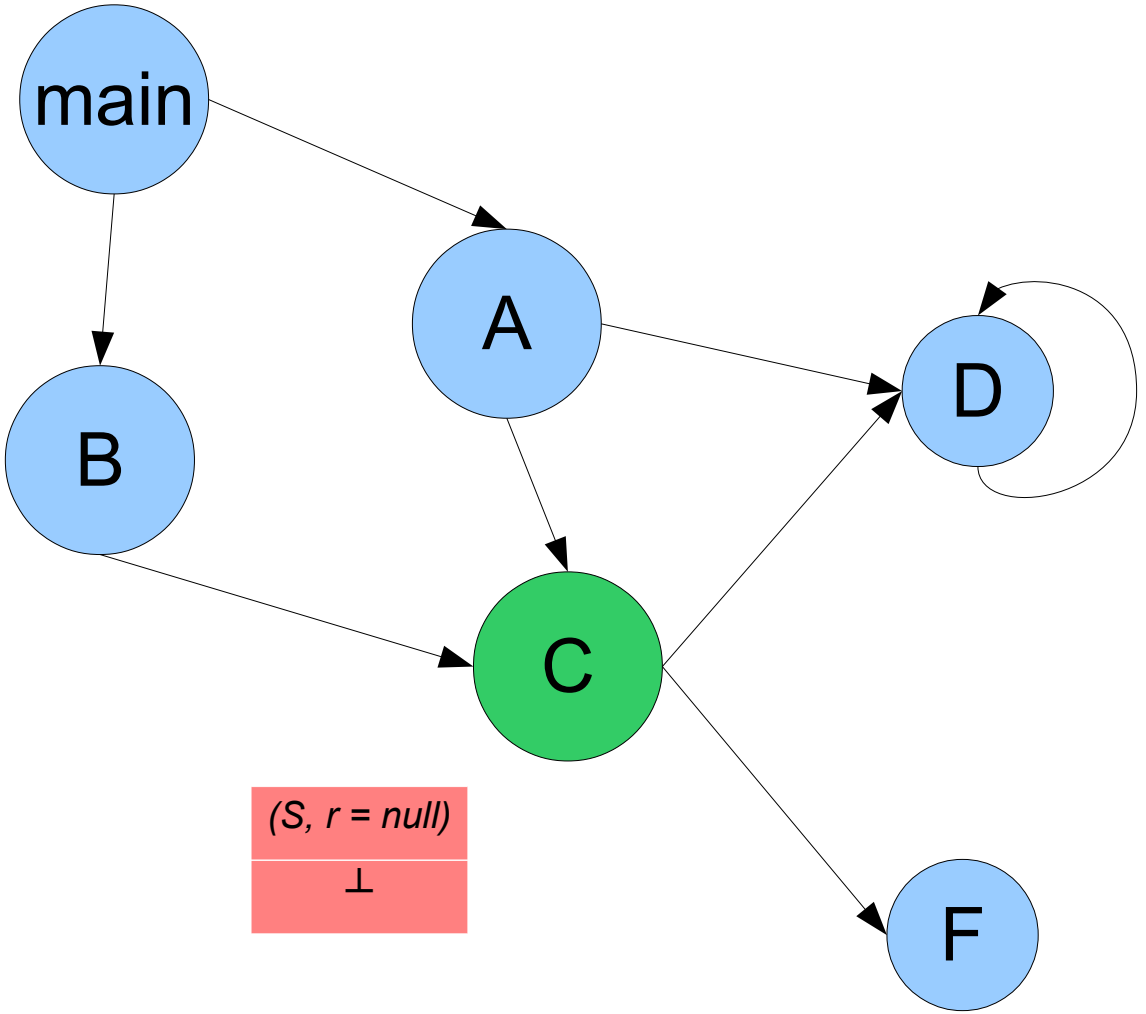
Handling Recursion



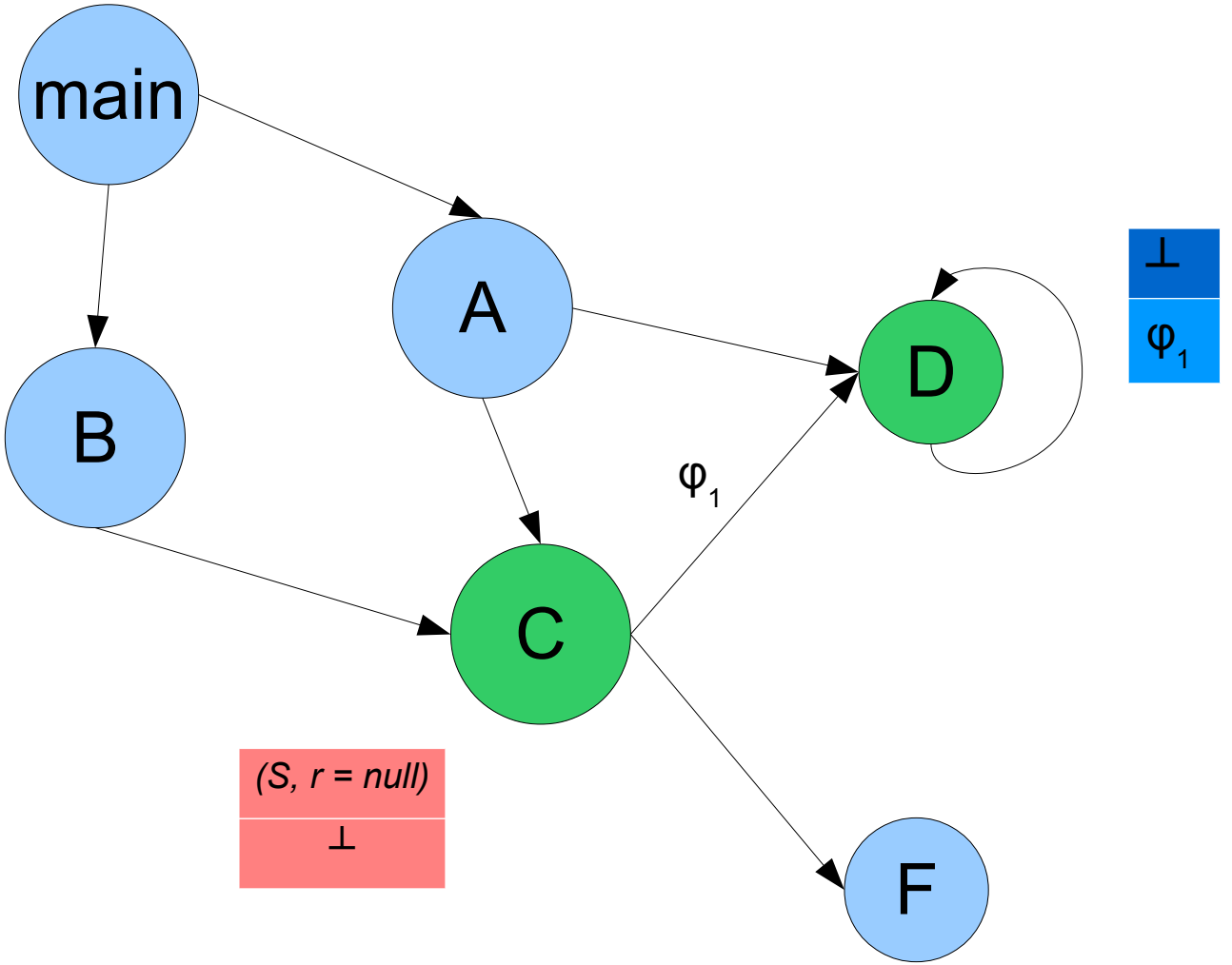
Handling Recursion



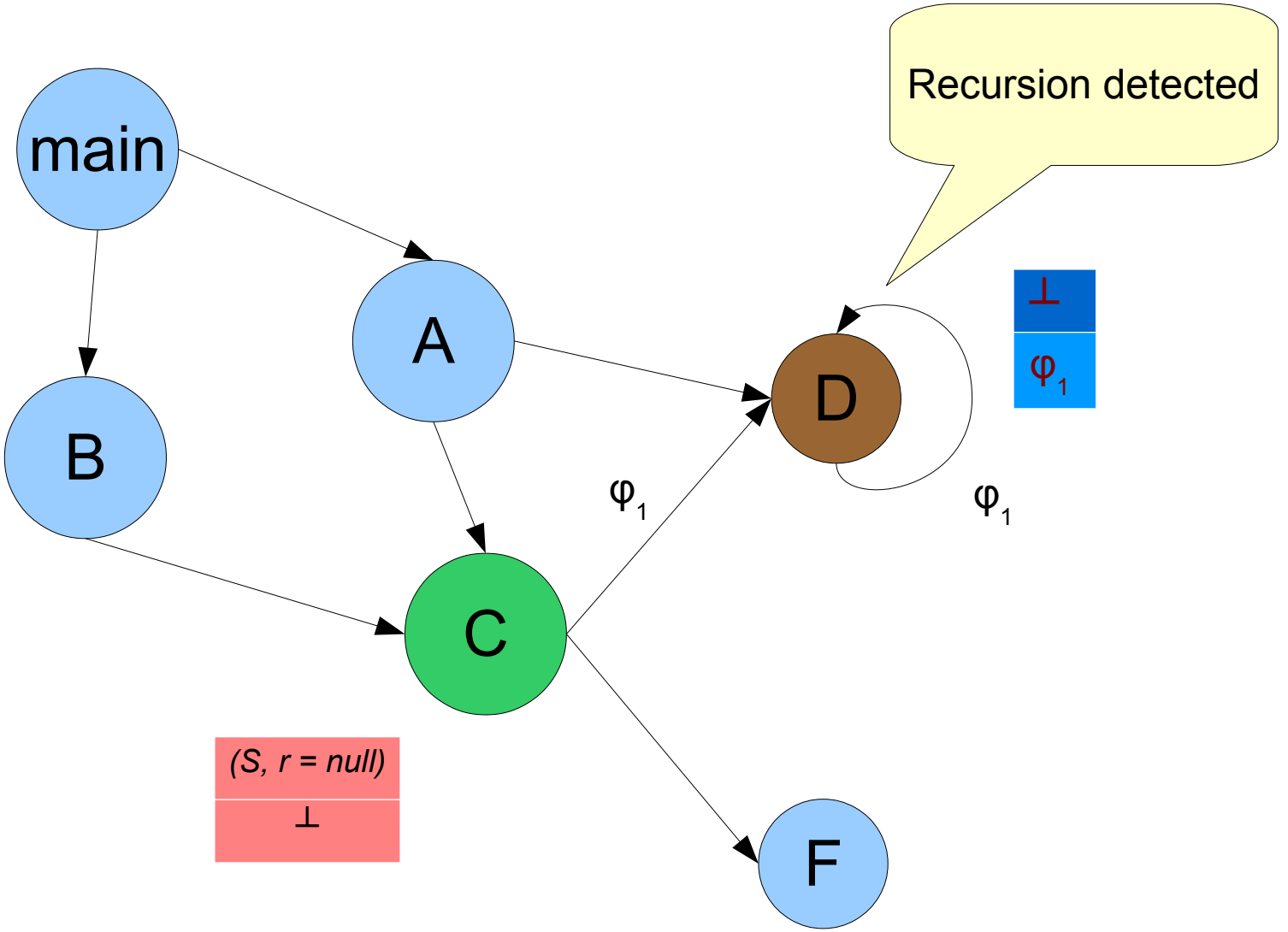
Handling Recursion



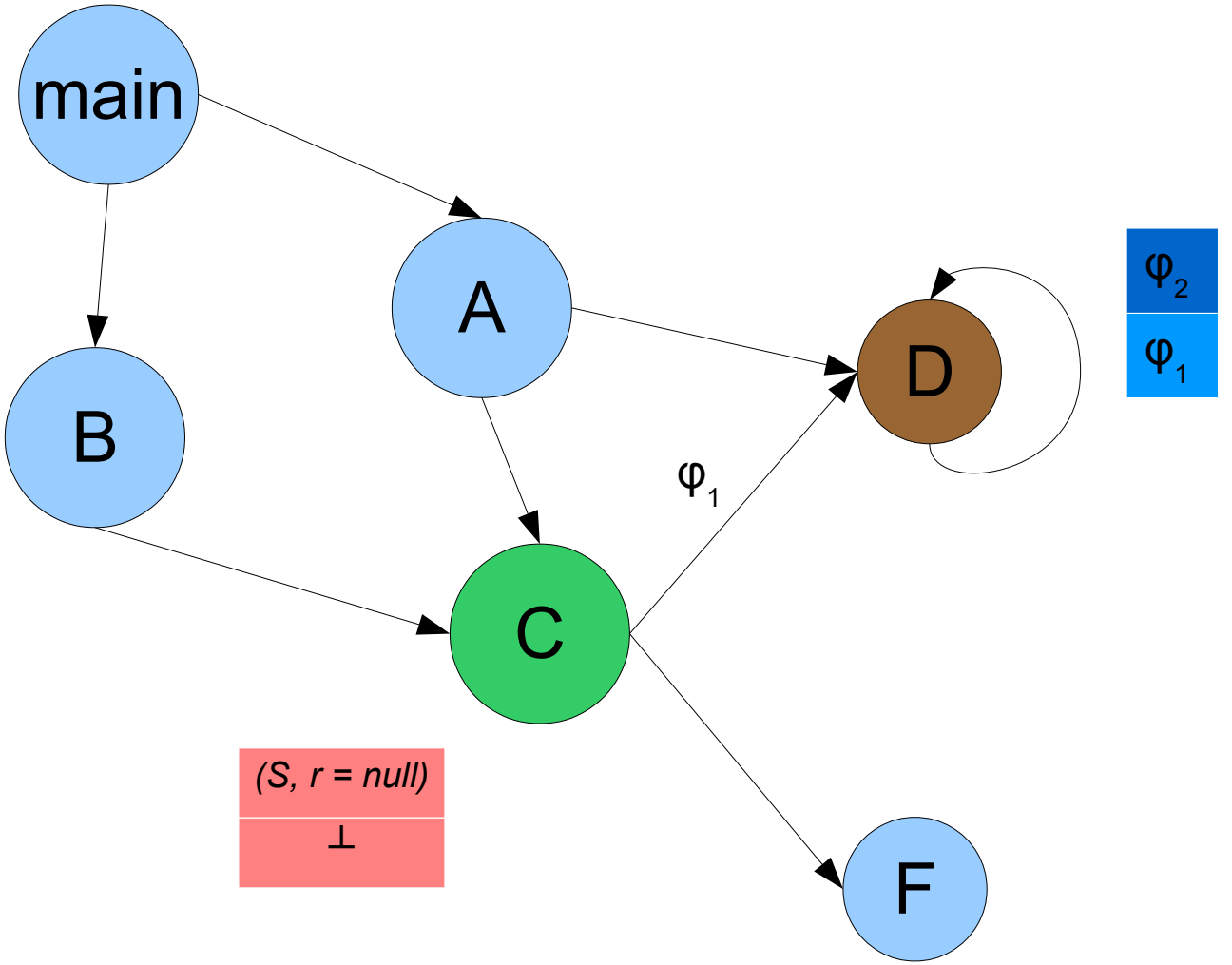
Handling Recursion



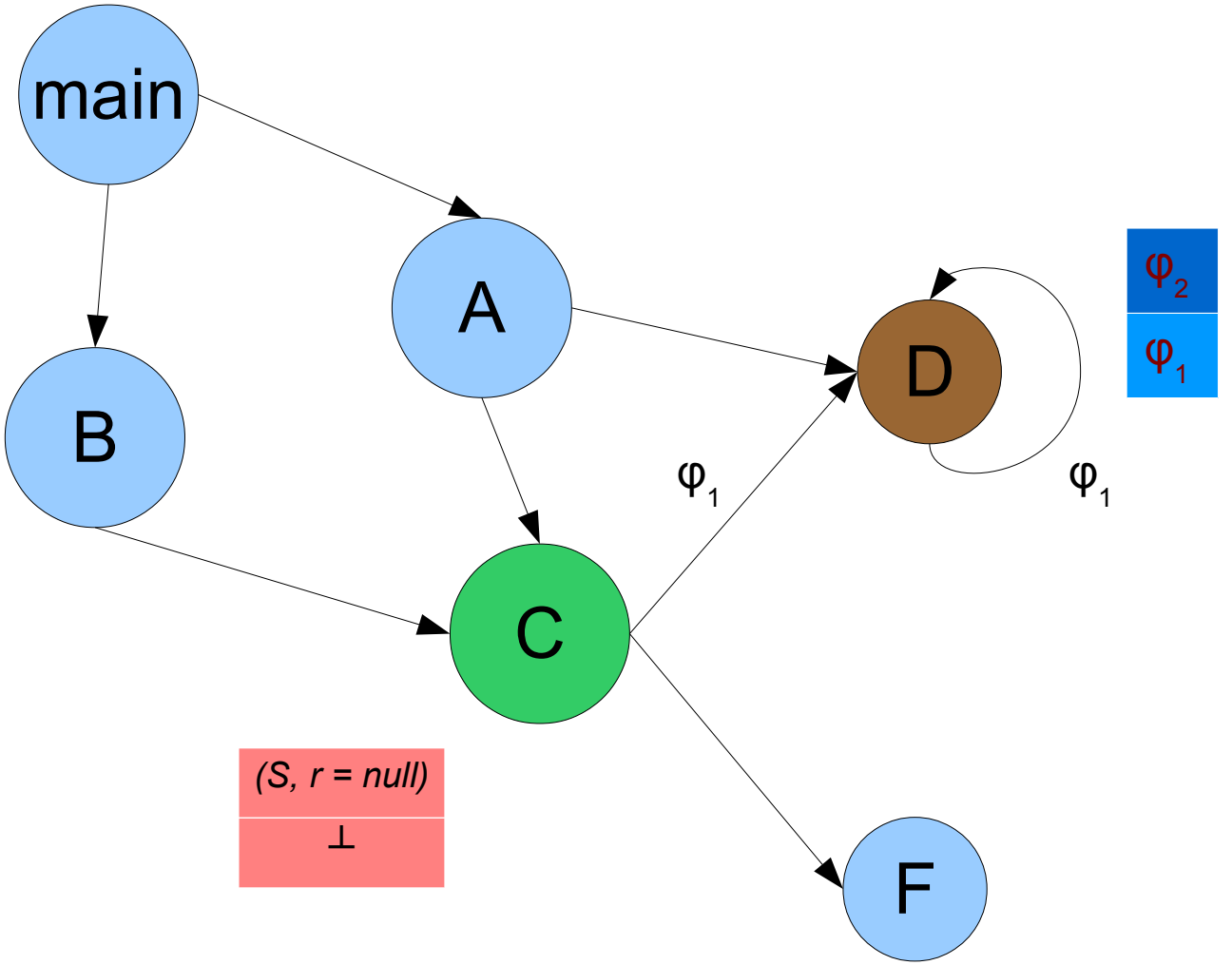
Handling Recursion



