

Automatic Inference of Ranking Functions by Abstract Interpretation

Caterina Urban

**MPRI 2-6: Abstract Interpretation,
Application to Verification and Static Analysis**

15th October 2014

The cost of software failure

- **Patriot MIM-104** failure, 25 February 1991
(death of 28 soldiers¹)
- **Ariane 5** failure, 4 June 1996
(cost estimated at more than 370 000 000 US\$²)
- **Toyota** electronic throttle control system failure, 2005
(at least 89 death³)
- **Heartbleed** bug in OpenSSL, April 2014
- economic cost of software bugs is tremendous⁴

¹ R. Skeel. "Roundoff Error and the Patriot Missile". SIAM News, volume 25, nr 4.

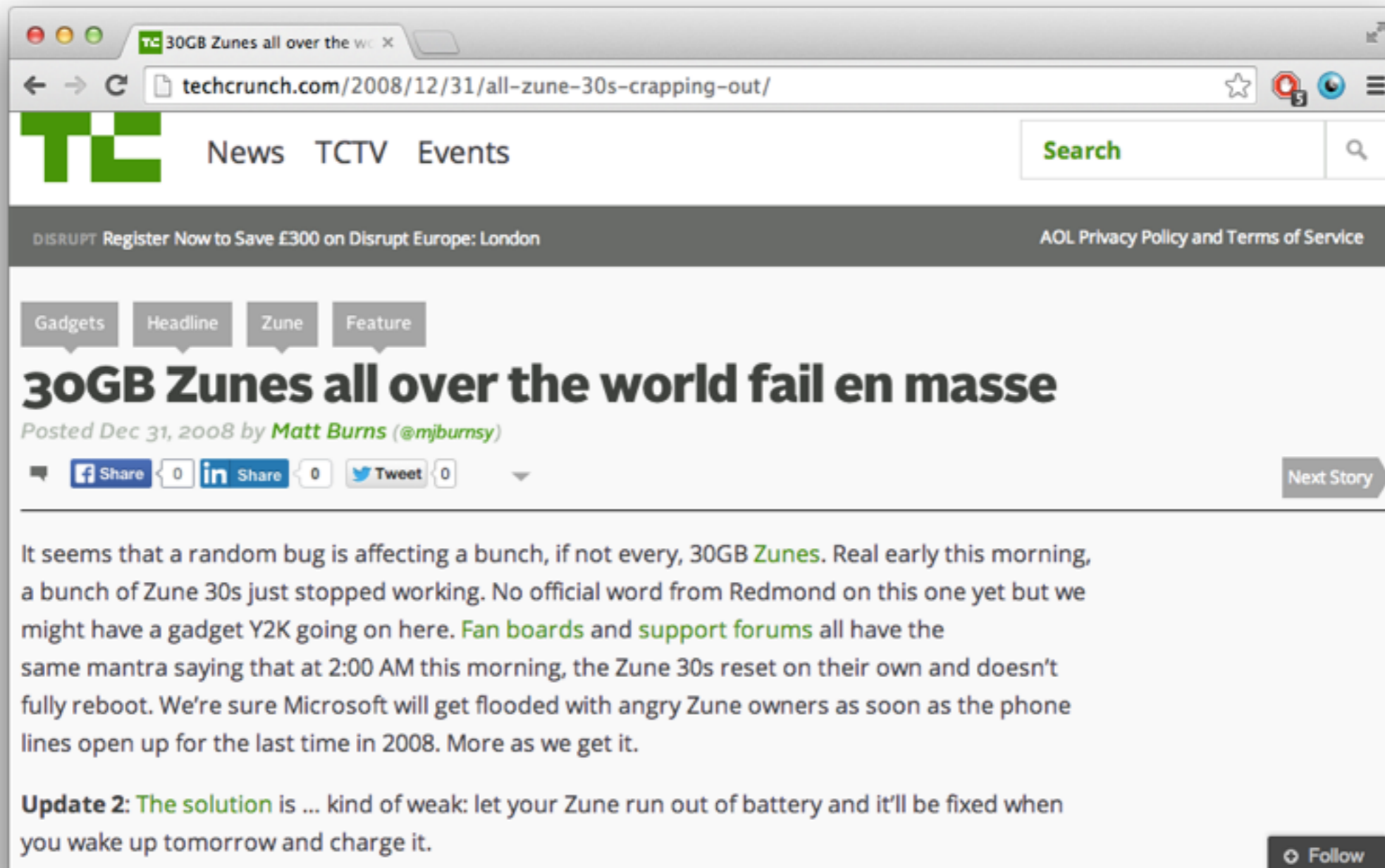
² M. Dowson. "The Ariane 5 Software Failure". Software Engineering Notes 22 (2): 84, March 1997.

³ CBSNews. Toyota "Unintended Acceleration" Has Killed 89. 20 March 2014.

⁴ NIST. Software errors cost U.S. economy \$59.5 billion annually. Tech. report, NIST Planning Report, 2002.

The Zune Bug

December 31st, 2008



- failure due to **non-termination**

<http://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/>

The Zune Bug



- failure due to

<http://techcrunch.com>

A screenshot of a TechCrunch article titled "Zune bug explained in detail" posted on Dec 31, 2008 by Devin Coldewey. The article includes social sharing buttons for Facebook (10 shares), LinkedIn (0 shares), and Twitter (2 tweets). The main text discusses the Zune bug and includes a code snippet. The code snippet is as follows:

```
year = ORIGINYEAR; /* = 1980 */  
  
while (days > 365)  
{  
    if (IsLeapYear(year))  
    {  
        if (days > 366)  
        {  
            days -= 366;  
            year += 1;  
        }  
    }  
    else  
    {  
        days -= 365;  
        year += 1;  
    }  
}
```

The article continues with the text: "You can see the details here, but the important bit is that today, the day count is 366. As you" and a "Follow" button.

The Zune Bug



- failure due to

<http://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/>

A screenshot of a TechCrunch article titled "Zune bug explained in detail" posted on Dec 31, 2008, by Devin Coldewey. The article explains the Zune bug and includes a code snippet. A red circle highlights the code block in the original image.

```
year = ORIGINYEAR; /* = 1980 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

You can see the details here, but the important bit is that today, the day count is 366. As you

Follow

Liveness properties

Idea: **liveness property** $P \in \mathcal{P}(\Sigma^\infty)$

Liveness properties model that “something good eventually occurs”

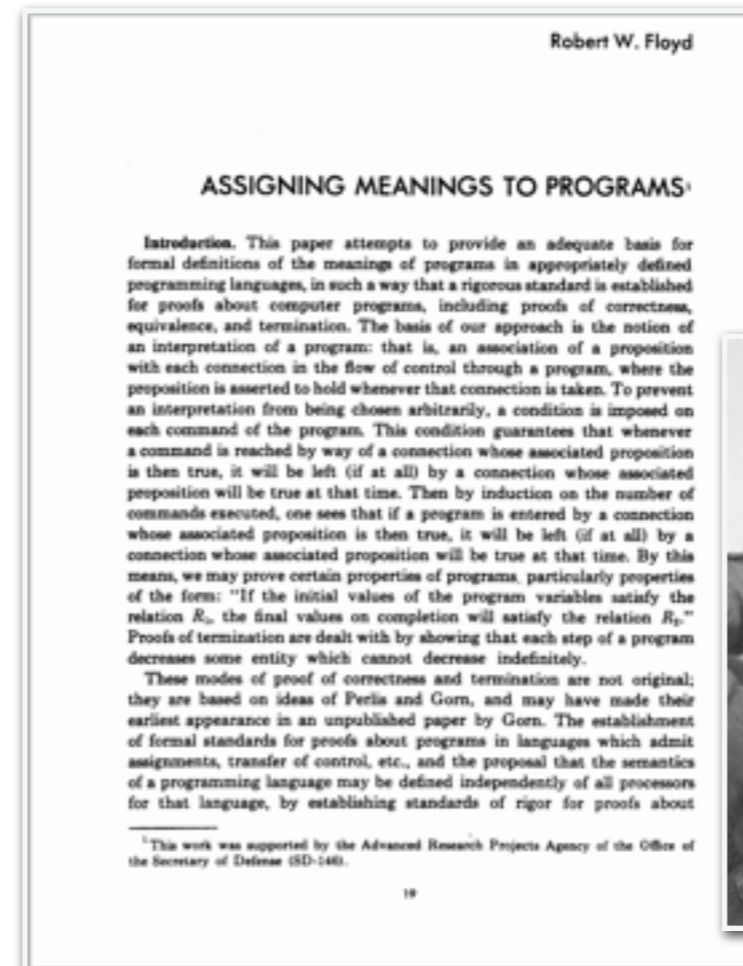
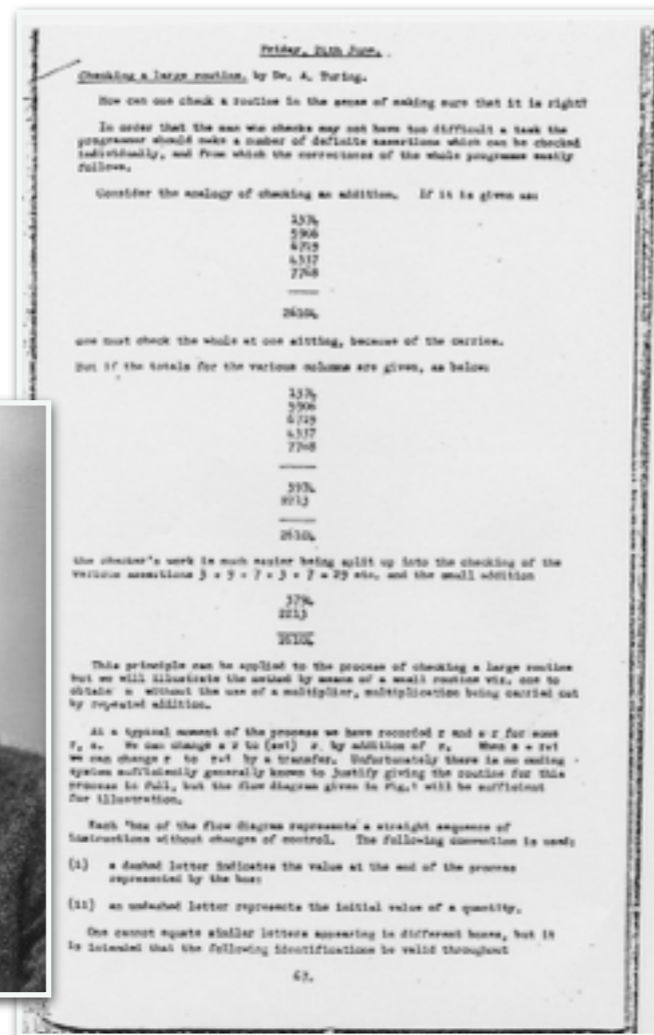
- P cannot be proved by testing
(if nothing good happens in a prefix execution,
it can still happen in the rest of the execution)
- disproving P requires exhibiting an infinite execution not in P

Examples:

- **termination:** $P \stackrel{\text{def}}{=} \Sigma^*$,
- **inevitability:** $P \stackrel{\text{def}}{=} \Sigma^* \cdot a \cdot \Sigma^\infty$,
(a eventually occurs in all executions)
- state properties are **not** liveness properties.

Ranking Functions

- functions that strictly **decrease** at each program step...
- ...and that are **bounded** from below



Turing - *Checking a Large Routine* (1949)
Floyd - *Assigning Meanings To Programs* (1967)

Proving liveness properties

Variance proof method: (informal definition)

Find a **decreasing quantity** until something good happens.

Example: termination proof

- find $f : \Sigma \rightarrow \mathcal{S}$ where $(\mathcal{S}, \sqsubseteq)$ is **well-ordered**;
(f is called a “ranking function”)
- $\sigma \in \mathcal{B} \implies f = \min \mathcal{S}$;
- $\sigma \rightarrow \sigma' \implies f(\sigma') \sqsubseteq f(\sigma)$.

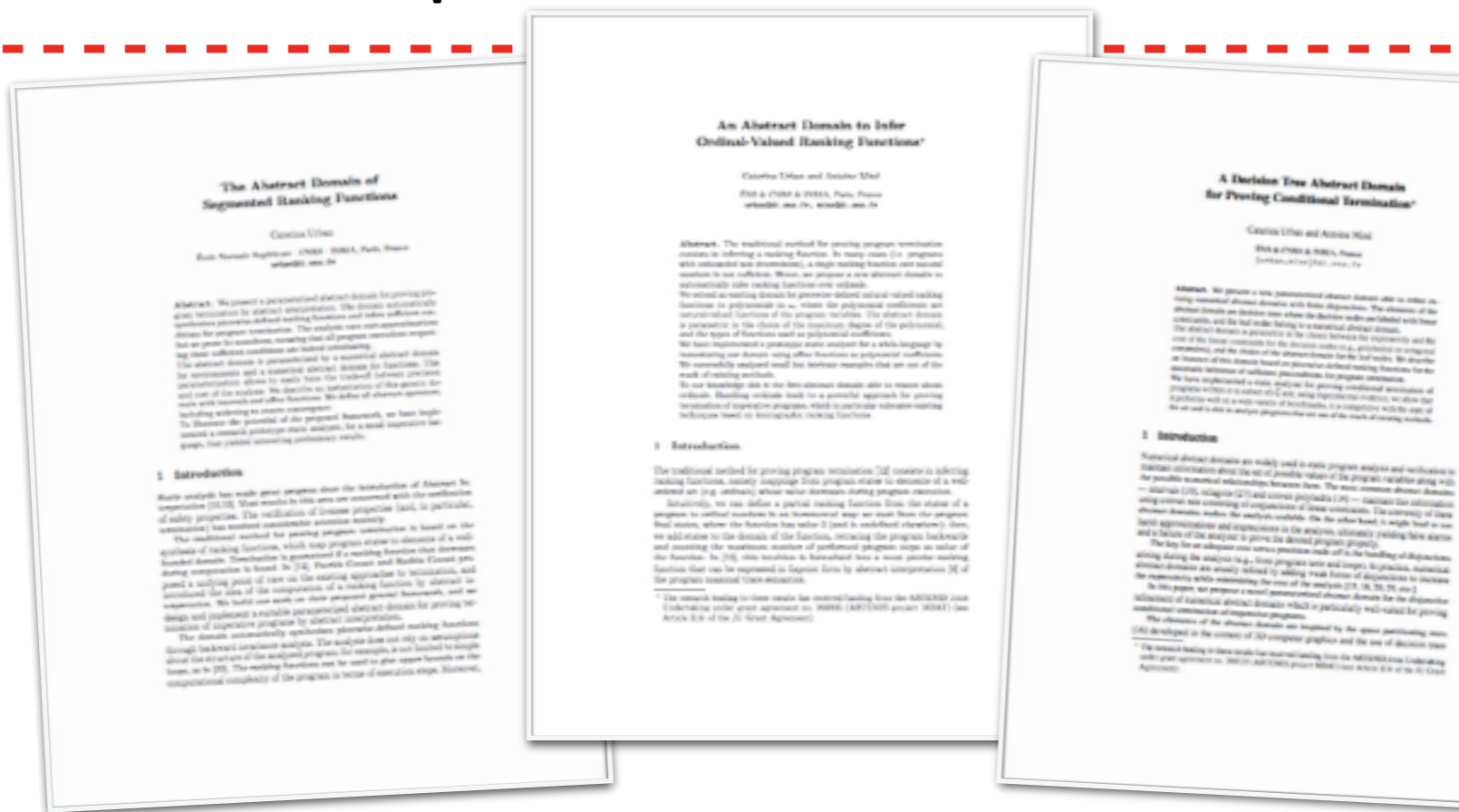
(f counts the number of steps remaining before termination)

- idea: inference of ranking functions by abstract interpretation



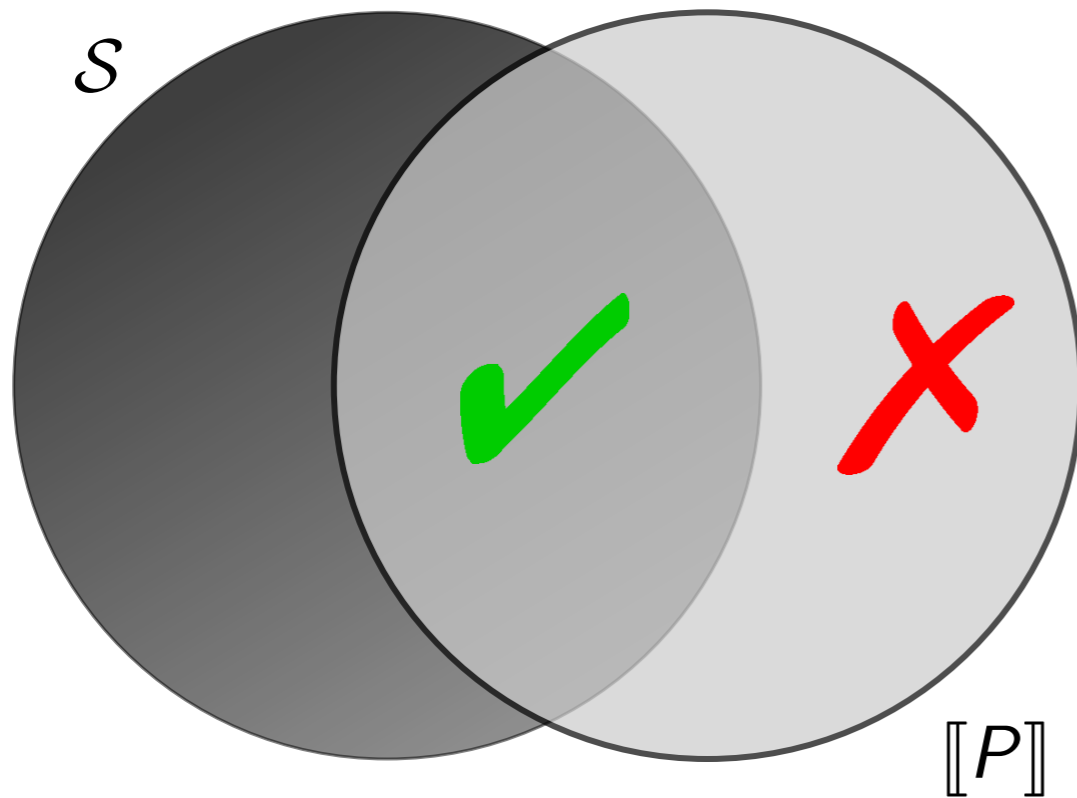
- **idea**: inference of ranking functions by **abstract interpretation**

- family of **abstract domains** for program termination
 - **piecewise-defined ranking functions**
 - **backward analysis**
 - **sufficient preconditions** for termination



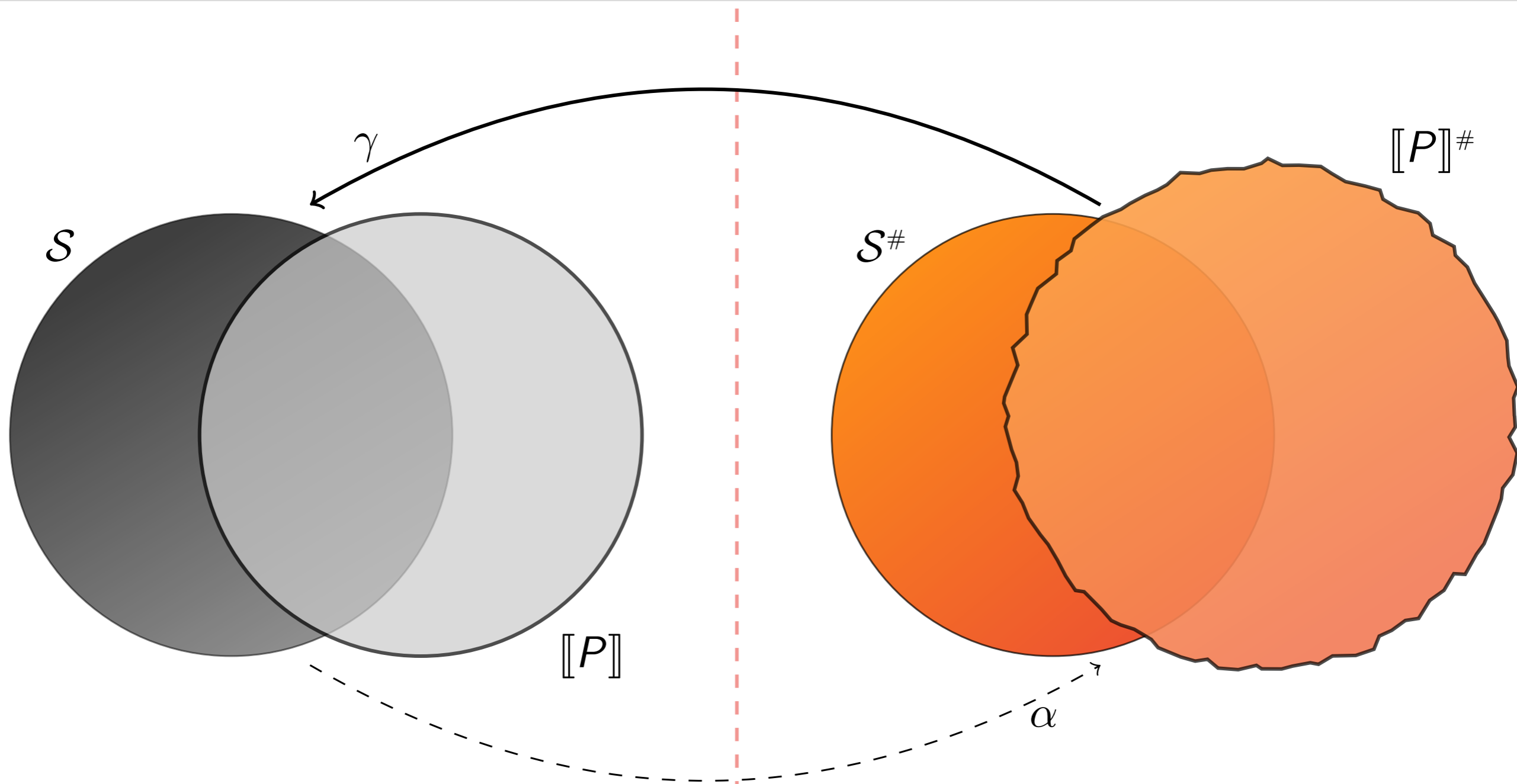
Urban - *The Abstract Domain of Segmented Ranking Functions* (SAS 2013)
Urban&Miné - *An Abstract Domain to Infer Ordinal-Valued Ranking Functions* (ESOP 2014)
Urban&Miné - *A Decision Tree Abstract Domain for Proving Conditional Termination* (SAS 2014)

Abstract Interpretation



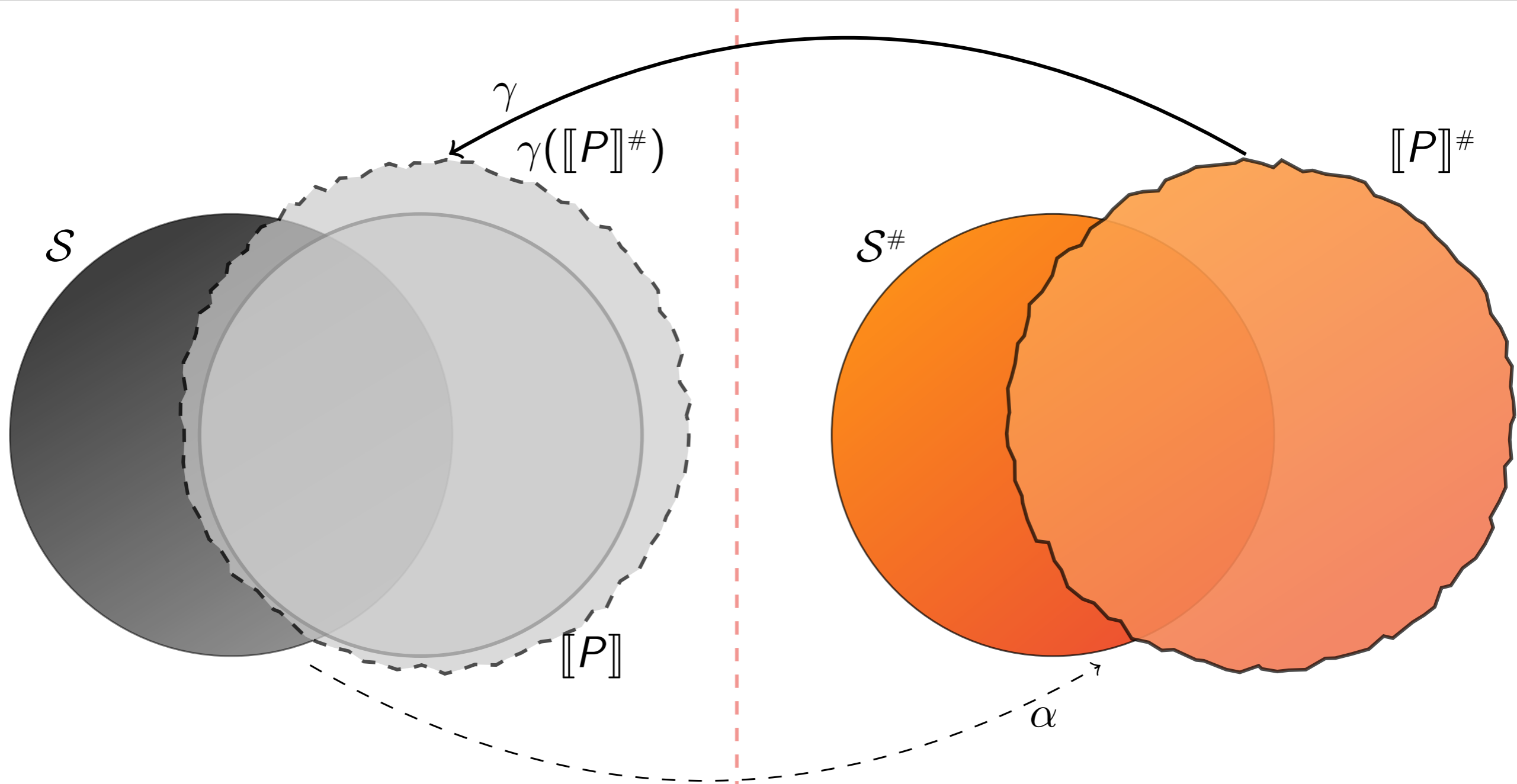
Cousot&Cousot - *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints* (POPL 1977)

Abstract Interpretation

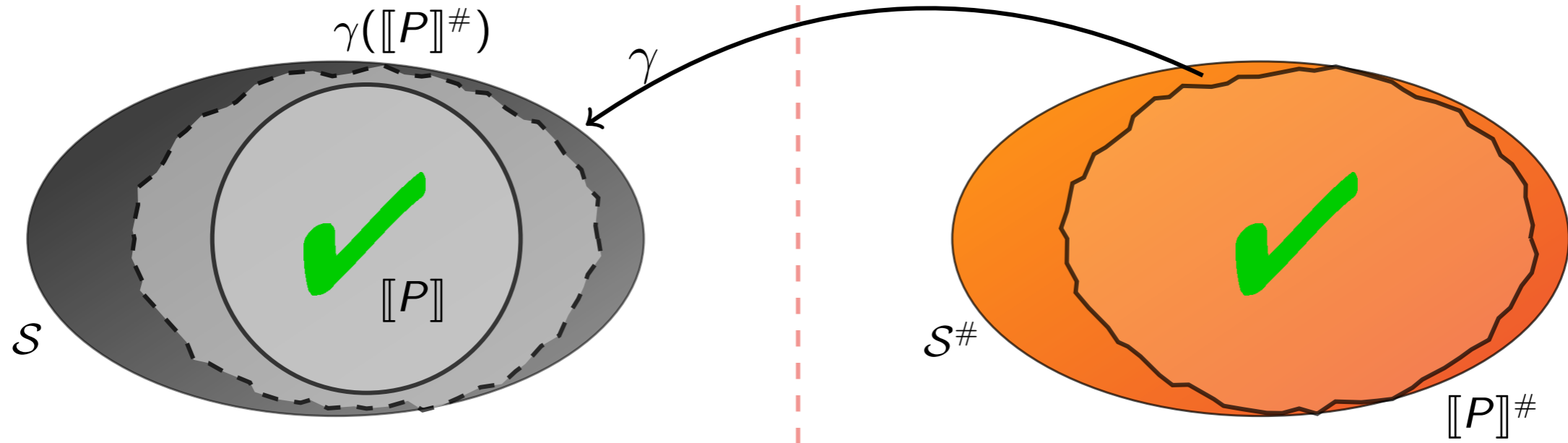


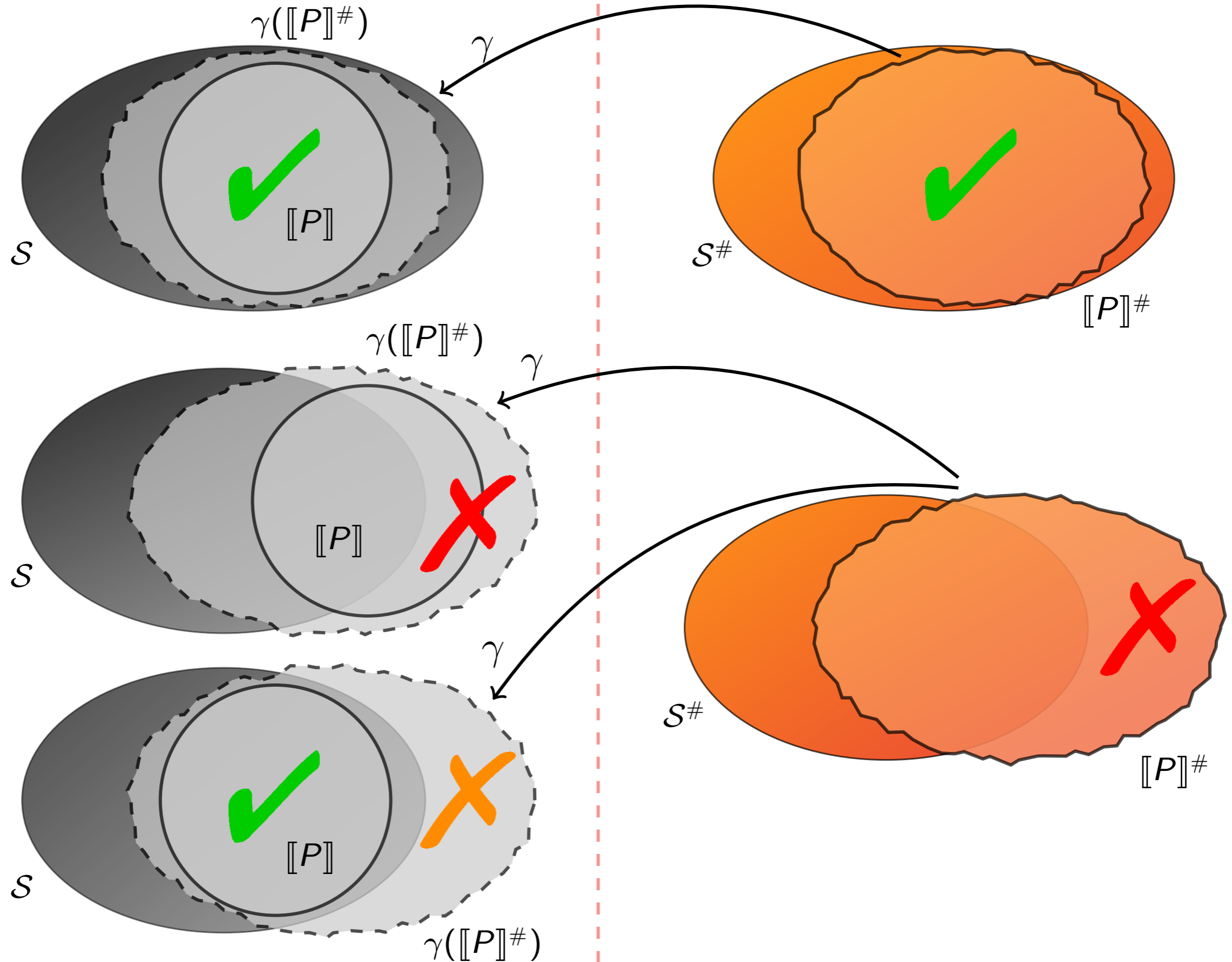
Cousot&Cousot - *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints* (POPL 1977)

Abstract Interpretation



Cousot&Cousot - *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints* (POPL 1977)





Termination Semantics

An Abstract Interpretation Framework for Termination

Patrick Cousot

CNRS, Ecole Normale Supérieure, and INRIA, France
Cousot@enscm.fr, NNT@LSA
cousot@enscm.fr, pcousot@cs.nyu.edu

Rachid Cousot

CNRS, Ecole Normale Supérieure, and INRIA, France
racha@enscm.fr

Abstract

Proof, verification and analysis methods for termination all rely on two induction principles: (1) a variant function or induction set decreasing program towards the end and (2) some form of induction on the program structure.

The abstract interpretation design principle is first illustrated for the design of new forward and backward proof, verification and analysis methods for safety. The safety-critical semantics defining the strongest safety property of programs is first expressed as a constructive logical basis. Safety proof and checking/verification methods then approximate safety by logical abstraction. Static analysis of abstract safety properties such as invariants are automatically designed by logical abstraction (or approximation) to (automatically) infer safety properties. As for new static design principles that rely on termination or that the existing approaches are tailored and largely not amenable with each other.

In (2), we show that the design principle applies equally well to forward and backward induction. The fact-based termination collecting semantics is given a logical definition. Its abstraction yields a logical definition of the least variant function. By further abstraction of this least variant function, we derive the Floyd-style termination proof method as well as new static analysis methods to efficiently compute approximate times of this least variant function.

In (3), we introduce a generalization of the classical notion of standard induction (as found in Floyd [19]) into a semantic structural induction based on the new semantic concept of inductive trace-covering condition (trace) by expressing a new logic for terminating program properties. Its abstractions allow the generation of forward proof, verification and static analysis methods by induction on both program structure, control, and data. Examples of particular interest include Floyd's handling of loop exit points as well as forward loops, Rasmussen's invariants over time and termination proof method, and Podelski's Floyd-style verification technique.

Categories and Subject Descriptors: D.1.4 [Software Program Verification]; D.2.1 [Formal Definitions and Theory]; D.2.2 [Operations and Analysis]; D.2.3 [Verification and Formal Reasoning]

General Terms: Languages, Reliability, Security, Theory, Verification.

Keywords: Abstract Interpretation, Induction, Proof, Safety, Static Analysis, Variant Function, Verification, Termination.

1. Introduction

Floyd-style program proof methods for invariance and termination [19, 18, 28] have inspired direct, sound static analysis methods.

The static termination analysis by abstract interpretation [24, 22], a key step is to express the strongest invariant as a logical and new to approximate this invariant invariance to automatically infer an abstract inductive invariant using the constructive Floyd approximation method.

For static termination analysis, the discovery of variant functions is often decidable in limited cases [30] or else is based on the Floyd-style idea of variant functions into well-founded sets.

*Work supported in part by the CNRS-SNP Foundation in Grenoble and INRIA.

Permission is made digital or hard copies of all or part of this work for personal or classroom use, provided that the copies are not made for commercial or promotional purposes, for advertising or promotional purposes, for creating new collective works, for resale, or for redistribution.

© 2012 ACM 978-1-4558-6242-1/12/000000-0000

abstractly observing quantities that strictly decrease within loops while increasing, lower-bounded, or stable. So these termination analysis methods inherently reduce to a relational semantics analysis hence can reuse classical static analysis methods.

The abstract interpretation design principle is instantiated with suitable abstractions for safety and termination analysis, proof, and checking/verification (under potential invariance or definite termination for non-deterministic queries).

The first main idea for termination is that there exists a most precise variant function that can be expressed by logical forms by abstract interpretation of a termination collecting semantics that abstracting the program operational trace semantics. This yields new static analysis methods automatically inferring abstractions of the variant function by the constructive Floyd approximation method of abstract interpretation.

The second main idea introduced in this paper both for safety and termination is that of semantic structural induction, including termination proofs, and trace segment covers and their abstractions. These segments are stack generalizations of Floyd's abstracted traces which have been used traditionally in program verification proofs (for example, the invariant variants used in [22] as binary relation abstractions of the set of trace segments). Examples include structural induction on the program traces (including loop invariants) in Floyd [19], induction on data, in Rasmussen [25], the covering of the weakest liberal closure by well-founded relations, in Podelski's Floyd-style [26], their combination and generalizations.

2. Floyd-style, Floyd induction, abstraction, and approximation

We express semantics as a sequence of steps $f: A \rightarrow A$ i.e. elements $a \in A$ such that $a = f(a)$. We let f^0, f^1, \dots for the least fixpoint of $f: A \rightarrow A$ on the power $\mathcal{P}(A)$, i.e. greatest fixpoint of $f: \mathcal{P}(A) \rightarrow \mathcal{P}(A)$, if any. The dual notion is that of greatest fixpoint $gfp(f)$. We write $fp(f)$ if f is the infimum of A , and $fp(f)$ if the partial order \leq is clear from the context. By DeMorgan's Law (DML), $fp(f) = \neg(gfp(\neg \circ f \circ \neg))$, $fp(f) = \neg(gfp(\neg \circ f \circ \neg))$, $fp(f) = \neg(gfp(\neg \circ f \circ \neg))$, $fp(f) = \neg(gfp(\neg \circ f \circ \neg))$. The Floyd-style f^0, f^1, \dots is increasing but not continuous, transfinite iterations may have to be used [22].

f^0, f^1, \dots is increasing (also monotone, i.e. $f^0 \leq f^1 \leq f^2 \leq \dots$) and only if f is increasing (also monotone, i.e. $f^0 \leq f^1 \leq f^2 \leq \dots$).

f^0, f^1, \dots is complete (also c.l.s., i.e. f^0, f^1, \dots is a complete lattice) if and only if f is complete (also c.l.s., i.e. f^0, f^1, \dots is a complete lattice).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).

f^0, f^1, \dots is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.) if and only if f is a complete partial order (c.p.o., i.e. f^0, f^1, \dots is a c.p.o.).



Transition systems: definition

Language-neutral formalism to discuss about program semantics.

Transition system: (Σ, τ)

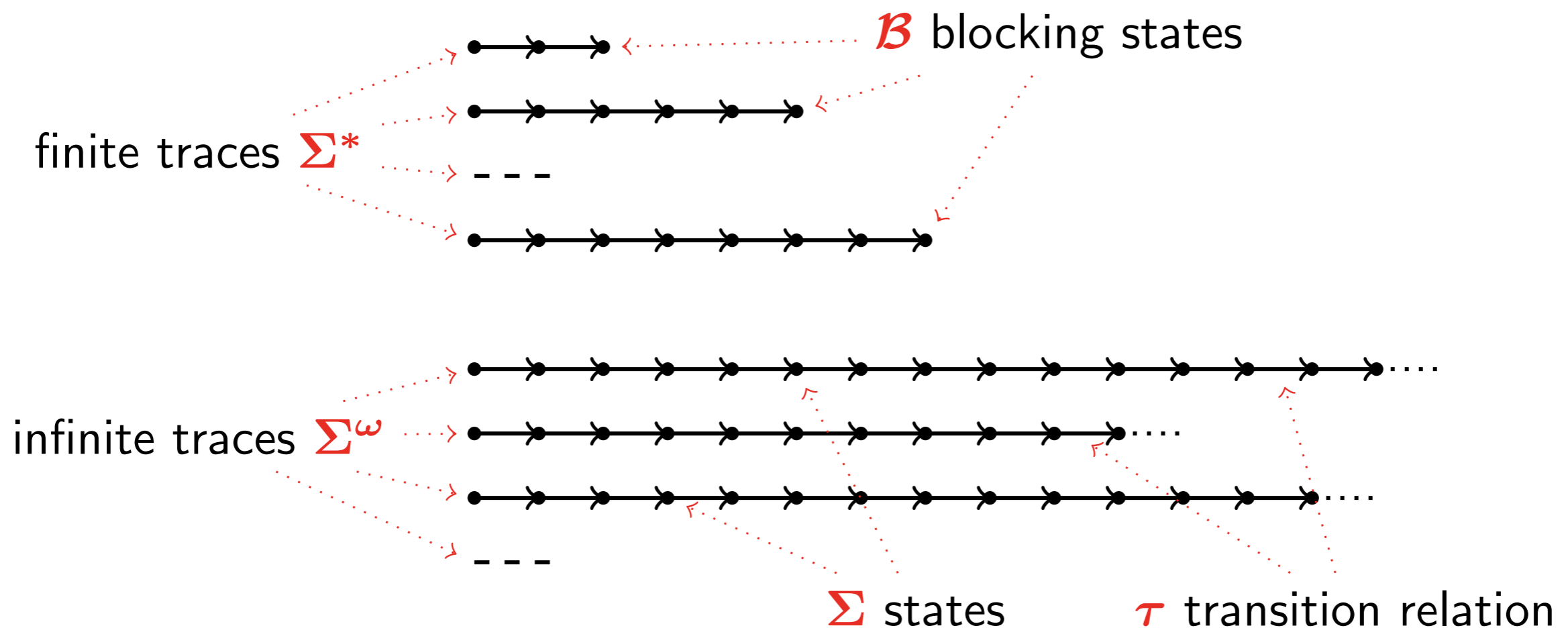
- set of states Σ ,
(memory states, λ -terms, configurations, etc., generally infinite)
- transition relation $\tau \subseteq \Sigma \times \Sigma$.

(Σ, τ) is a general form of small-step operational semantics.

$(\sigma, \sigma') \in \tau$ is noted $\sigma \rightarrow \sigma'$:

starting in state σ , after an execution step, we can go to state σ' .

program \mapsto **maximal trace semantics**



Least fixpoint formulation of maximal traces

Idea: To get a fixpoint formulation for whole \mathcal{M}_∞ ,
merge finite and infinite maximal trace fixpoint forms.

Fixpoint fusion

$\mathcal{M}_\infty \cap \Sigma^*$ is best defined on $(\Sigma^*, \subseteq, \cup, \cap, \emptyset, \Sigma^*)$.

$\mathcal{M}_\infty \cap \Sigma^\omega$ is best defined on $(\Sigma^\omega, \supseteq, \cap, \cup, \Sigma^\omega, \emptyset)$.

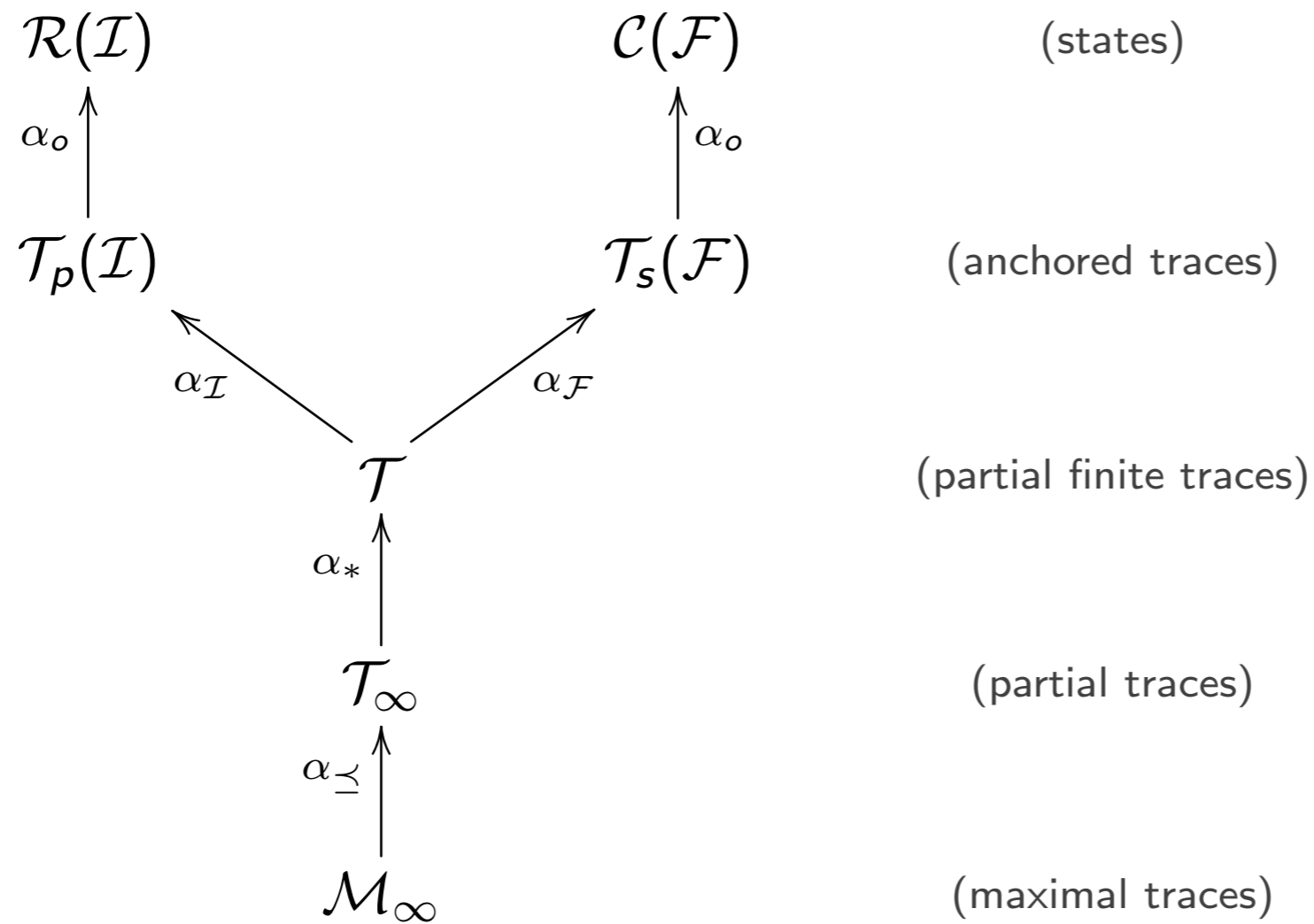
We mix them into a **new** complete lattice $(\Sigma^\infty, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$:

- $A \sqsubseteq B \stackrel{\text{def}}{\iff} (A \cap \Sigma^*) \subseteq (B \cap \Sigma^*) \wedge (A \cap \Sigma^\omega) \supseteq (B \cap \Sigma^\omega)$
- $A \sqcup B \stackrel{\text{def}}{=} ((A \cap \Sigma^*) \cup (B \cap \Sigma^*)) \cup ((A \cap \Sigma^\omega) \cap (B \cap \Sigma^\omega))$
- $A \sqcap B \stackrel{\text{def}}{=} ((A \cap \Sigma^*) \cap (B \cap \Sigma^*)) \cup ((A \cap \Sigma^\omega) \cup (B \cap \Sigma^\omega))$
- $\perp \stackrel{\text{def}}{=} \Sigma^\omega$
- $\top \stackrel{\text{def}}{=} \Sigma^*$

In this lattice, $\mathcal{M}_\infty = \text{lfp } F_s$ where $F_s(T) \stackrel{\text{def}}{=} \mathcal{B} \cup \tau \frown T$.

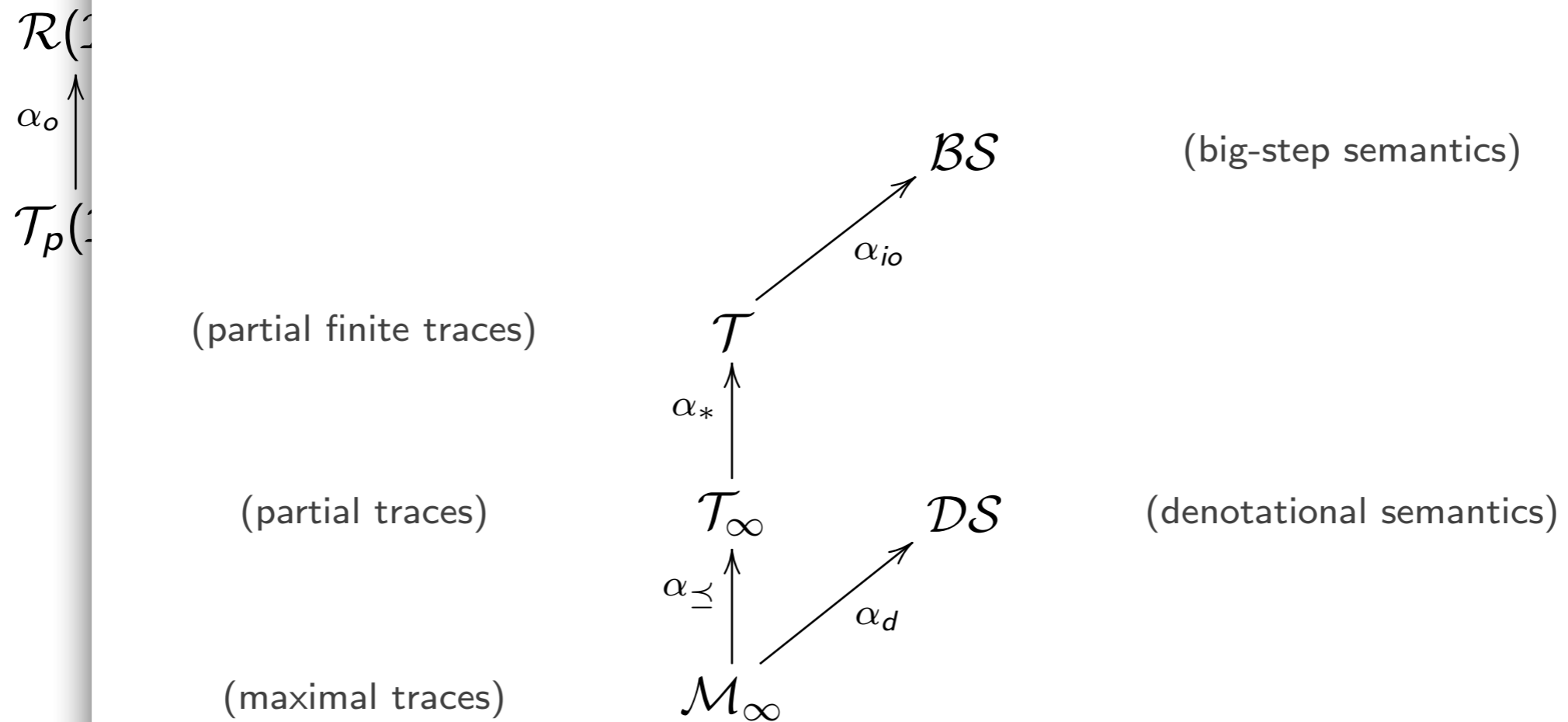
(proof on next slides)

(Partial) hierarchy of semantics



(Partial) hierarchy of semantics

Another part of the hierarchy of semantics



See [Cou82] for more semantics in this diagram.

program \mapsto maximal trace semantics \rightarrow **termination semantics**

$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_t \stackrel{\text{def}}{=} \text{lfp } F_t$$

idea = define a ranking function **counting the number of program steps** from the end of the program

$$F_t(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \mathcal{B} \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \\ \text{undefined} & \text{otherwise} \end{cases}$$

program \mapsto maximal trace semantics \rightarrow **termination semantics**

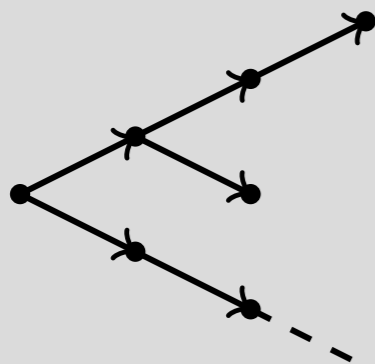
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_t \stackrel{\text{def}}{=} \text{lfp } F_t$$

idea = define a ranking function **counting the number of program steps** from the end of the program

$$F_t(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \mathcal{B} \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



program \mapsto maximal trace semantics \rightarrow **termination semantics**

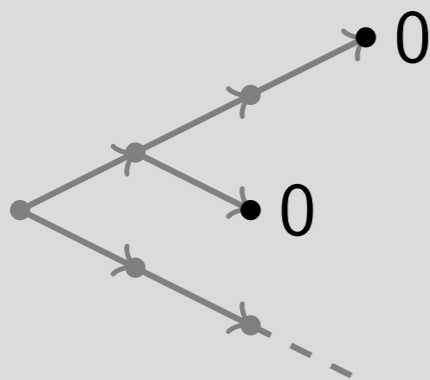
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_t \stackrel{\text{def}}{=} \text{lfp } F_t$$

idea = define a ranking function **counting the number of program steps** from the end of the program

$$F_t(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \mathcal{B} \leftarrow \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



program \mapsto maximal trace semantics \rightarrow **termination semantics**

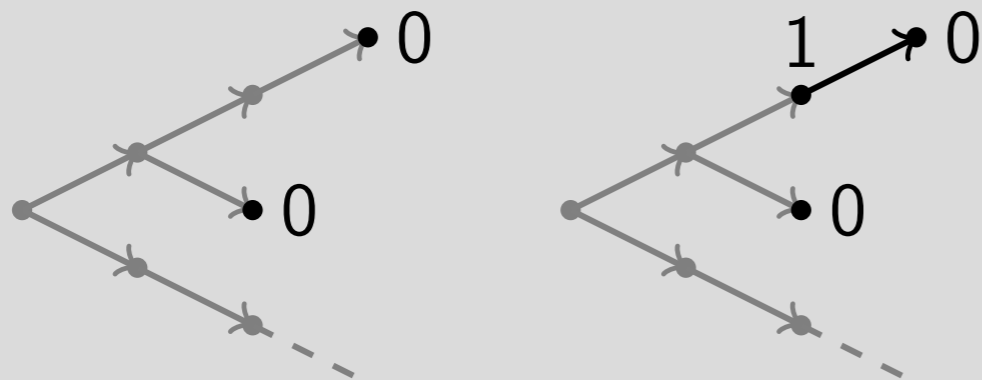
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_t \stackrel{\text{def}}{=} \text{lfp } F_t$$

idea = define a ranking function **counting the number of program steps** from the end of the program

$$F_t(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \mathcal{B} \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \leftarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



program \mapsto maximal trace semantics \rightarrow **termination semantics**

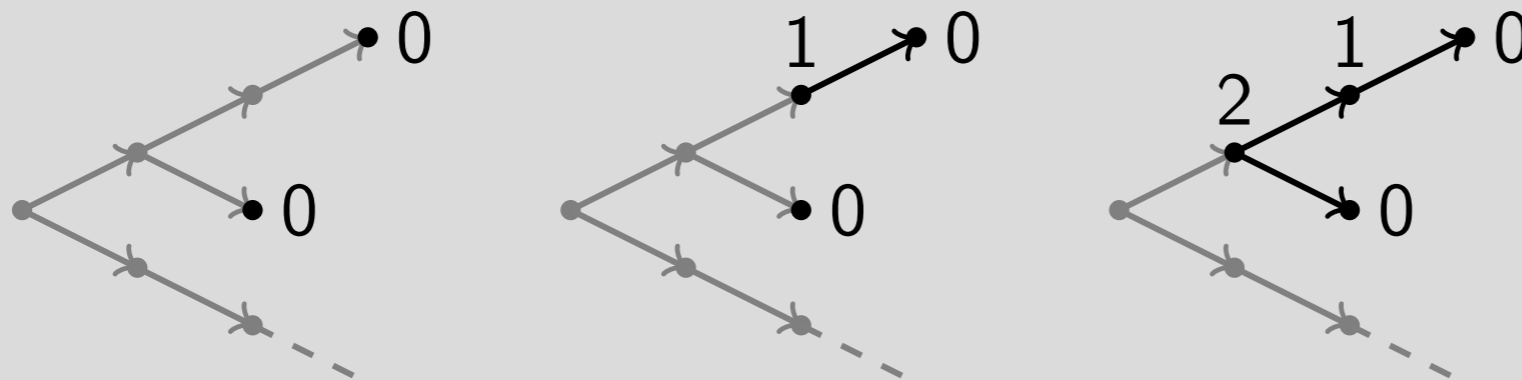
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_t \stackrel{\text{def}}{=} \text{lfp } F_t$$

idea = define a ranking function **counting the number of program steps** from the end of the program

$$F_t(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \mathcal{B} \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \leftarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



program \mapsto maximal trace semantics \rightarrow **termination semantics**

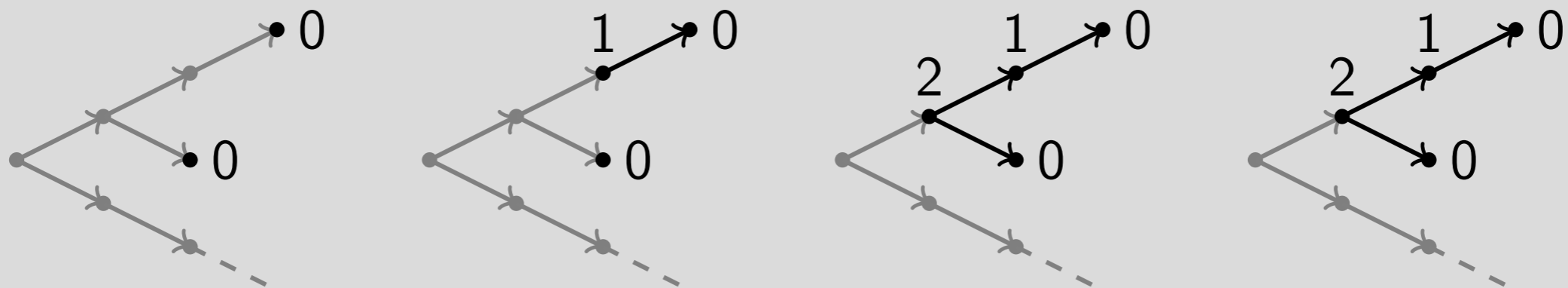
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_t \stackrel{\text{def}}{=} \text{lfp } F_t$$

idea = define a ranking function **counting the number of program steps** from the end of the program

$$F_t(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \mathcal{B} \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



program \mapsto maximal trace semantics \rightarrow **termination semantics**

$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_t \stackrel{\text{def}}{=} \text{lfp } F_t$$

$$F_t(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \mathcal{B} \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition (Computational Order)

$$v_1 \preceq v_2 \stackrel{\text{def}}{=} \text{dom}(v_1) \subseteq \text{dom}(v_2) \wedge \forall x \in \text{dom}(v_1) : v_1(x) \leq v_2(x)$$

program \mapsto maximal trace semantics \rightarrow **termination semantics**

$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_t \stackrel{\text{def}}{=} \text{lfp } F_t$$

$$F_t(\mathbf{v})\mathbf{s} \stackrel{\text{def}}{=} \begin{cases} \mathbf{0} & \mathbf{s} \in \mathcal{B} \\ \sup\{ \mathbf{v}(s') + \mathbf{1} \mid \mathbf{s} \rightarrow s' \} & \mathbf{s} \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \\ \text{undefined} & \text{otherwise} \end{cases}$$

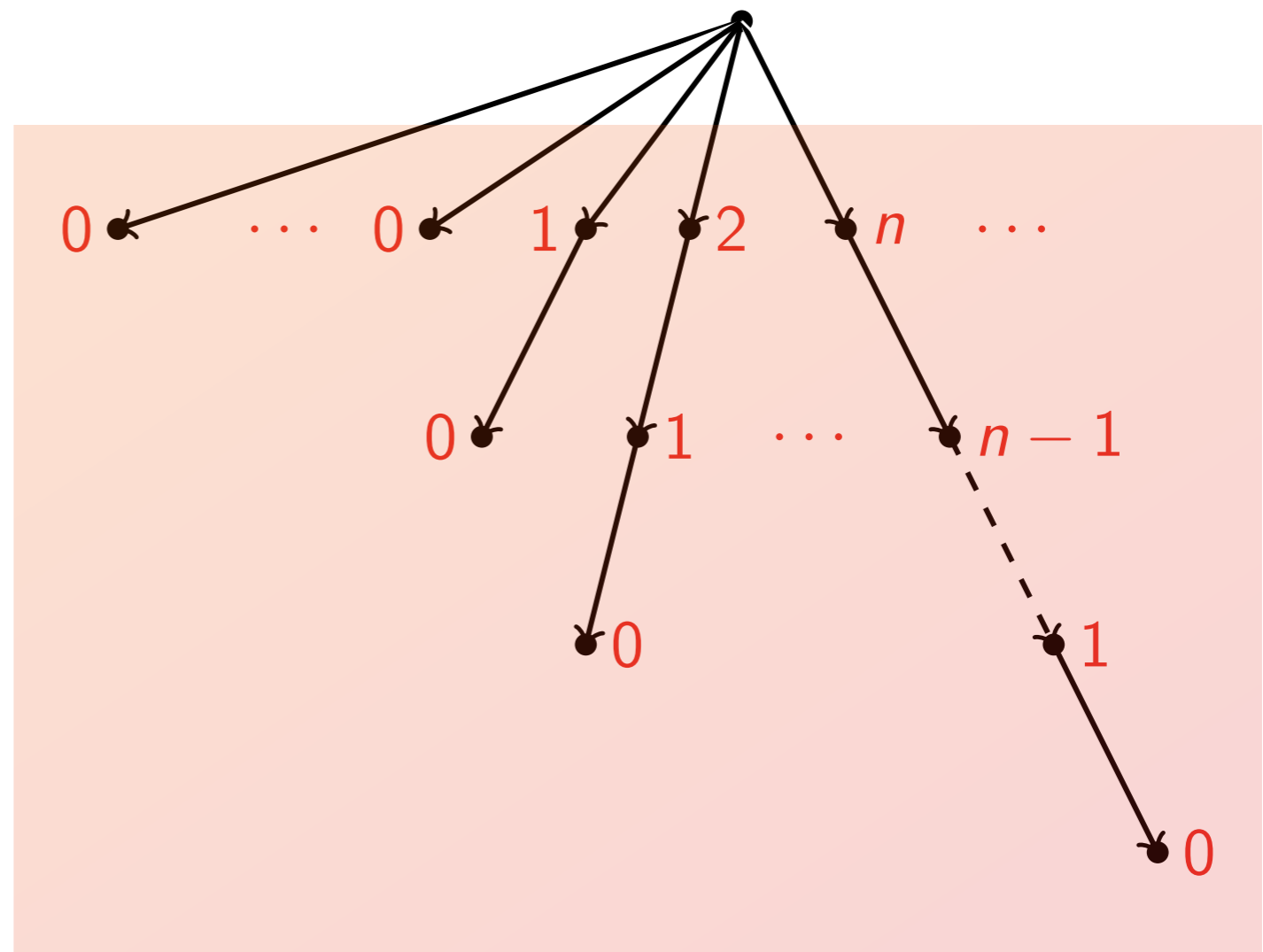
Theorem (Soundness and Completeness)

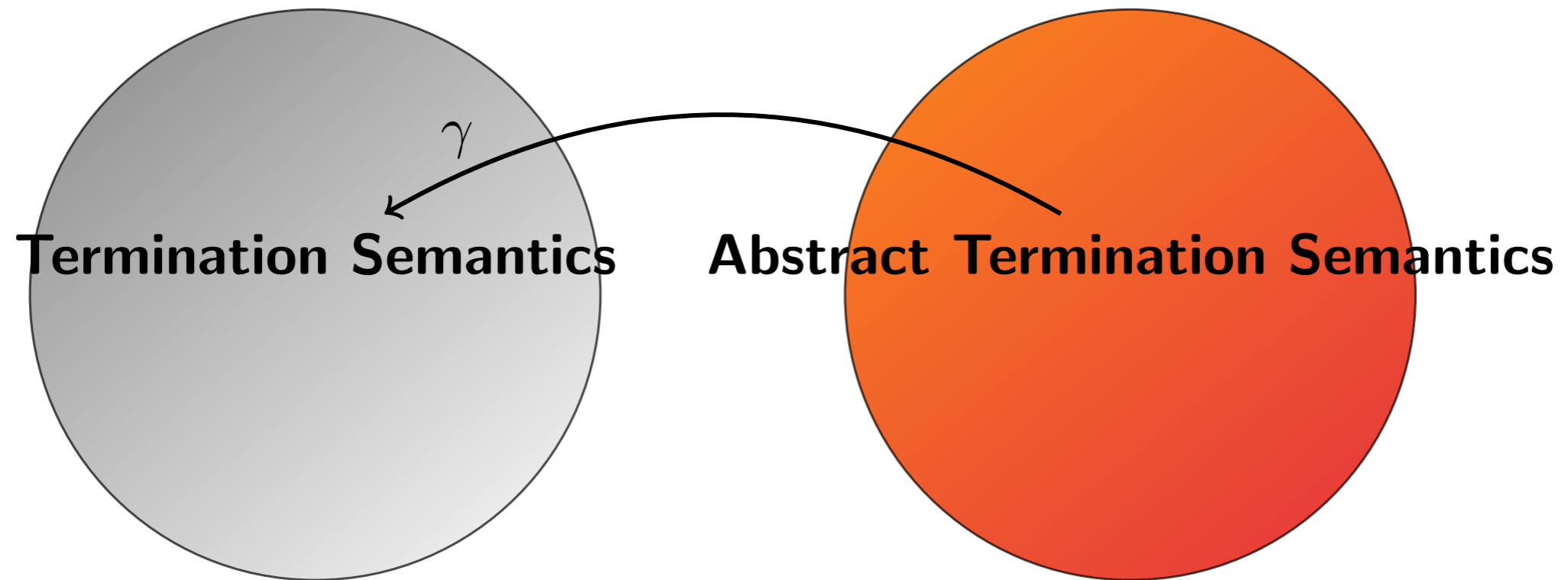
*the termination semantics is **sound** and **complete**
 to prove the termination of programs*

- **remark:** the termination semantics is **not computable!**

Example