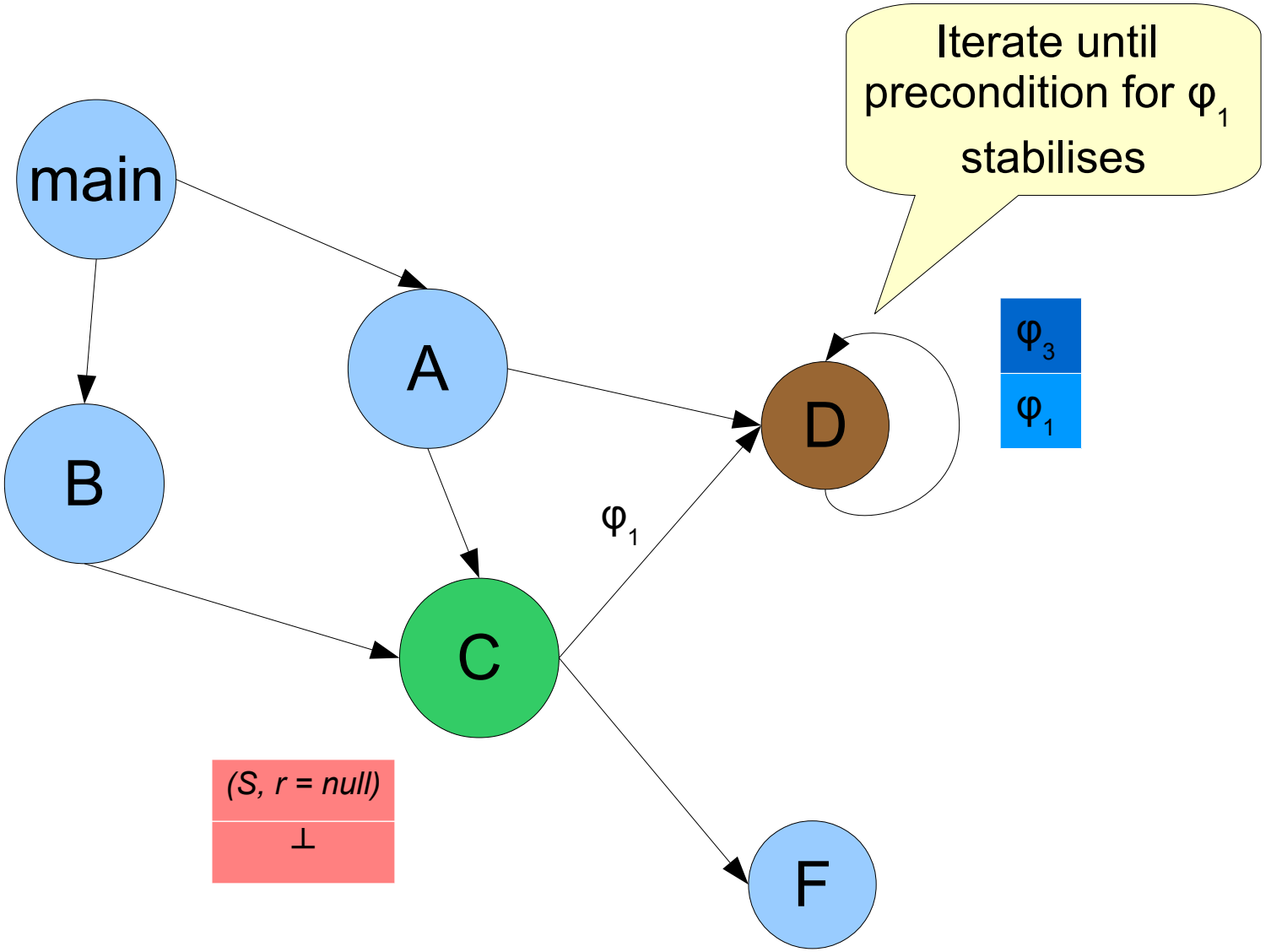
Handling Recursion



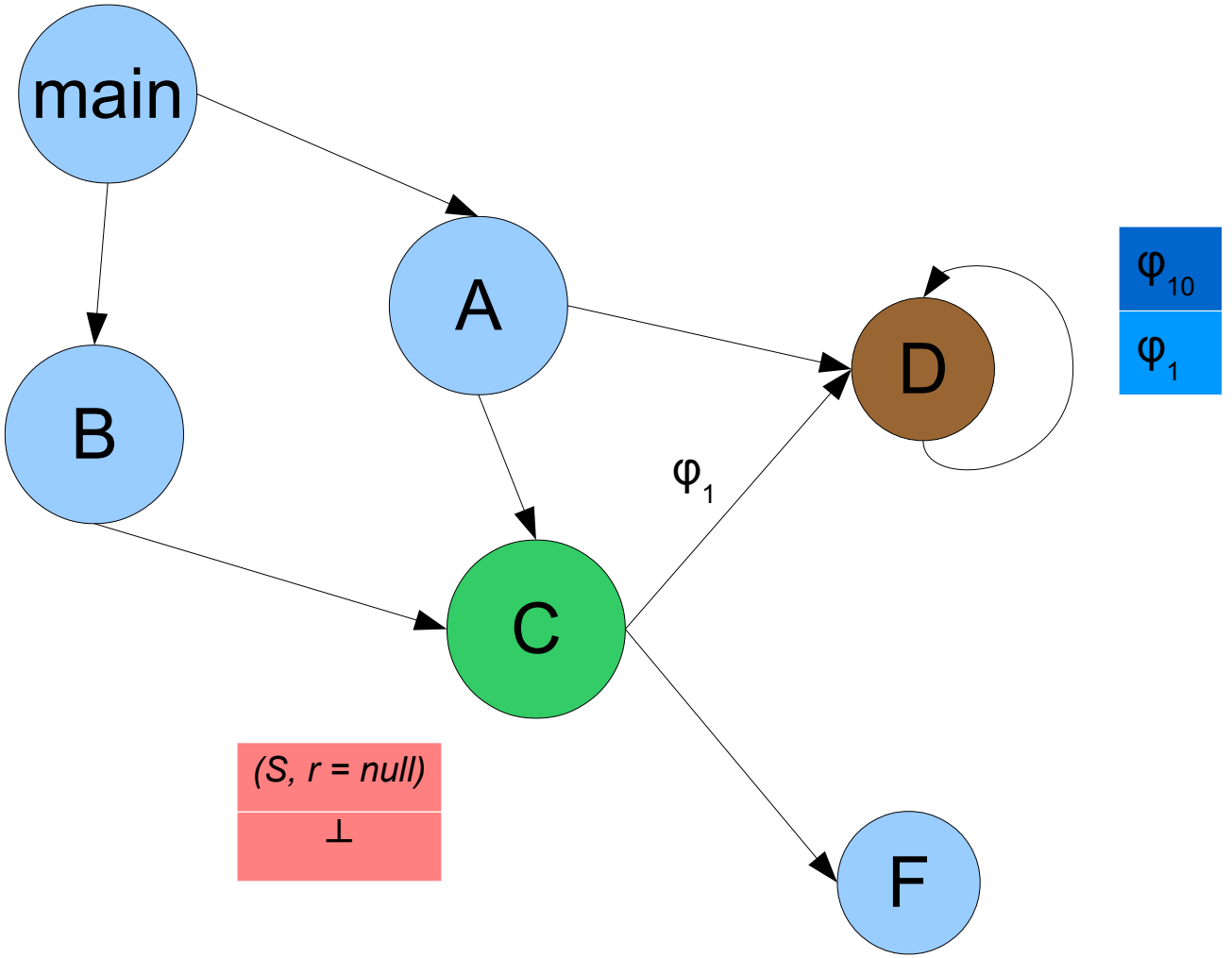
Handling Recursion



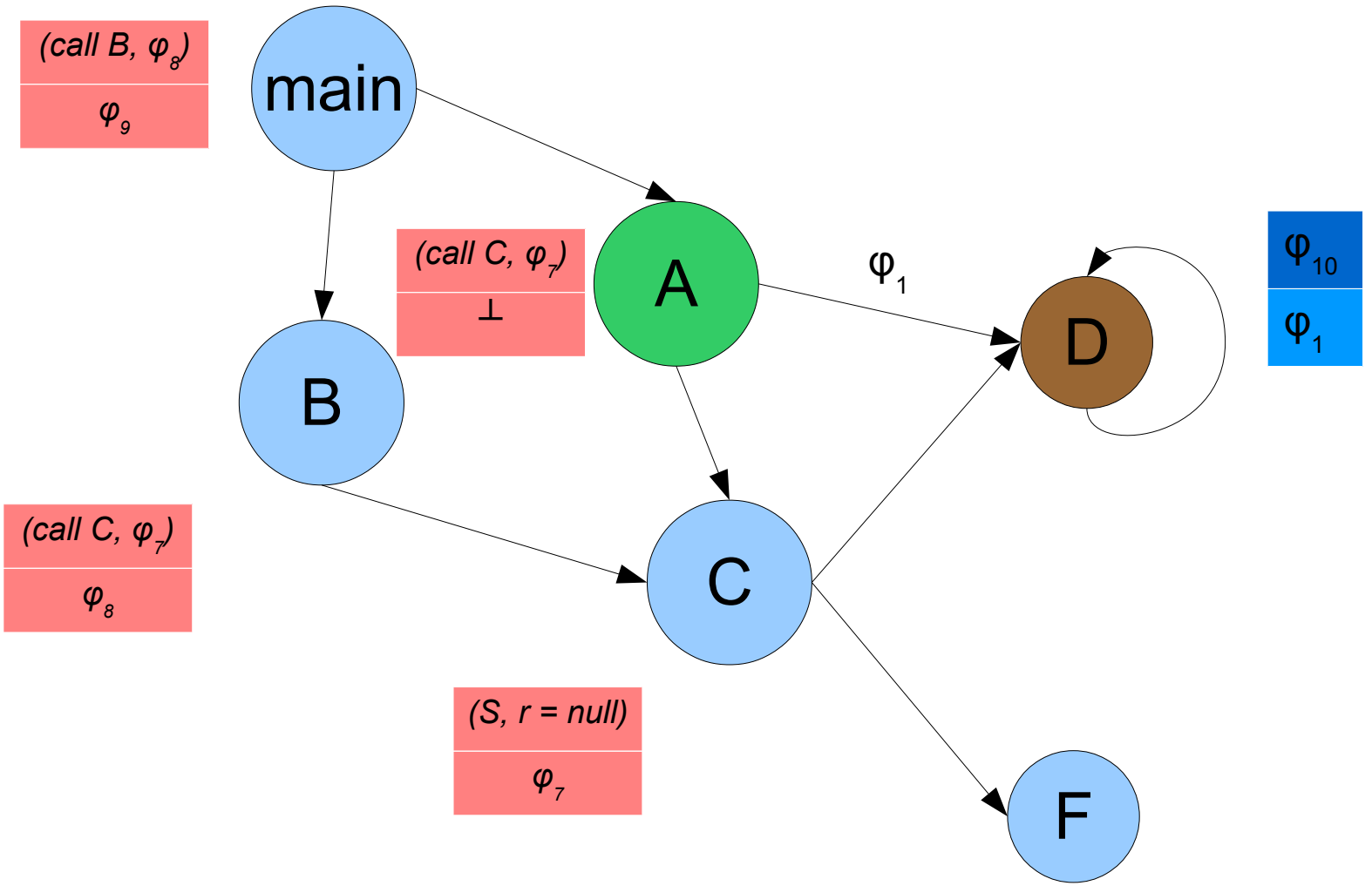
Handling Recursion



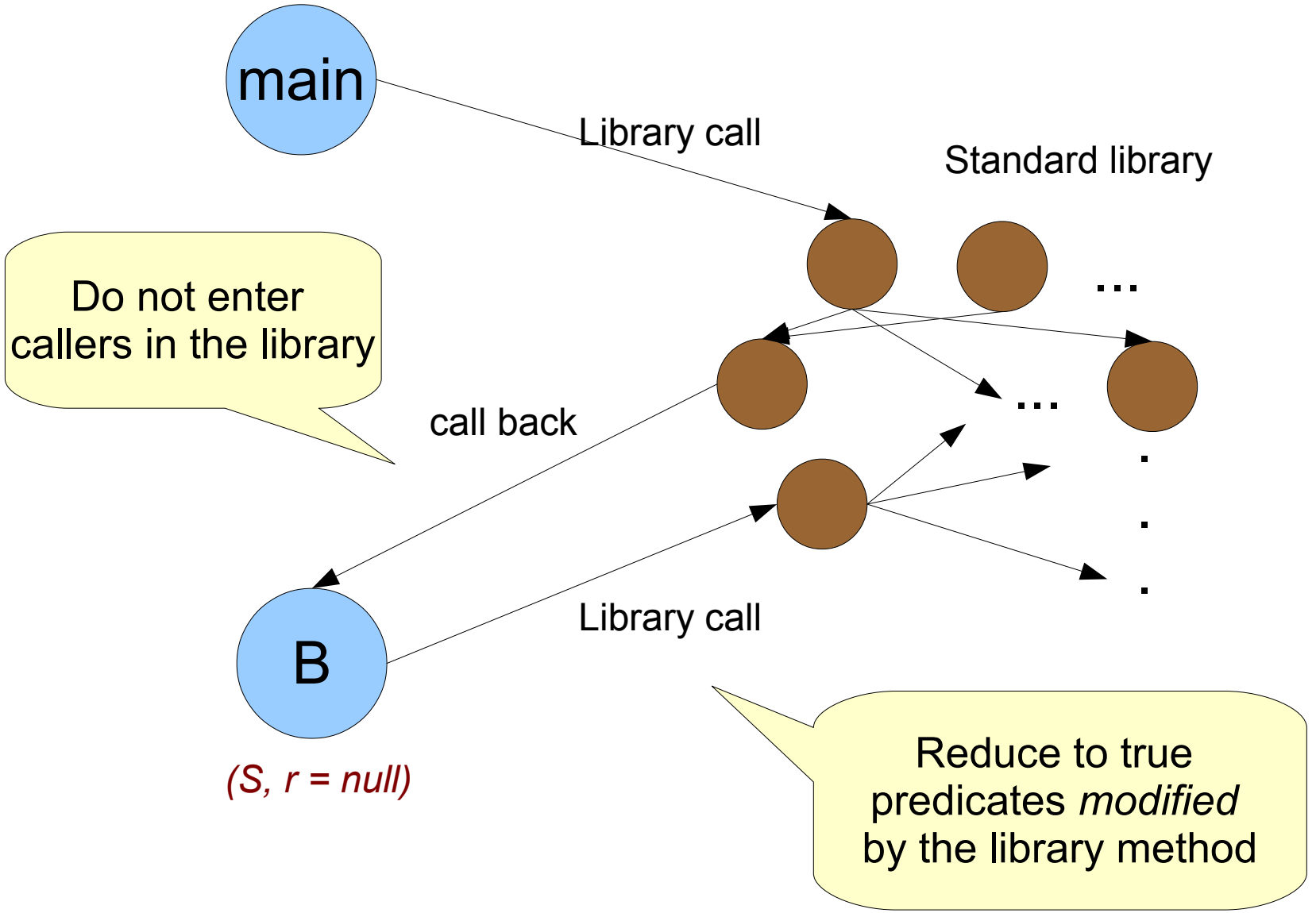
Handling Recursion



Handling Recursion



Challenges: Calls to/from standard library