```
int : x  
x := ?  
while (x > 0) do  
  x := x - 1  
od
```





Abstractions in the concretization framework

Given a concrete (C, \leq) and an abstract (A, \sqsubseteq) posets
 and a **monotonic concretization** $\gamma : A \rightarrow C$

($\gamma(a)$ is the “meaning” of a in C ; we use intervals in our examples)

- $a \in A$ is a **sound abstraction** of $c \in C$ if $c \leq \gamma(a)$.

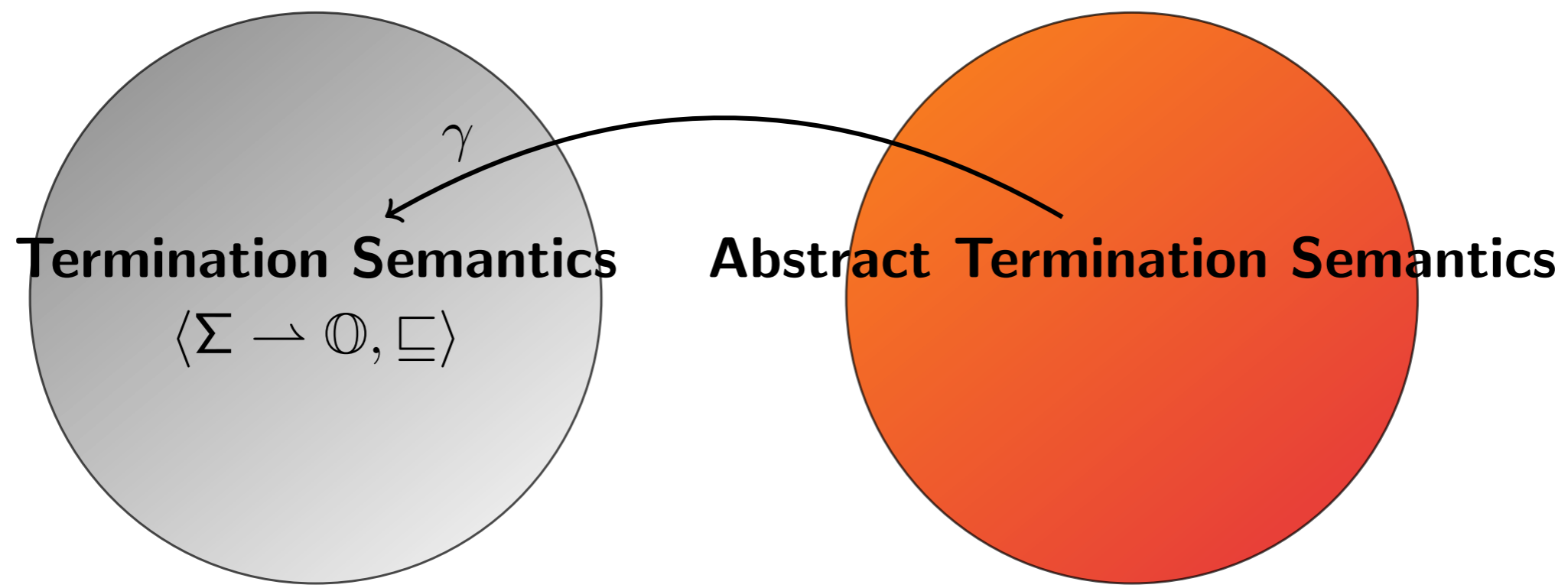
(e.g.: $[0, 10]$ is a sound abstraction of $\{0, 1, 2, 5\}$ in the integer interval domain)

- $g : A \rightarrow A$ is a **sound abstraction** of $f : C \rightarrow C$
 if $\forall a \in A: (f \circ \gamma)(a) \leq (\gamma \circ g)(a)$.

(e.g.: $\lambda([a, b]).[-\infty, +\infty]$ is a sound abstraction of $\lambda X.\{x + 1 \mid x \in X\}$ in the interval domain)

- $g : A \rightarrow A$ is an **exact abstraction** of $f : C \rightarrow C$ if
 $f \circ \gamma = \gamma \circ g$.

(e.g.: $\lambda([a, b]).[a + 1, b + 1]$ is an exact abstraction of $\lambda X.\{x + 1 \mid x \in X\}$ in the interval domain)



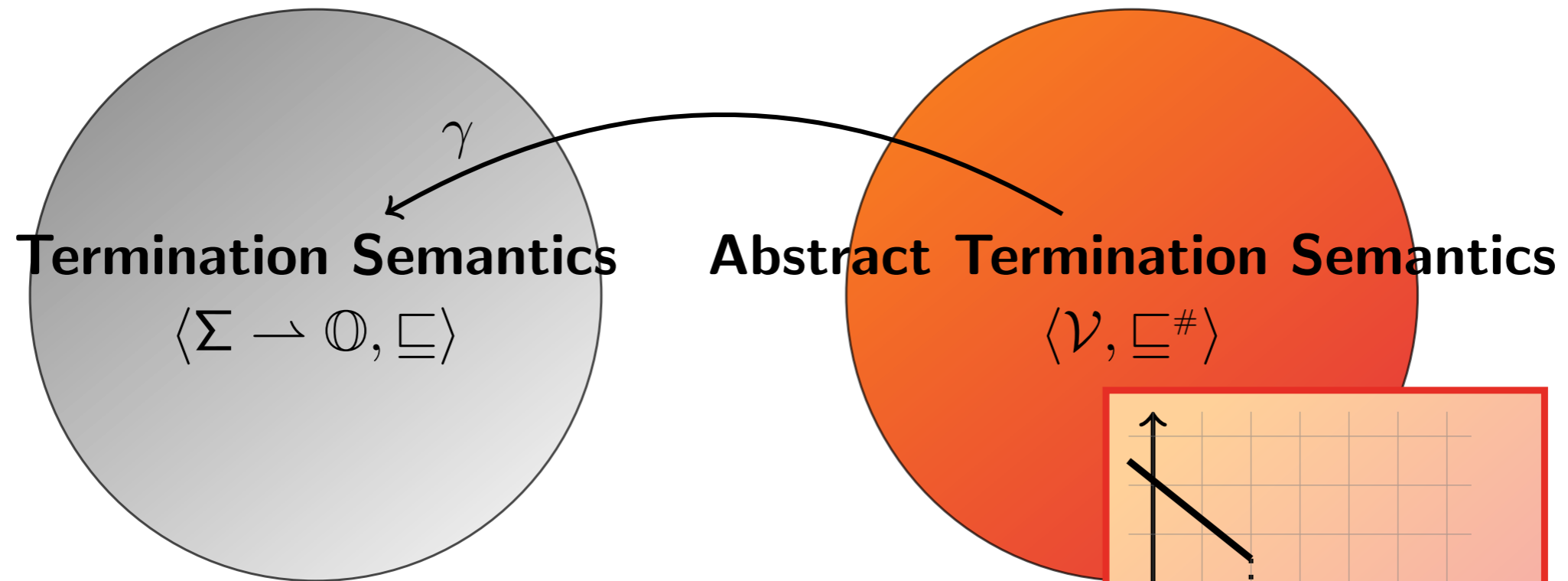
Definition (Approximation Order)

$$v_1 \sqsubseteq v_2 \stackrel{\text{def}}{=} \text{dom}(v_1) \supseteq \text{dom}(v_2) \wedge \forall x \in \text{dom}(v_1) : v_1(x) \leq v_2(x)$$

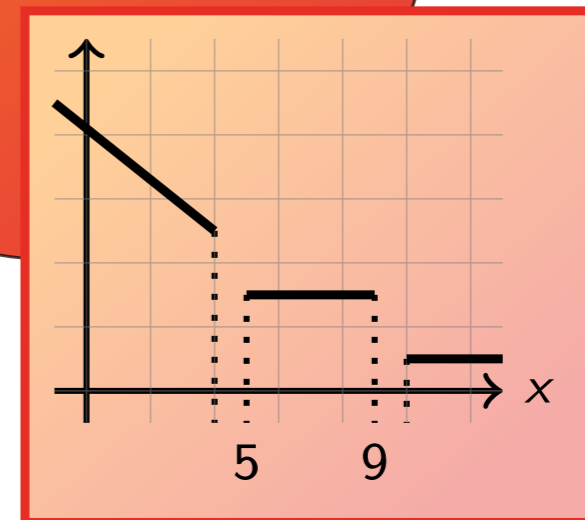
Use of posets (informally)

Posets are a very useful notion to discuss about:

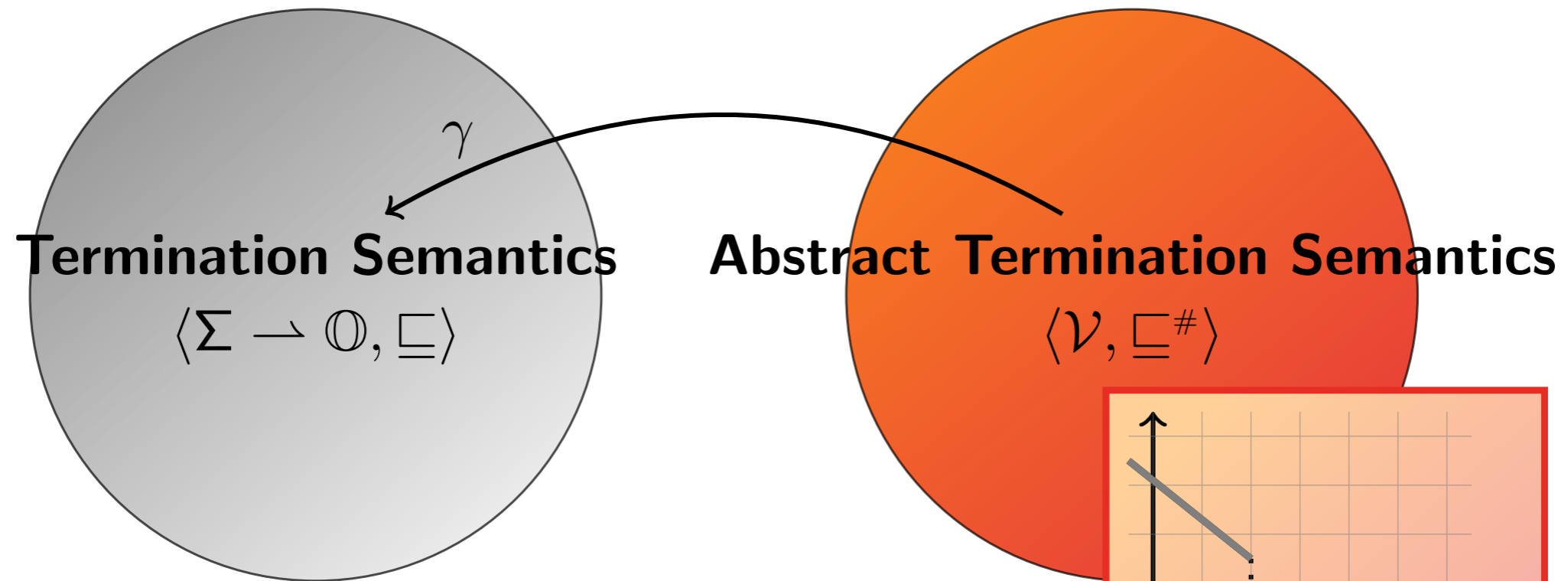
- **logic**: ordered by implication \implies
- **approximations**: \sqsubseteq is an information order
(“ $a \sqsubseteq b$ ” means: “ a carries more information than b ”)
- **program verification**: program semantics \sqsubseteq specification
(e.g.: behaviors of program \subseteq accepted behaviors)
- **iteration**: fixpoint computation
(e.g., a computation is directed, with a limit: $X_1 \sqsubseteq X_2 \sqsubseteq \dots \sqsubseteq X_n$)



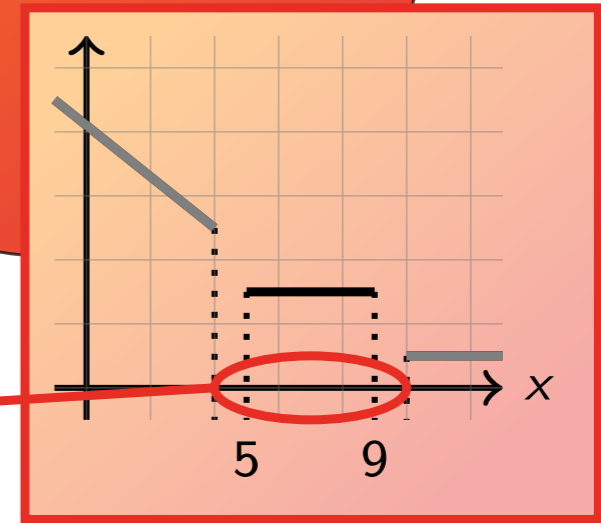
- States Abstract Domain
- Functions Abstract Domain
- Piecewise-Defined Ranking Functions Abstract Domain $V(S, F)$**

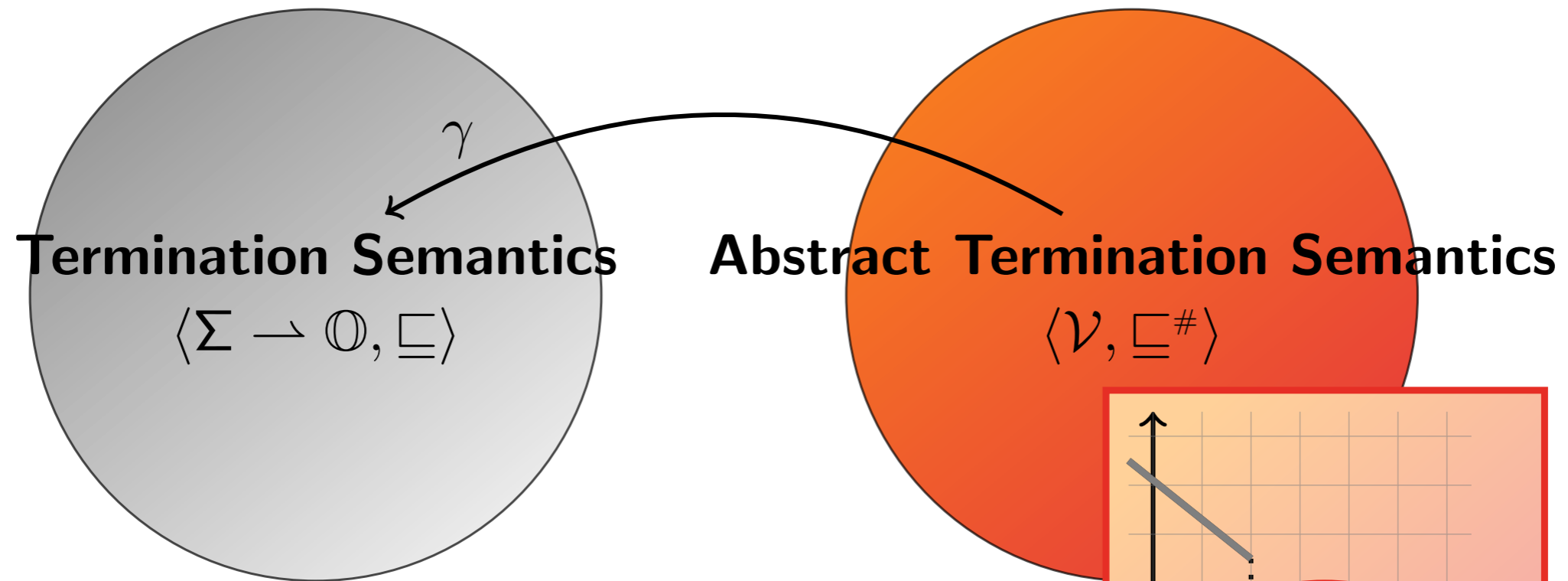


S
F

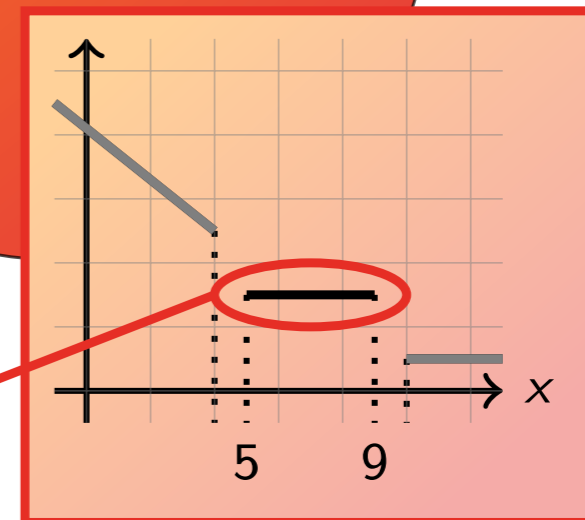


- **States Abstract Domain** ←
- Functions Abstract Domain
- Piecewise-Defined Ranking Functions Abstract Domain V(S, F)

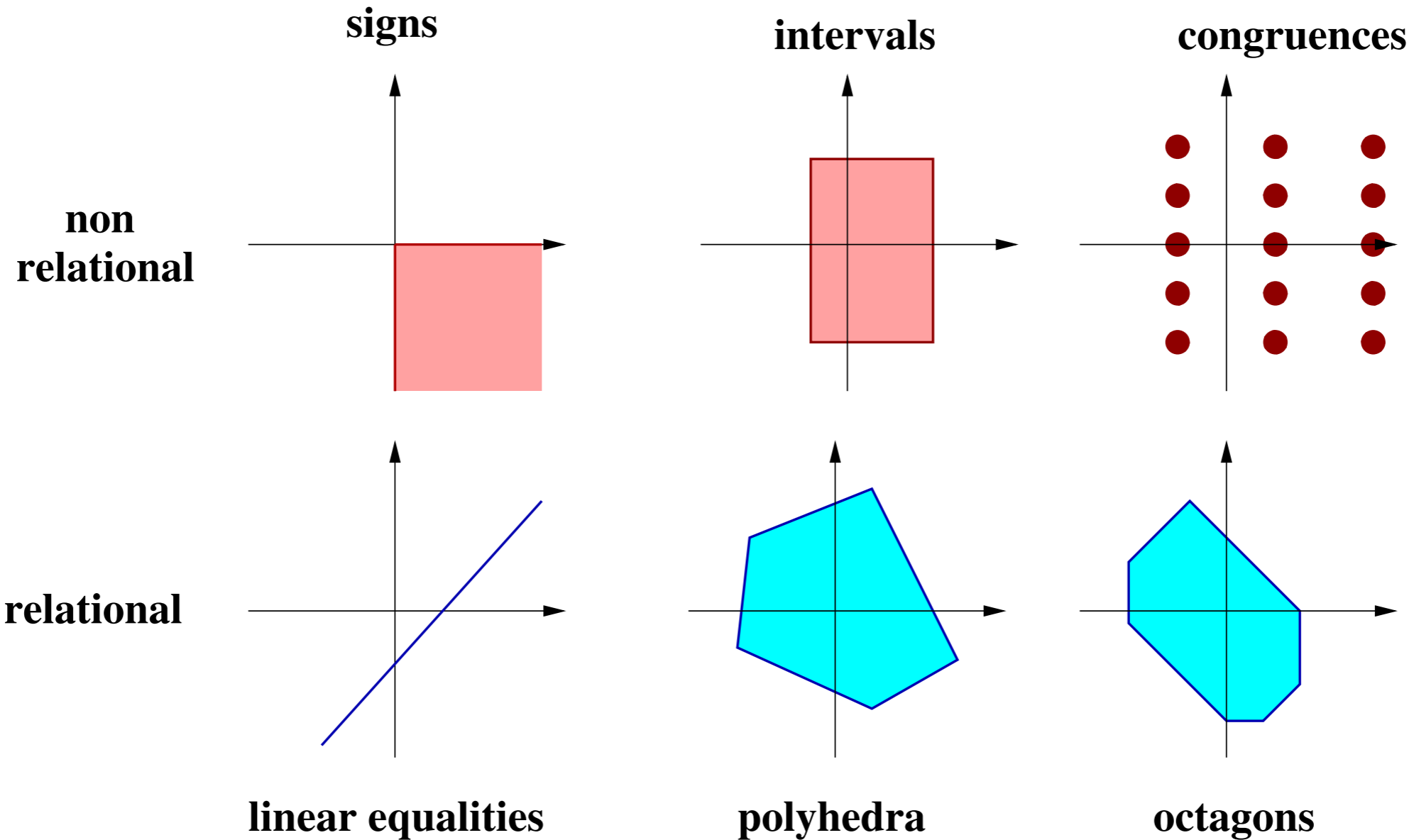


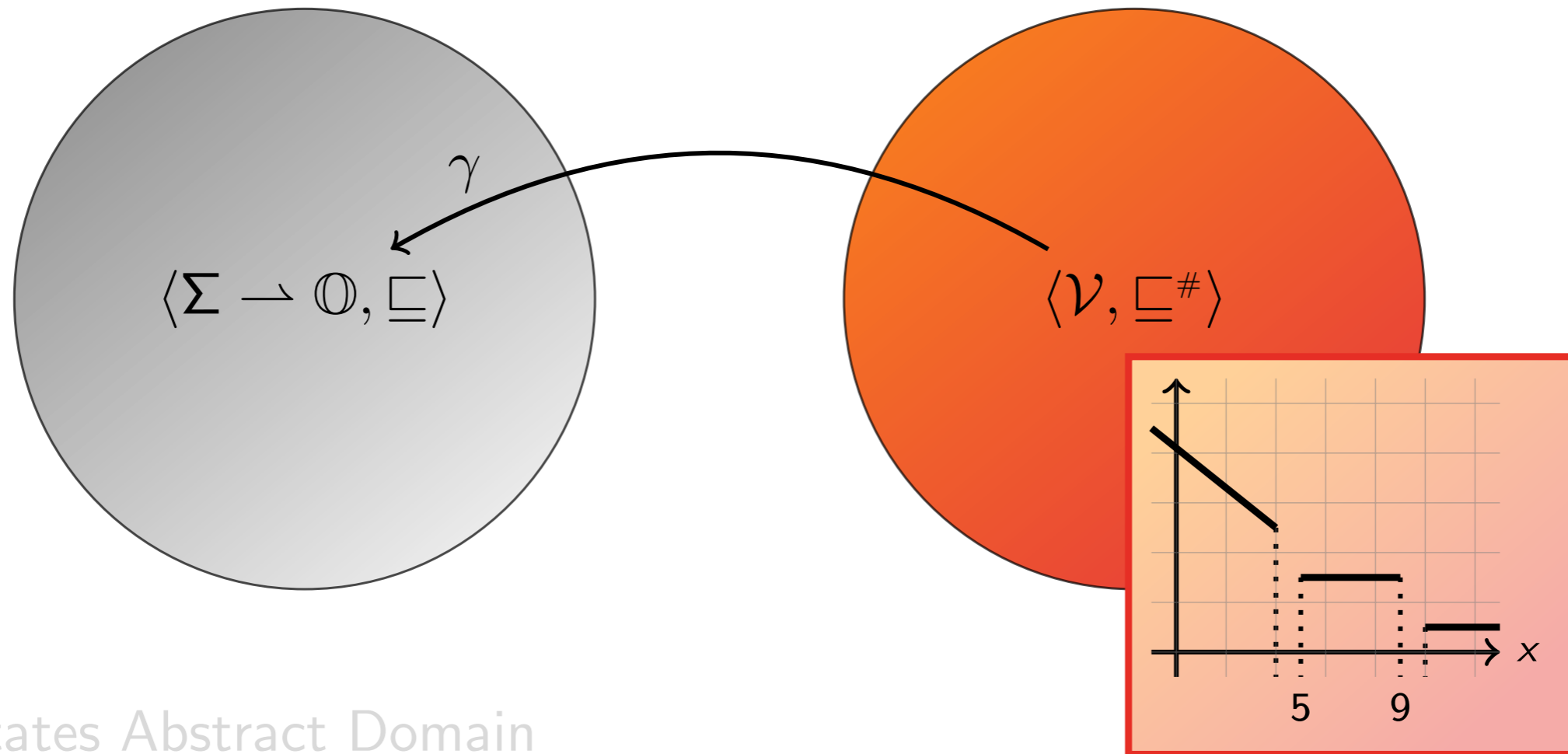


- States Abstract Domain
 - **Functions Abstract Domain**
 - Piecewise-Defined Ranking Functions Abstract Domain
- S
F
V(S, F)

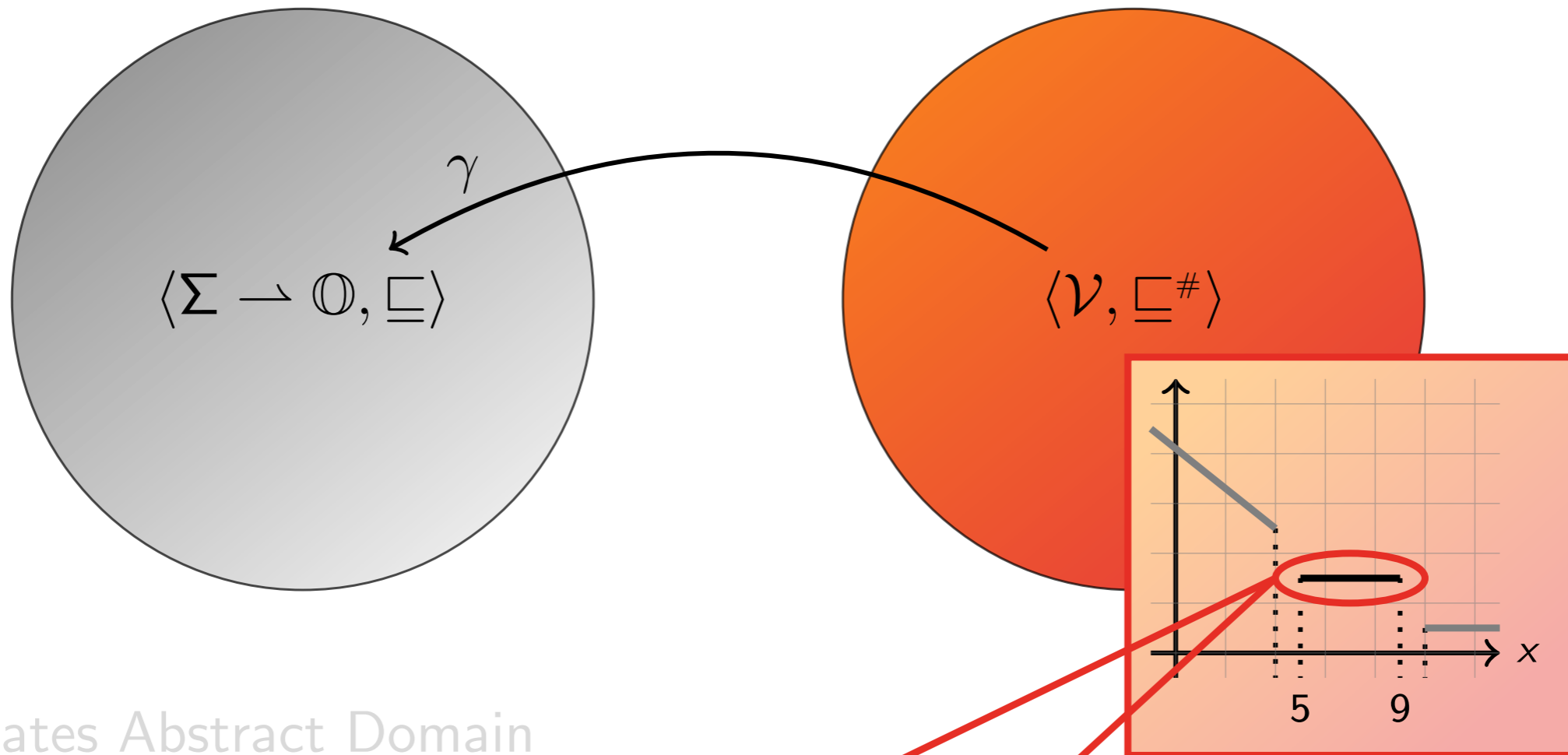


Numerical abstract domain examples

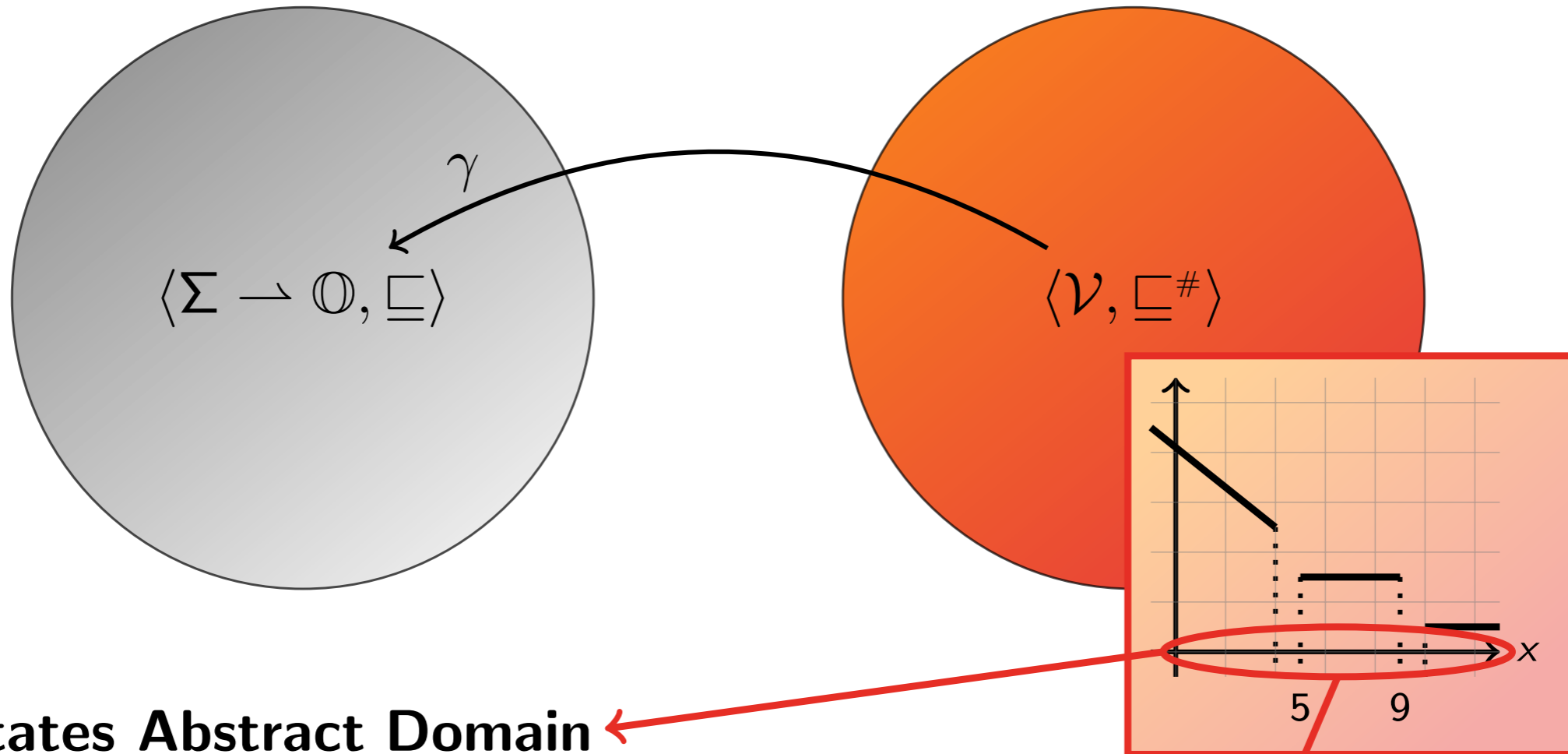




- States Abstract Domain
 - $\mathcal{L} \stackrel{\text{def}}{=} \text{Interval/Octagonal/Polyhedral Linear Constraints}$
- Functions Abstract Domain
 - $\mathcal{F} \stackrel{\text{def}}{=} \{\perp\} \cup \{f \mid f \in \mathbb{Z}^n \rightarrow \mathbb{N}\} \cup \{\top\}$
 where $f \equiv f(x_1, \dots, x_n) = m_1x_1 + \dots + m_nx_n + q$
- **Piecewise-Defined Ranking Functions Abstract Domain**
 - $\mathcal{V} \stackrel{\text{def}}{=} \{\text{LEAF} : f \mid f \in \mathcal{F}\} \cup \{\text{NODE}\{c\} : t_1, t_2 \mid c \in \mathcal{L} \wedge t_1, t_2 \in \mathcal{V}\}$



- States Abstract Domain
 - $\mathcal{L} \stackrel{\text{def}}{=} \text{Interval/Octagonal/Polyhedral Linear Constraints}$
- **Functions Abstract Domain**
 - $\mathcal{F} \stackrel{\text{def}}{=} \{\perp\} \cup \{f \mid f \in \mathbb{Z}^n \rightarrow \mathbb{N}\} \cup \{\top\}$
 where $f \equiv f(x_1, \dots, x_n) = m_1x_1 + \dots + m_nx_n + q$
- Piecewise-Defined Ranking Functions Abstract Domain
 - $\mathcal{V} \stackrel{\text{def}}{=} \{\mathbf{LEAF} : \mathbf{f} \mid \mathbf{f} \in \mathcal{F}\} \cup \{\mathbf{NODE}\{c\} : t_1, t_2 \mid c \in \mathcal{L} \wedge t_1, t_2 \in \mathcal{V}\}$



- **States Abstract Domain**
 - $\mathcal{L} \stackrel{\text{def}}{=} \text{Interval/Octagonal/Polyhedral Linear Constraints}$
- Functions Abstract Domain
 - $\mathcal{F} \stackrel{\text{def}}{=} \{\perp\} \cup \{f \mid f \in \mathbb{Z}^n \rightarrow \mathbb{N}\} \cup \{T\}$
 where $f \equiv f(x_1, \dots, x_n) = m_1x_1 + \dots + m_nx_n + q$
- Piecewise-Defined Ranking Functions Abstract Domain
 - $\mathcal{V} \stackrel{\text{def}}{=} \{\text{LEAF} : f \mid f \in \mathcal{F}\} \cup \{\text{NODE}\{c\} : t_1, t_2 \mid c \in \mathcal{L} \wedge t_1, t_2 \in \mathcal{V}\}$

Example

```
int : x, y  
while 1(x > 0) do  
  2x := x - y  
od3
```

Example

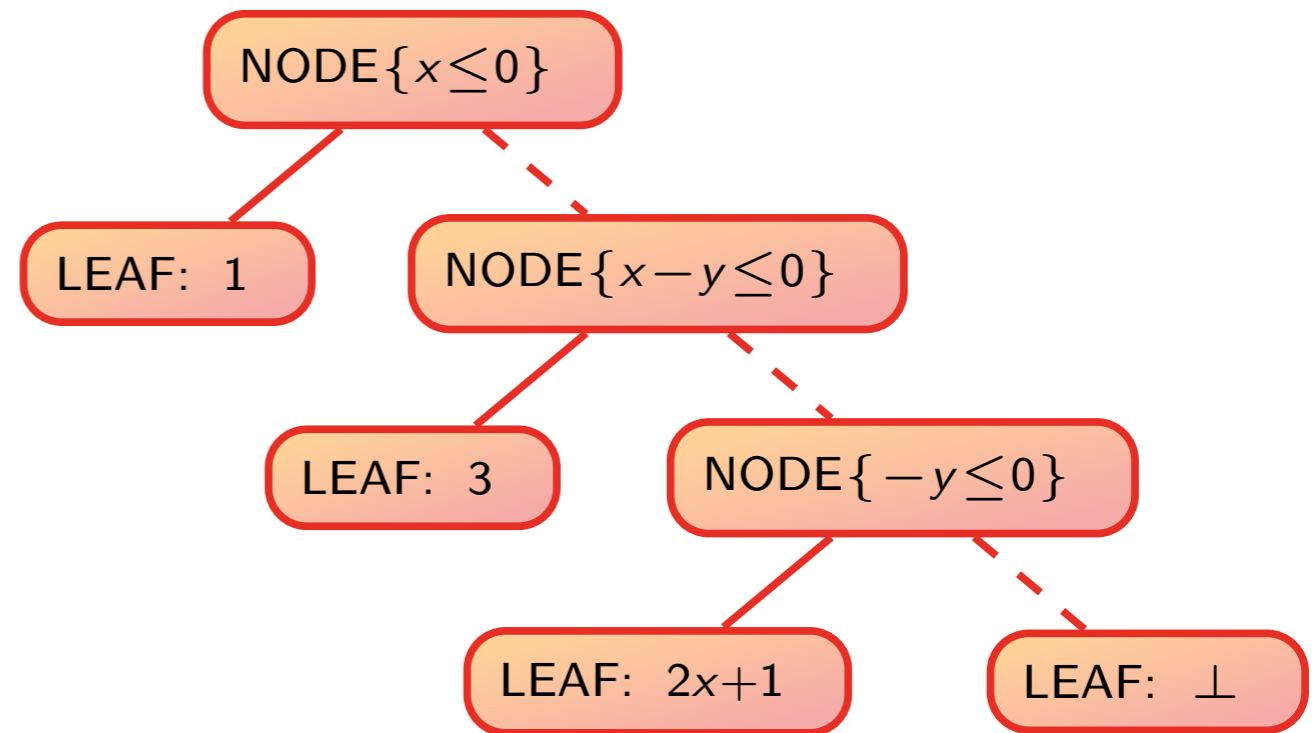
```
int : x, y  
while 1(x > 0) do  
  2x := x - y  
od3
```

the program terminates if
and only if $x \leq 0 \vee y > 0$

Example