Challenges: Explosion of formulas

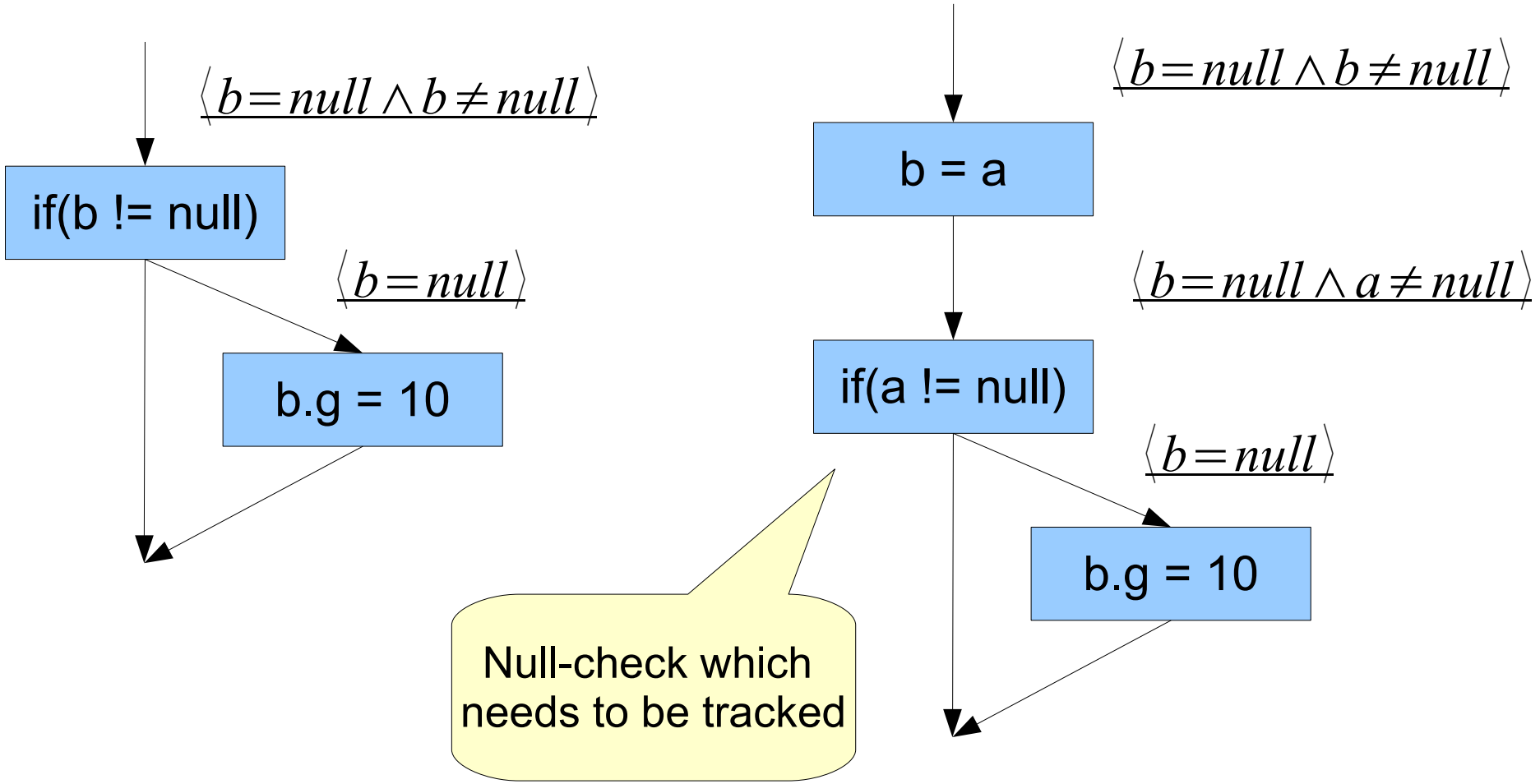
- Formula sizes can increase due to
  - Put-field statements (*aliasing predicates*)
  - Branch statements

- Put-field statements
  - We use an inexpensive alias analysis to invalidate alias predicates