```
int : x, y  
while 1(x > 0) do  
  2x := x - y  
od3
```

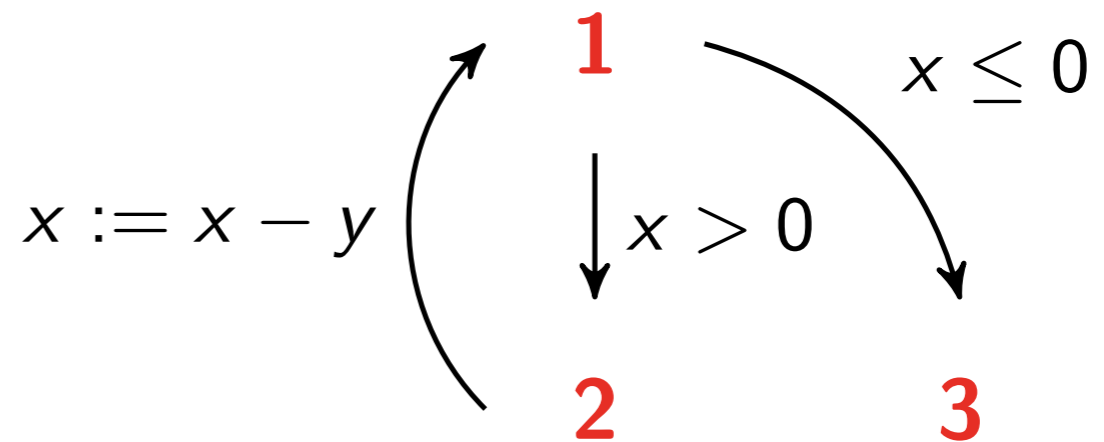
the program terminates if
and only if $x \leq 0 \vee y > 0$



Example

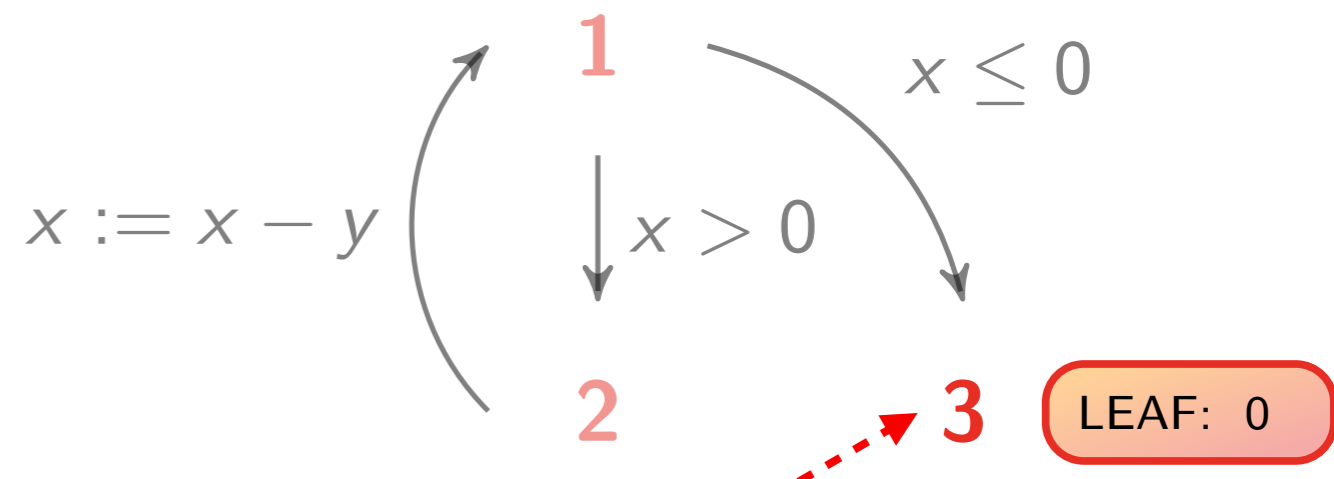
```
int : x, y  
while 1(x > 0) do  
  2x := x - y  
od3
```

we will map each point
to a function of x and y giving
an **upper bound** on the
steps before termination

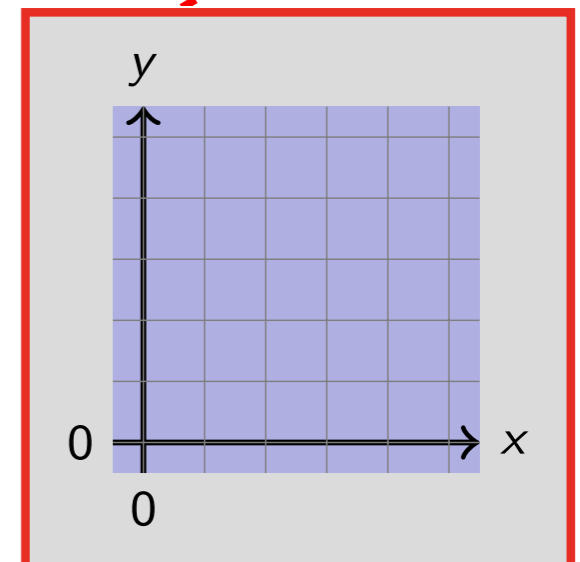


Example