  - $(ap_1 = ap_2) \rightarrow false$ if the access-paths are not *may-aliases*

Limiting Path-sensitivity

- Tracking all branch conditions without bounds will not scale
- Can we do better than arbitrarily bounding the disjunct sizes ?
- For null-dereference verification, **null-checks** are a good target !

Targeting null-checks



Limited path-sensitivity

- Track “AP op null” branches where AP aliases with AP in the root predicate

- **Ageing:**
  - Drop predicates propagated beyond a threshold
  - Track only k recent branch conditions

- **Parameterizable**: Predicate age, disjunct sizes and branches to be tracked can be configured.

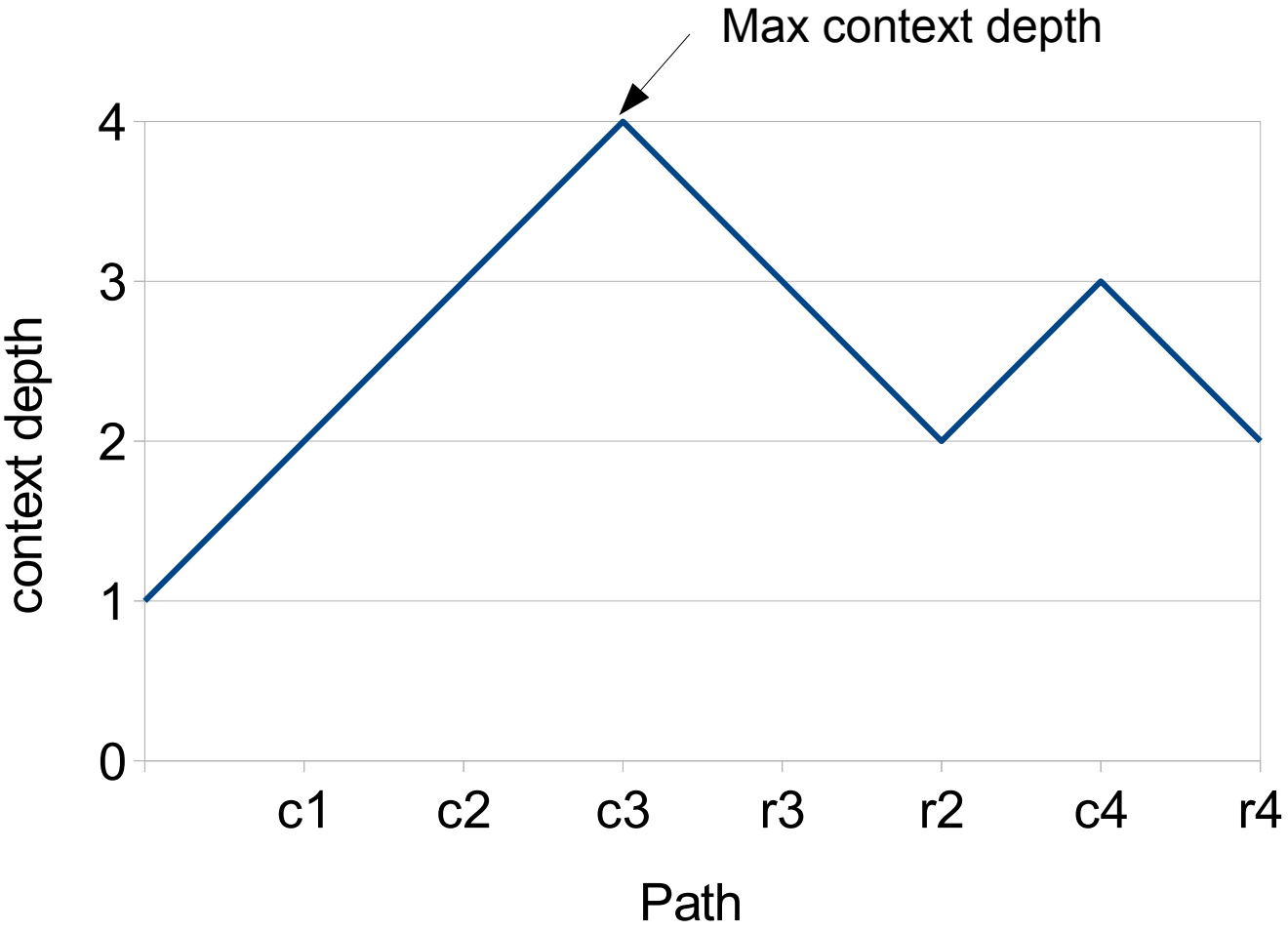
Results

Benchmark	Bytecode	Derefs	% deref verified
jlex	25K	2.5K	96.3%
javacup	29K	2.8K	78.7%
bcel	86K	10.1K	88.3%
jbidwatcher	105K	9.6K	84.7%
sablecc	157K	14K	84.9%
ourtunes	127K	16.4K	91.5%
proguard	185K	17.7K	84.4%
antlr	251K	17.4K	76.8%
freecol	260K	24K	70.9%
l2j	373K	36.8K	85.3%

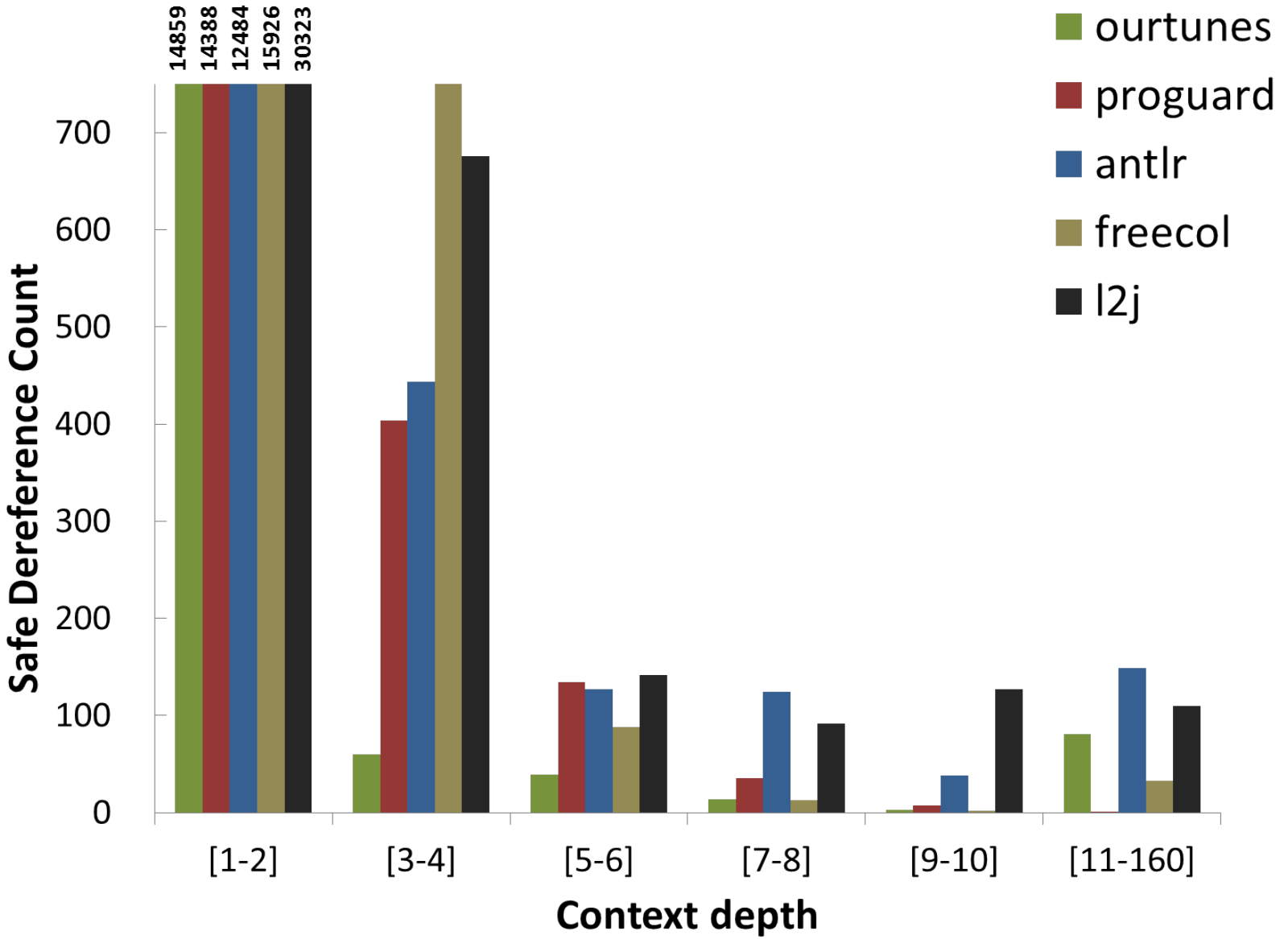
In all programs (except antlr), 93% of the derefs took < 250ms