```
int : x, y  
while 1(x > 0) do  
  2x := x - y  
od3
```

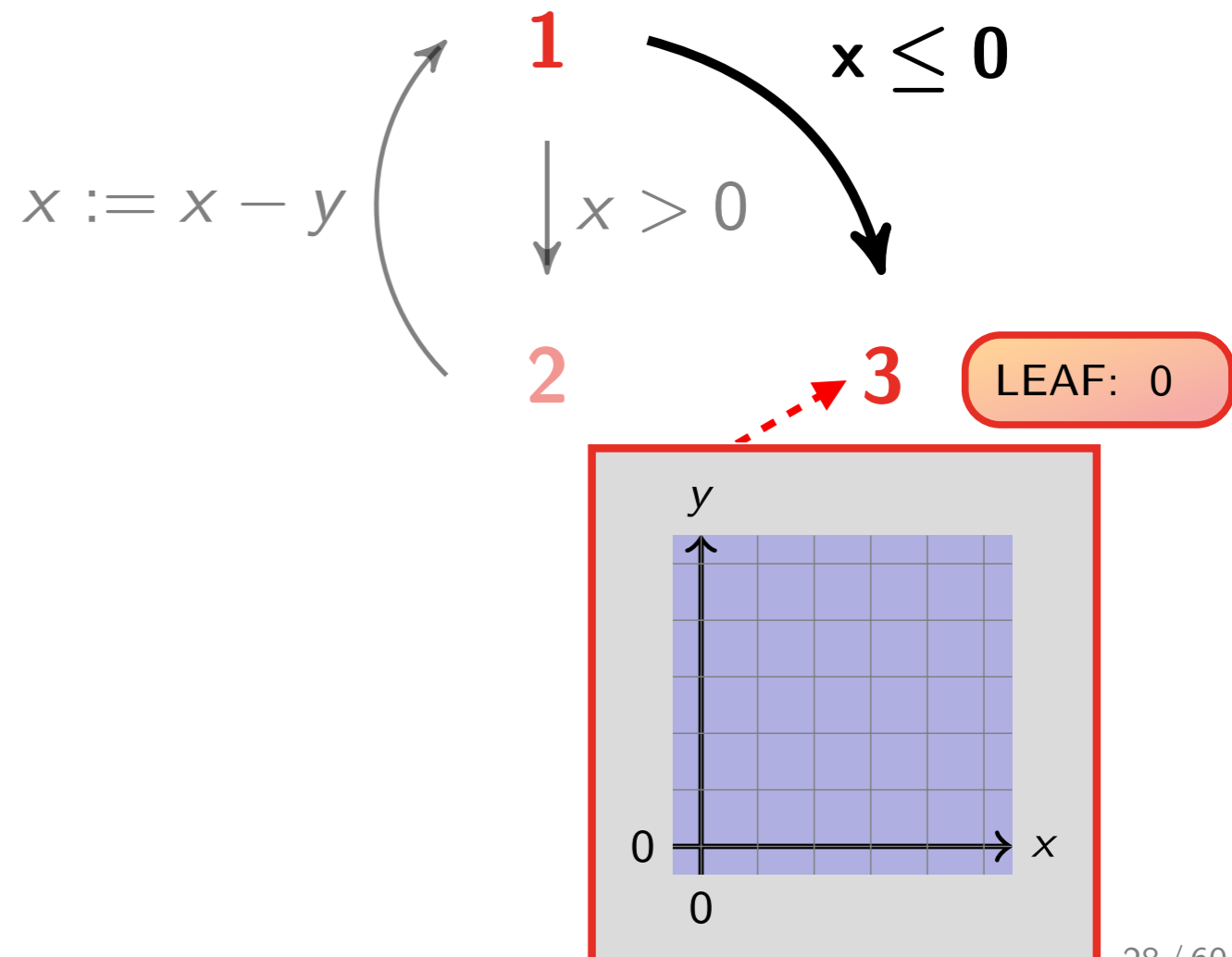


we start at the end
with 0 steps
before termination



Example

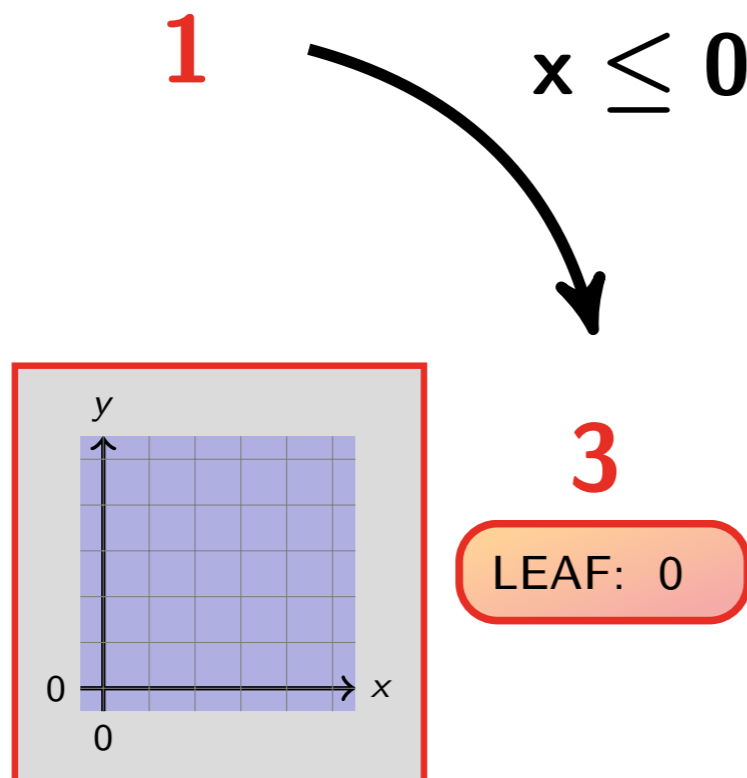
```
int : x, y  
while 1(x > 0) do  
  2x := x - y  
od3
```



Tests

Algorithm 4 : Tree Filter

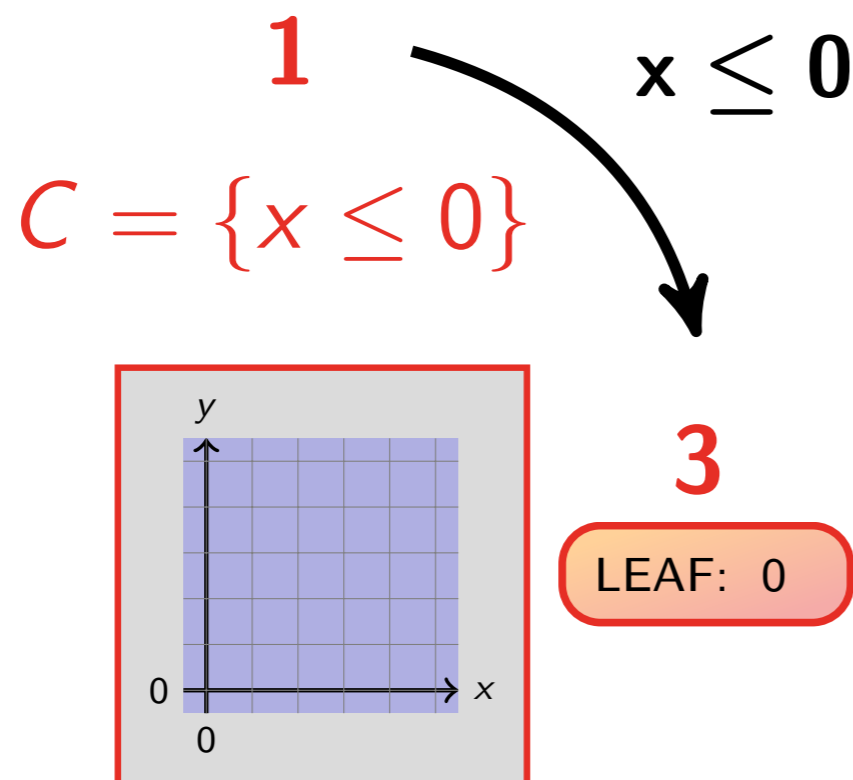
```
1: function FILTER-AUX( $t, c$ )  
2:   if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq$  LEAF :  $f$  */  
3:   else return NODE{ $t.c$ } : FILTER-AUX( $t.l, c$ ); FILTER-AUX( $t.r, c$ )  
  
4: function FILTER( $t, c$ )  
5:    $C \leftarrow$  FILTERL( $c$ )  
6:   return AUGMENT(FILTER-AUX( $t, c$ ),  $C$ )
```



Tests

Algorithm 4 : Tree Filter

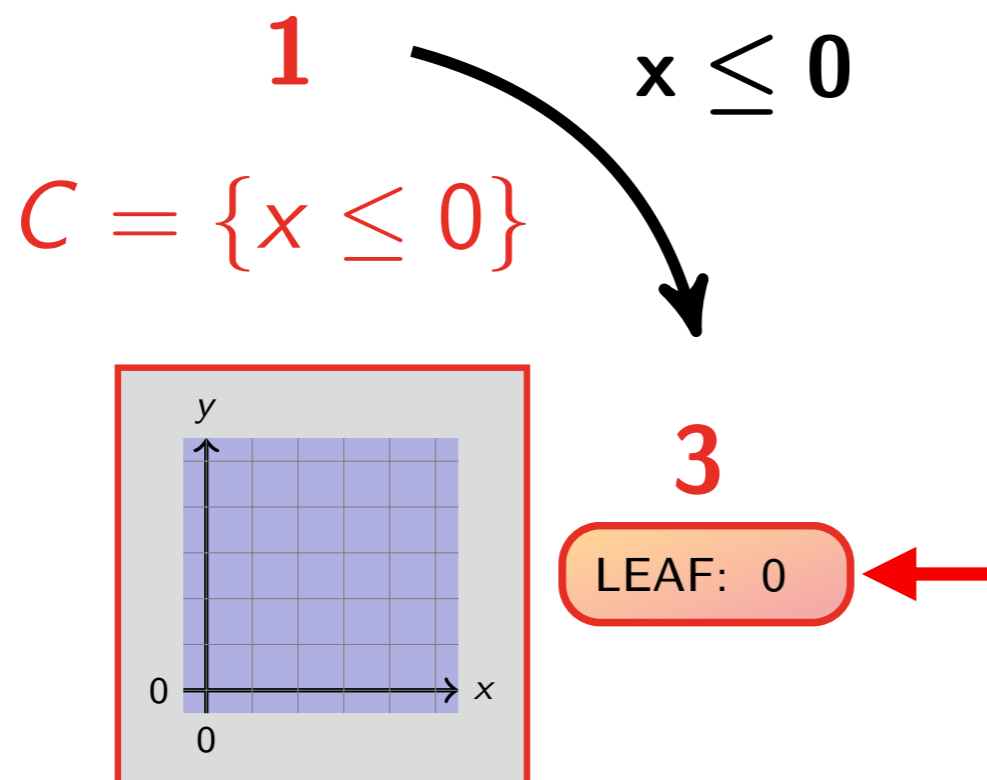
```
1: function FILTER-AUX( $t, c$ )  
2:   if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq$  LEAF :  $f$  */  
3:   else return NODE{ $t.c$ } : FILTER-AUX( $t.l, c$ ); FILTER-AUX( $t.r, c$ )  
  
4: function FILTER( $t, c$ )  
5:    $C \leftarrow$  FILTERL( $c$ )  
6:   return AUGMENT(FILTER-AUX( $t, c$ ),  $C$ )
```



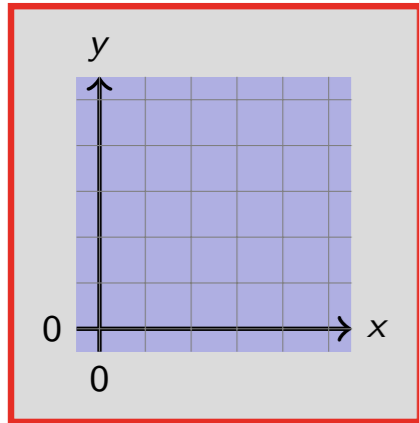
Tests

Algorithm 4 : Tree Filter

```
1: function FILTER-AUX( $t, c$ )  
2:   if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq$  LEAF :  $f$  */  
3:   else return NODE{ $t.c$ } : FILTER-AUX( $t.l, c$ ); FILTER-AUX( $t.r, c$ )  
  
4: function FILTER( $t, c$ )  
5:    $C \leftarrow$  FILTERL( $c$ )  
6:   return AUGMENT(FILTER-AUX( $t, c$ ),  $C$ )
```



Tests

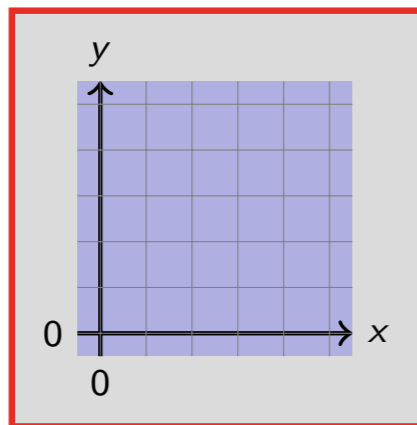


LEAF: 1

1

$$x \leq 0$$

$$C = \{x \leq 0\}$$



3

LEAF: 0

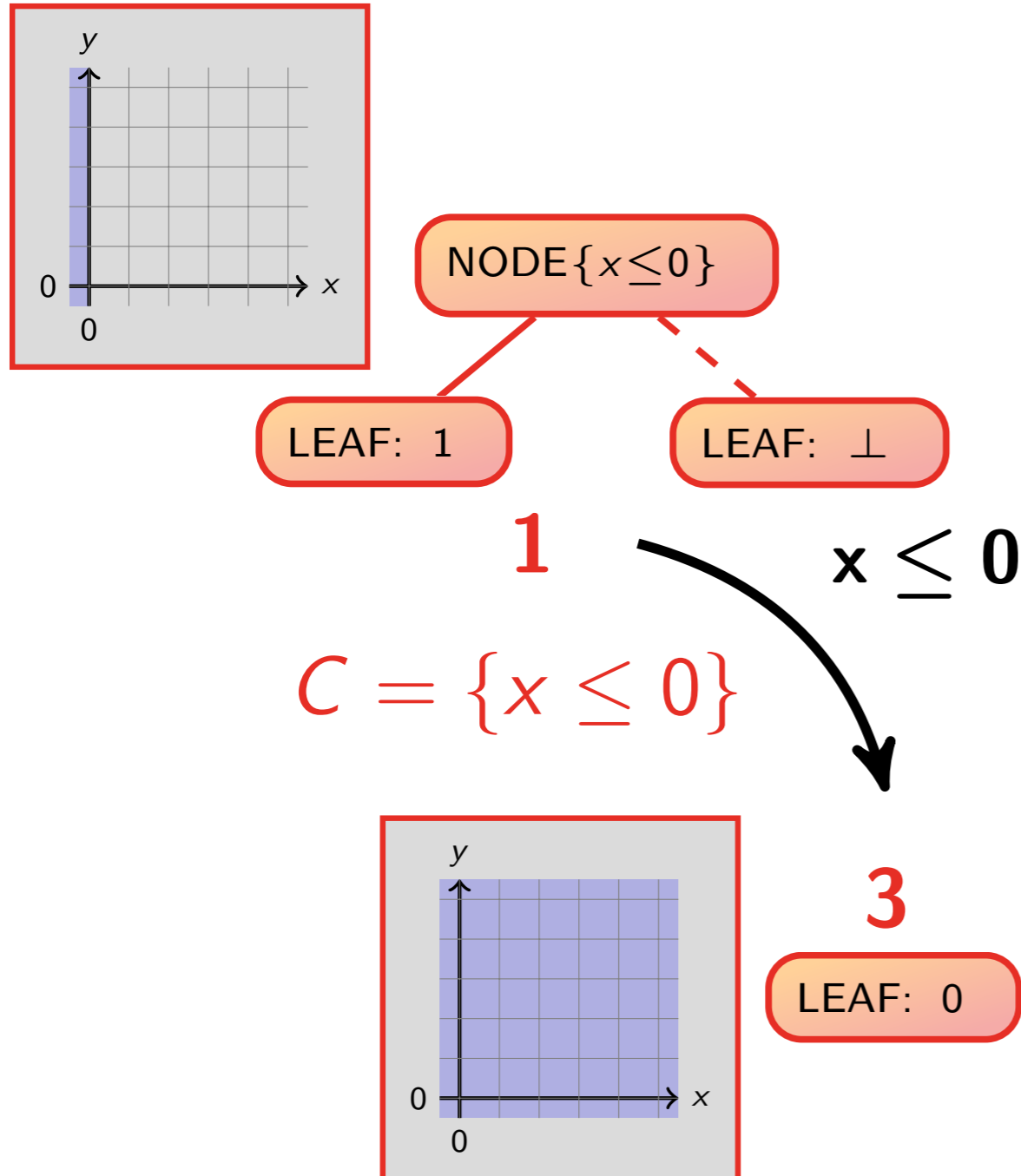
Algorithm 4 : Tree Filter

```

1: function FILTER-AUX( $t, c$ )
2:   if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq$  LEAF :  $f$  */
3:   else return NODE{ $t.c$ } : FILTER-AUX( $t.l, c$ ); FILTER-AUX( $t.r, c$ )

4: function FILTER( $t, c$ )
5:    $C \leftarrow$  FILTERL( $c$ )
6:   return AUGMENT(FILTER-AUX( $t, c$ ),  $C$ )
  
```

Tests



Algorithm 4 : Tree Filter

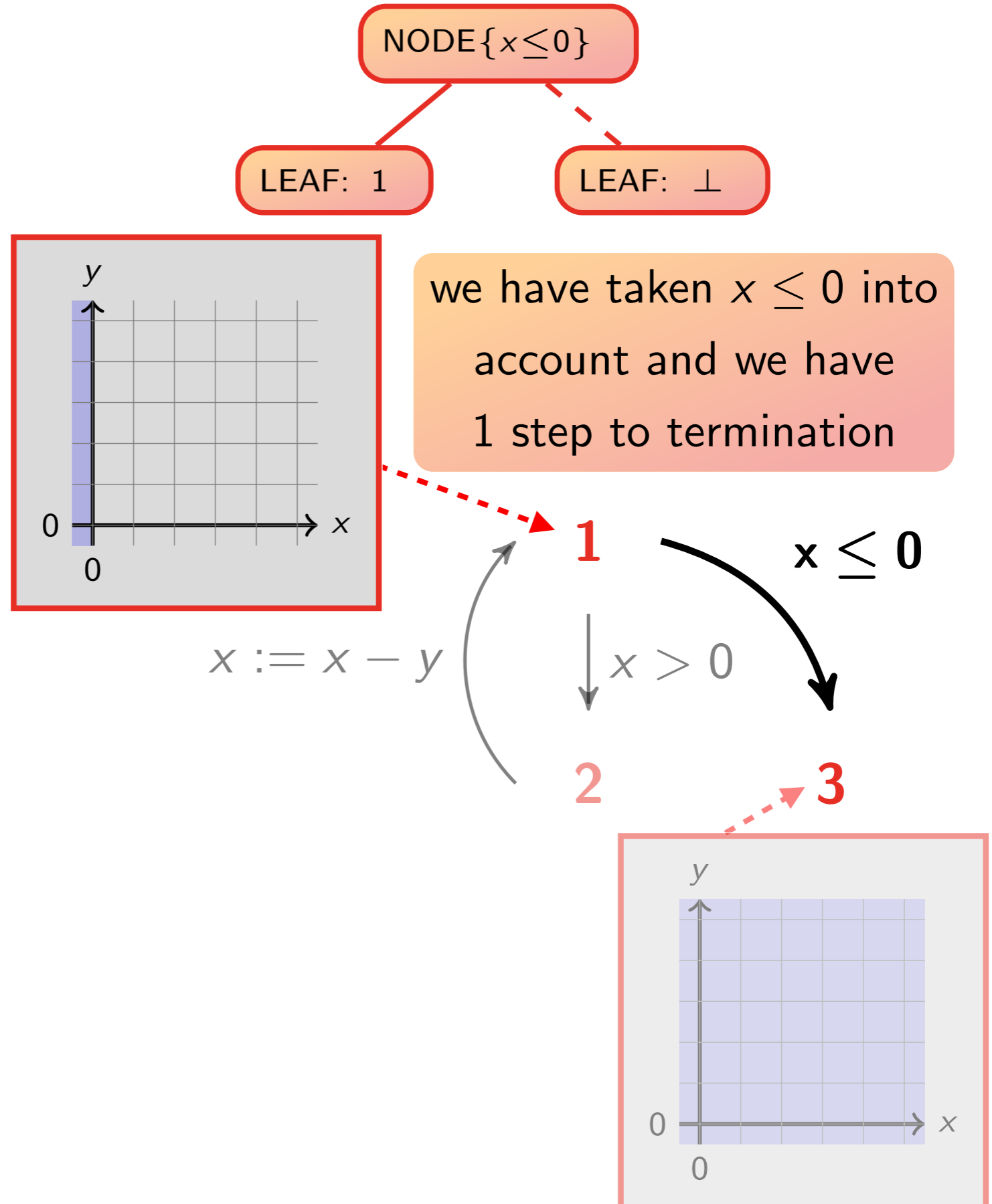
```

1: function FILTER-AUX( $t, c$ )
2:   if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq \text{LEAF} : f$  */
3:   else return NODE{ $t.c$ } : FILTER-AUX( $t.l, c$ ); FILTER-AUX( $t.r, c$ )

4: function FILTER( $t, c$ )
5:    $C \leftarrow \text{FILTER}_L(c)$ 
6:   return AUGMENT(FILTER-AUX( $t, c$ ),  $C$ )
  
```

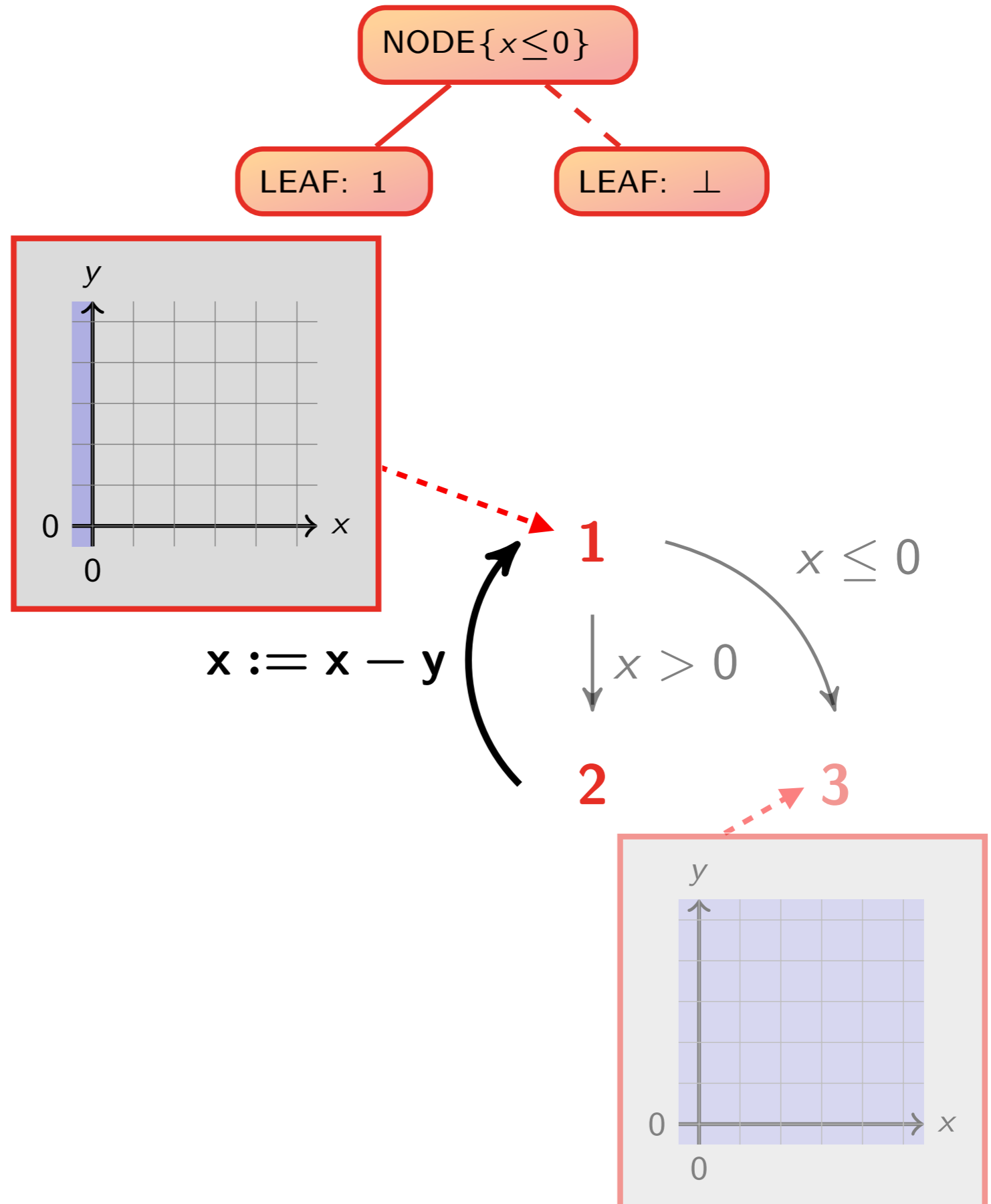
Example

```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

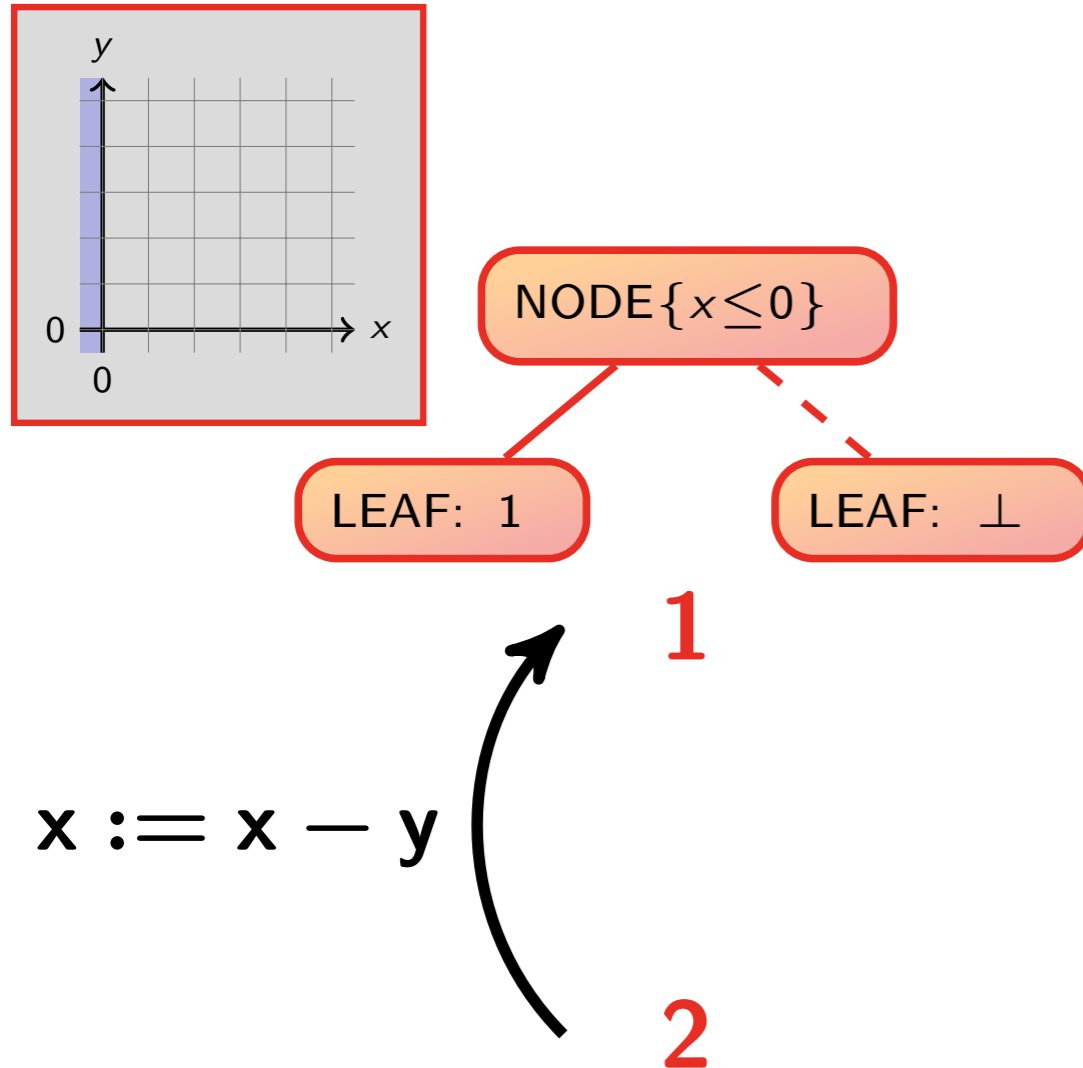


Example

```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



Assignments

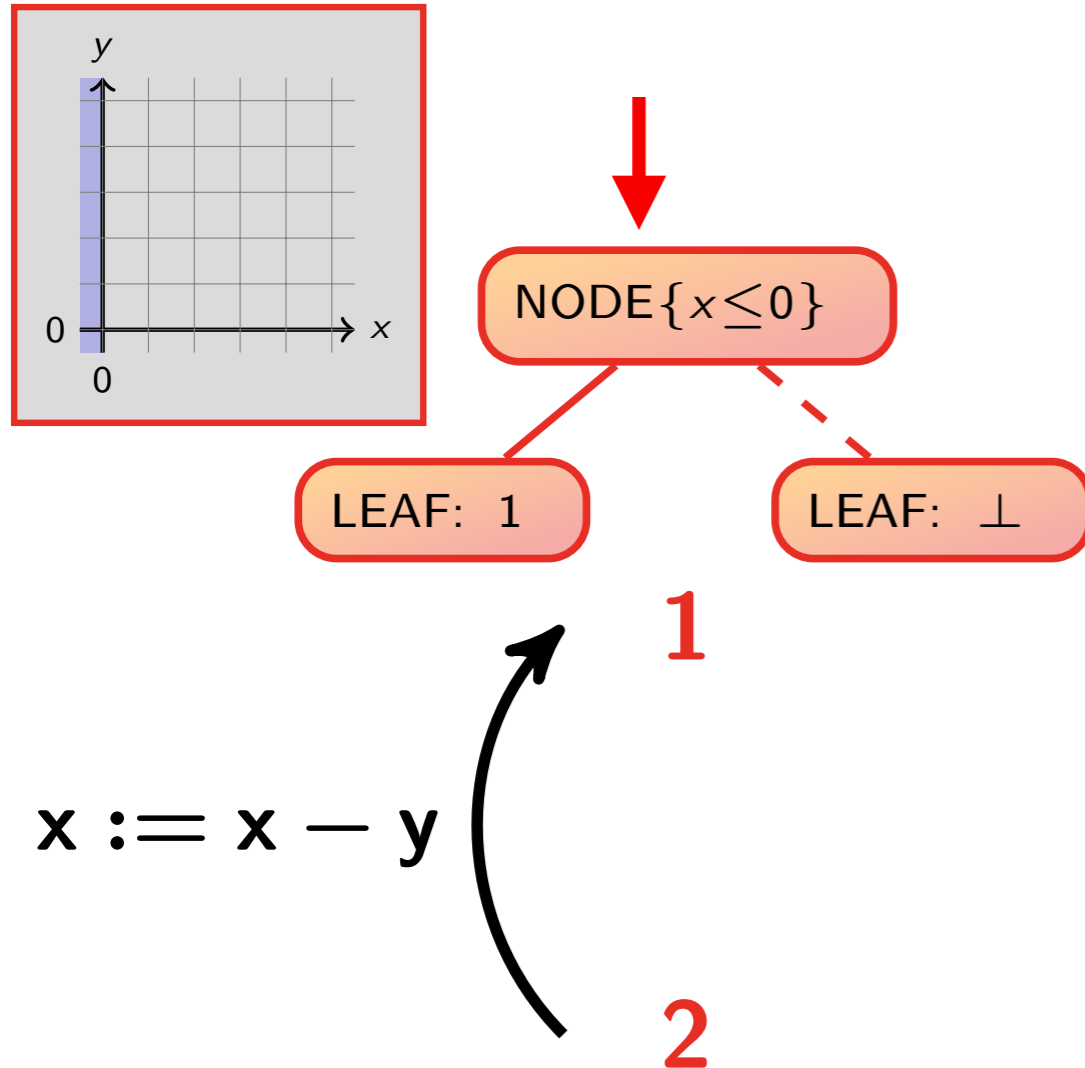


Algorithm 3 : Tree Assign