Complexity of verified dereferences

- **Context Depth**



Max context depth Vs safe derefs

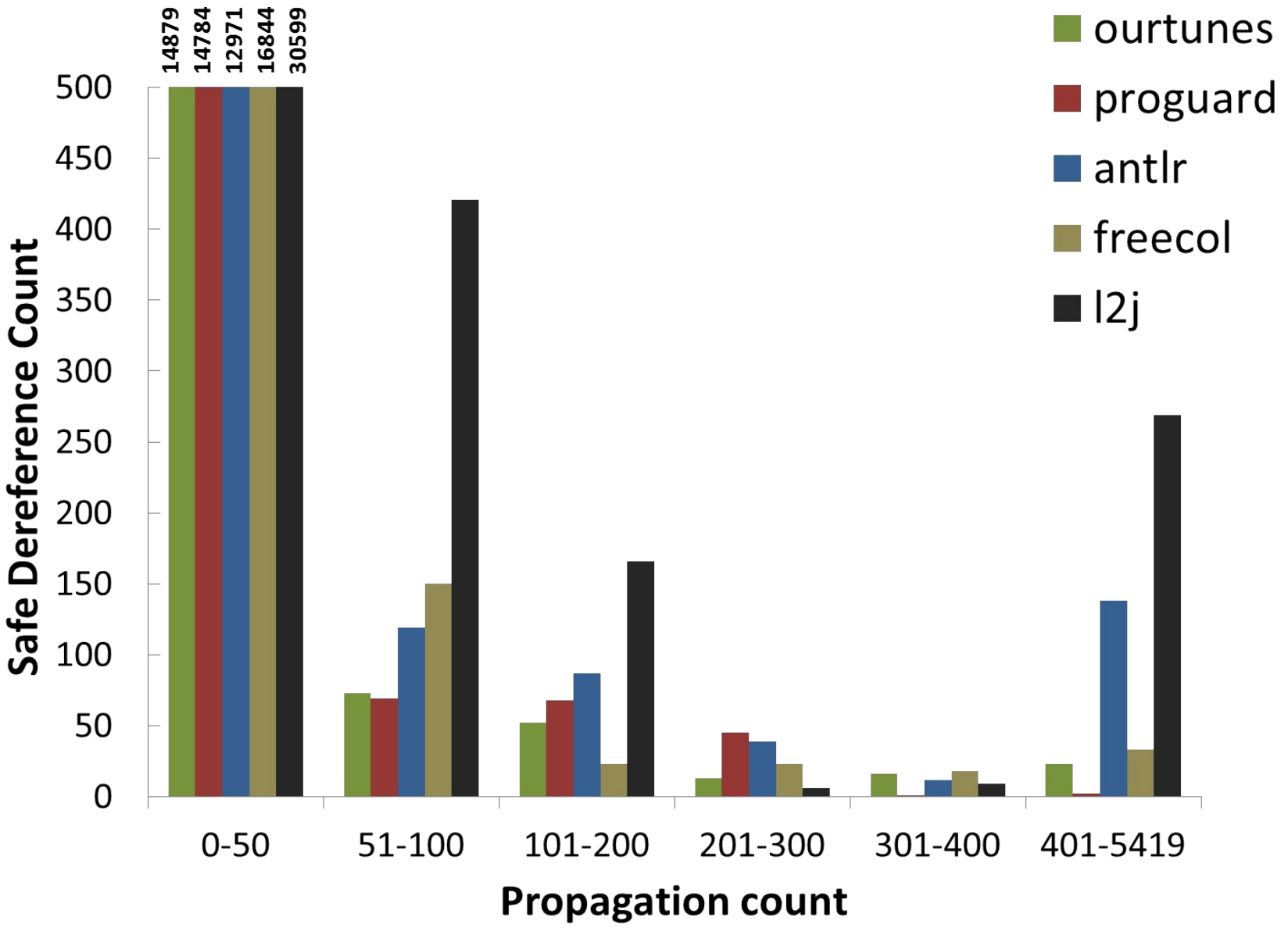


Example with high context depth

```
class L2Object {
    ObjectPosition _position
    public L2Object(){
        InitPosition();
    }
    public void initPosition(){
        SetObjectPosition(
            new CharPosition(this))
    }
    public void setObjectPosition(...) {
        _position = value
    }
    public ObjectPosition getPosition() {
        return _position;
    }
}
```

```
class L2Character extends L2Object {
    public L2Character() {
        super()
    }
}
class L2AirShipInstance extends Character{
    public L2AirShipInstance() {
        super()
    }
}
public void L2AirShipInstanceParseLine() {
    airship = new L2AirShipInstance(..);
    t = airship.getPosition()
    t.setHeading();
}
```

Max propagation count Vs safe derefs

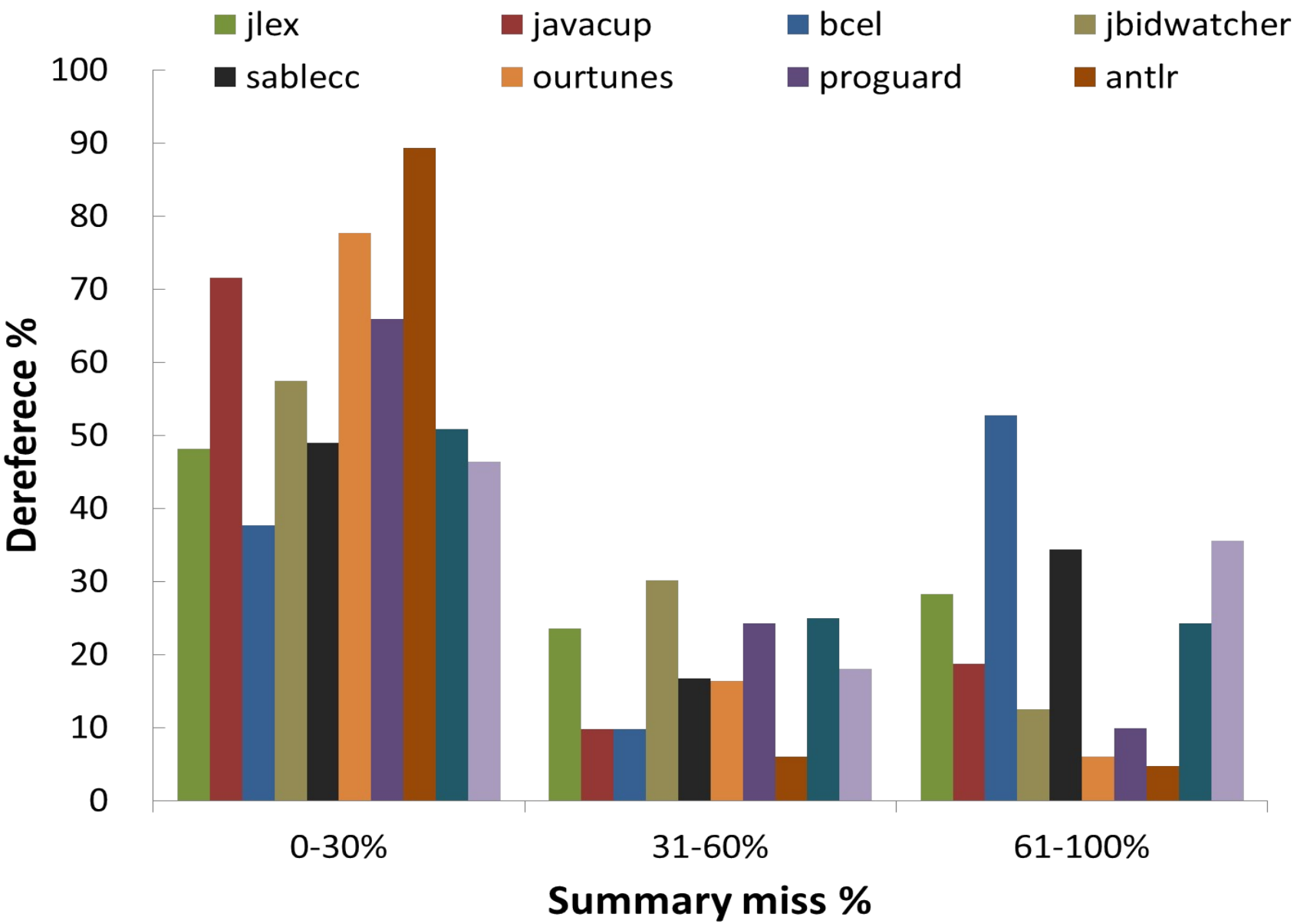


Evaluation of limited path-sensitivity

Benchmark	Ltd path sensitivity		No path sensitivity		K-recent branches	
bcel	1184	63s	2501	97s	1119	8123s
javacup	607	19s	1036	29s	463	881s
jlex	93	13s	296	33s	105	148s

Did not scale to the rest

Summary miss percentage



Related Work

- SALSA
  - Non demand-driven analysis
  - Performs limited scope analysis for scalability.
- Findbugs
  - Bug finding tool based on a set of heuristics
- Xylem
  - Uses a similar WP based approach
  - Bug finding tool (has false negatives/positives)

Related Work

- Snugglebug
  - Under-approximates WP_1 (computes a condition stronger than WP_1)
  - Can prove presence of bugs (not its absence)

- ESC/Java
  - Uses pre/post, loop-invariant annotations for computing WP

Conclusion

- Our evaluations on real Java programs reveal that
  - Deep inter-procedural analysis is important
  - Complex interactions with libraries (esp. with GUI, DB libs) present a huge challenge
  - Complications arise due to virtual method dispatch and exceptions
  - Our design trade-offs result in an highly responsive analysis with reasonable precision
  - Ideal for use in desktop development environments