```

1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )      /*  $t \triangleq$  LEAF :  $f$  */
3:   else
4:      $C \leftarrow$  ASSIGNL( $t.c, x := a$ )
5:     if ISEMPY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_{\top}$  ASSIGN( $t.r, x := a$ )
6:     else if ISUNSAT( $C$ ) then return ASSIGN( $t.r, x := a$ )
7:     else
8:        $l \leftarrow$  AUGMENT(ASSIGN( $t.l, x := a$ ),  $C$ )
9:        $r \leftarrow$  AUGMENT(ASSIGN( $t.r, x := a$ ),  $C$ )
10:    return NODE{ $l.c$ } :  $l; r$ 
    
```


Assignments



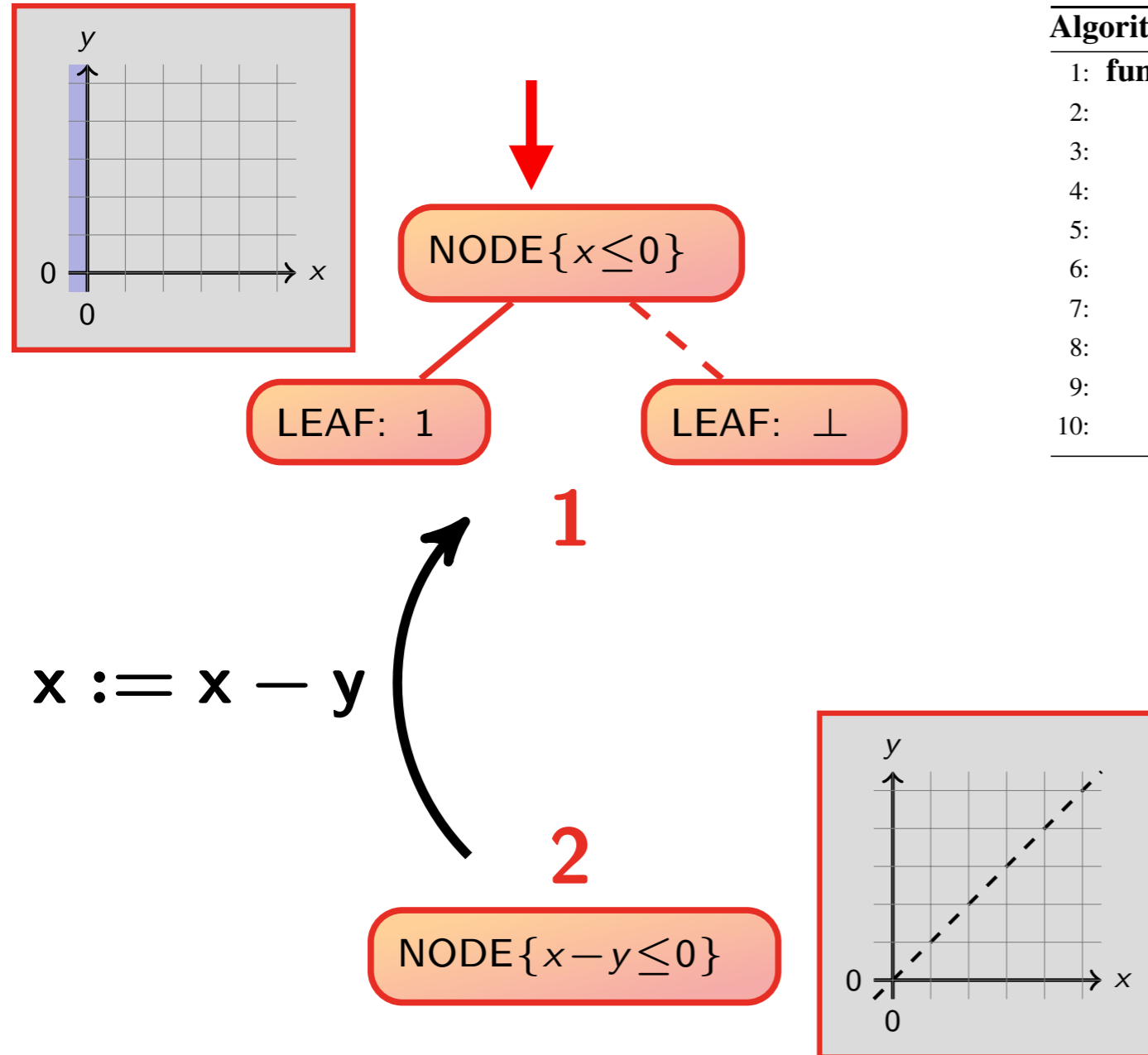
Algorithm 3 : Tree Assign

```

1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )      /*  $t \triangleq \text{LEAF} : f$  */
3:   else
4:      $C \leftarrow \text{ASSIGN}_L(t.c, x := a)$ 
5:     if ISEMPY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_{\top}$  ASSIGN( $t.r, x := a$ )
6:     else if ISUNSAT( $C$ ) then return ASSIGN( $t.r, x := a$ )
7:     else
8:        $l \leftarrow \text{AUGMENT}(\text{ASSIGN}(t.l, x := a), C)$ 
9:        $r \leftarrow \text{AUGMENT}(\text{ASSIGN}(t.r, x := a), C)$ 
10:    return NODE{ $l.c$ } :  $l; r$ 

```

Assignments



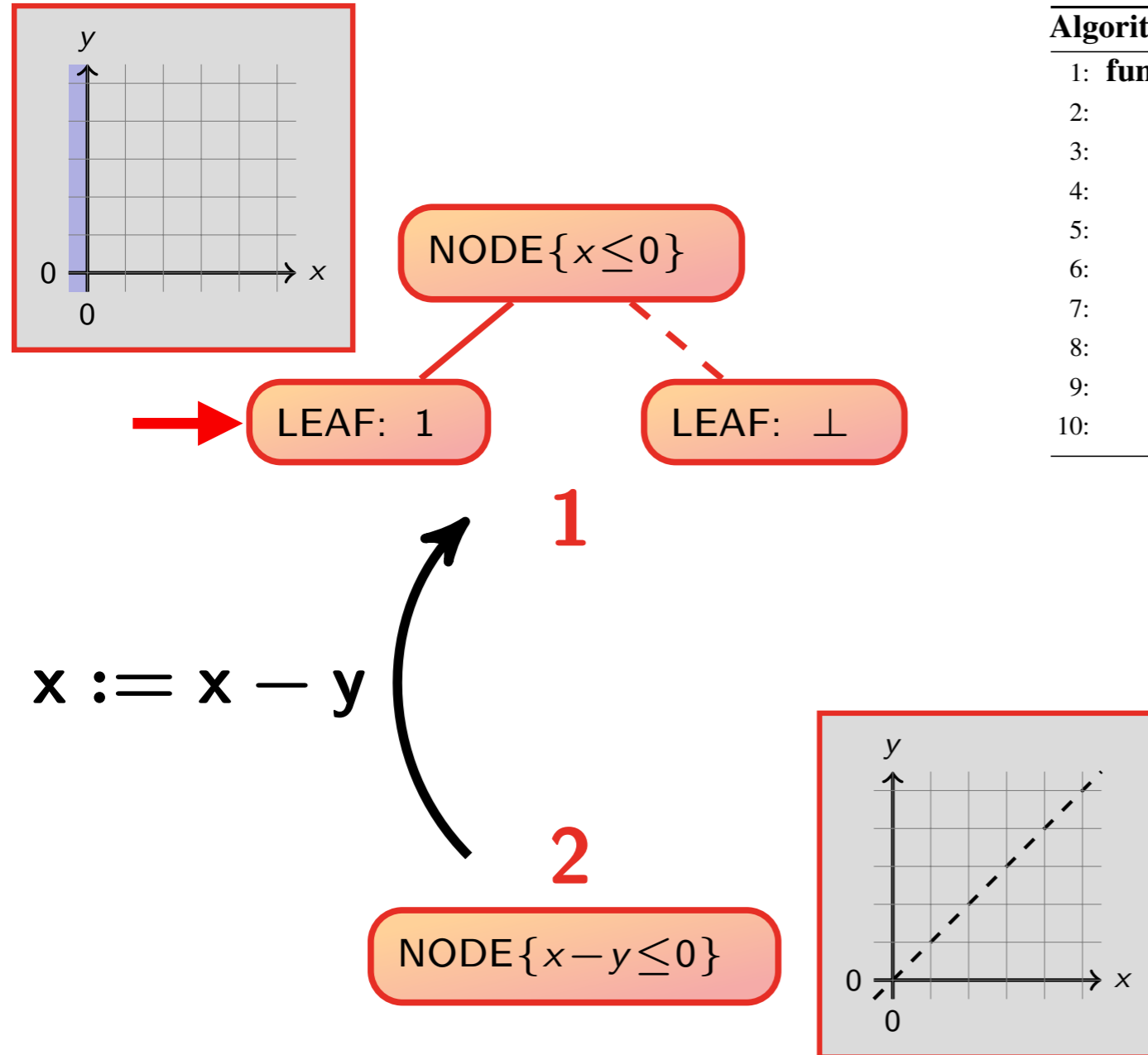
Algorithm 3 : Tree Assign

```

1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )      /*  $t \triangleq \text{LEAF} : f$  */
3:   else
4:      $C \leftarrow \text{ASSIGN}_L(t.c, x := a)$ 
5:     if ISEMPY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_{\top}$  ASSIGN( $t.r, x := a$ )
6:     else if ISUNSAT( $C$ ) then return ASSIGN( $t.r, x := a$ )
7:     else
8:        $l \leftarrow \text{AUGMENT}(\text{ASSIGN}(t.l, x := a), C)$ 
9:        $r \leftarrow \text{AUGMENT}(\text{ASSIGN}(t.r, x := a), C)$ 
10:    return NODE{ $l.c$ } :  $l; r$ 

```

Assignments



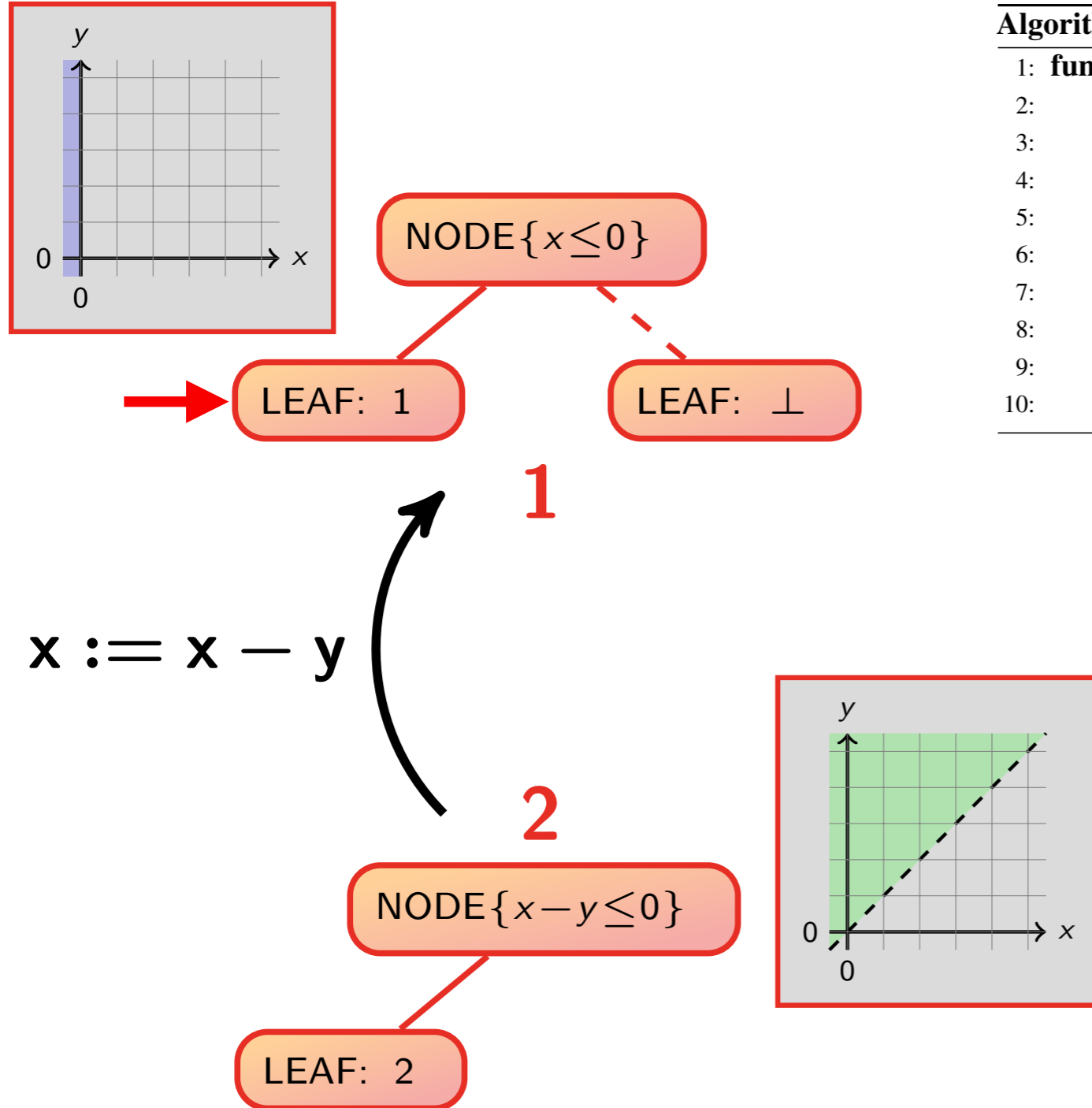
Algorithm 3 : Tree Assign

```

1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )      /*  $t \triangleq$  LEAF :  $f$  */
3:   else
4:      $C \leftarrow$  ASSIGNL( $t.c, x := a$ )
5:     if ISEMPY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_{\top}$  ASSIGN( $t.r, x := a$ )
6:     else if ISUNSAT( $C$ ) then return ASSIGN( $t.r, x := a$ )
7:     else
8:        $l \leftarrow$  AUGMENT(ASSIGN( $t.l, x := a$ ),  $C$ )
9:        $r \leftarrow$  AUGMENT(ASSIGN( $t.r, x := a$ ),  $C$ )
10:    return NODE $\{l.c\} : l; r$ 

```

Assignments



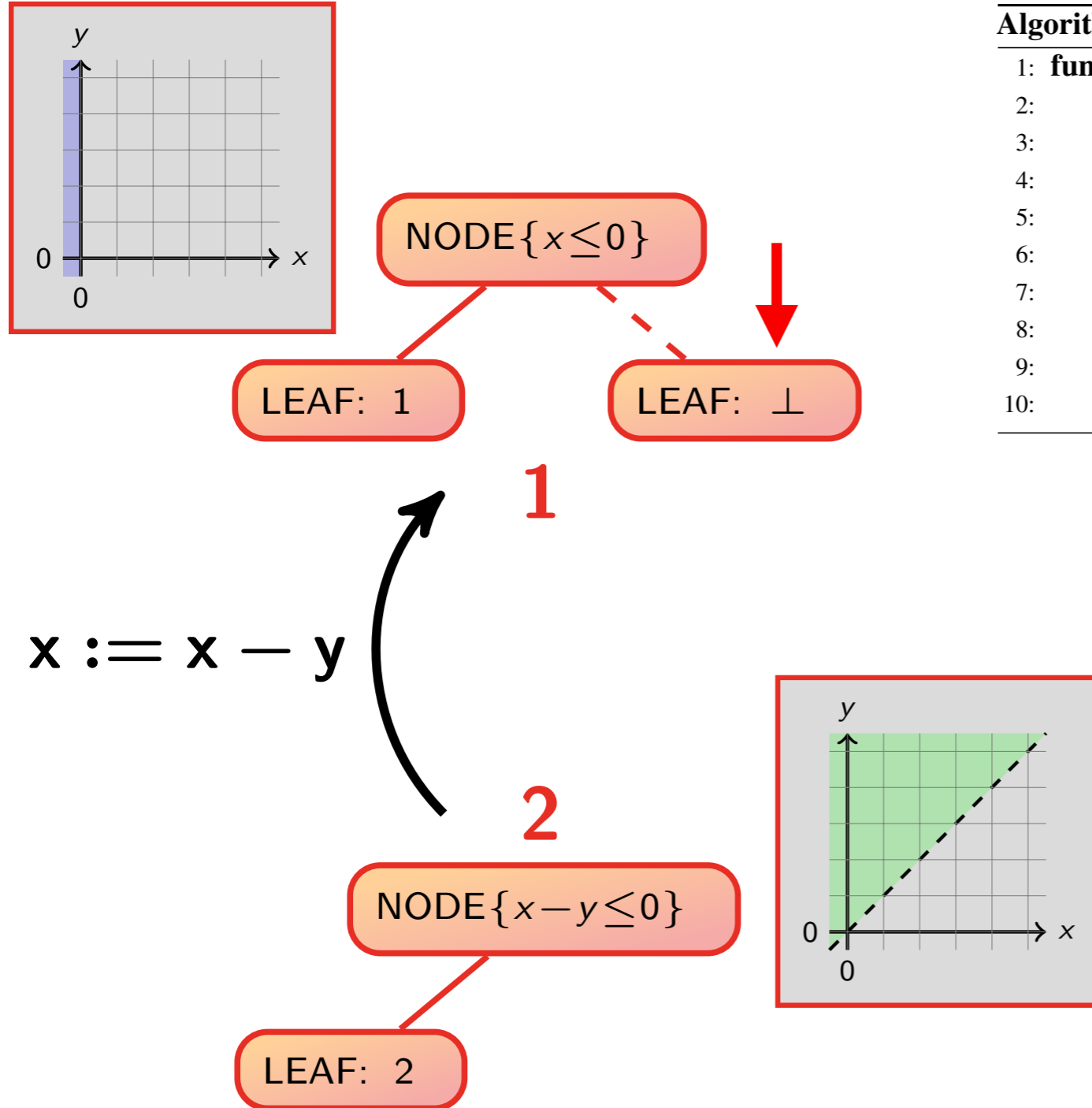
Algorithm 3 : Tree Assign

```

1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )      /*  $t \triangleq$  LEAF :  $f$  */
3:   else
4:      $C \leftarrow$  ASSIGNL( $t.c, x := a$ )
5:     if ISEMPY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_{\top}$  ASSIGN( $t.r, x := a$ )
6:     else if ISUNSAT( $C$ ) then return ASSIGN( $t.r, x := a$ )
7:     else
8:        $l \leftarrow$  AUGMENT(ASSIGN( $t.l, x := a$ ),  $C$ )
9:        $r \leftarrow$  AUGMENT(ASSIGN( $t.r, x := a$ ),  $C$ )
10:    return NODE $\{l.c\} : l; r$ 

```

Assignments



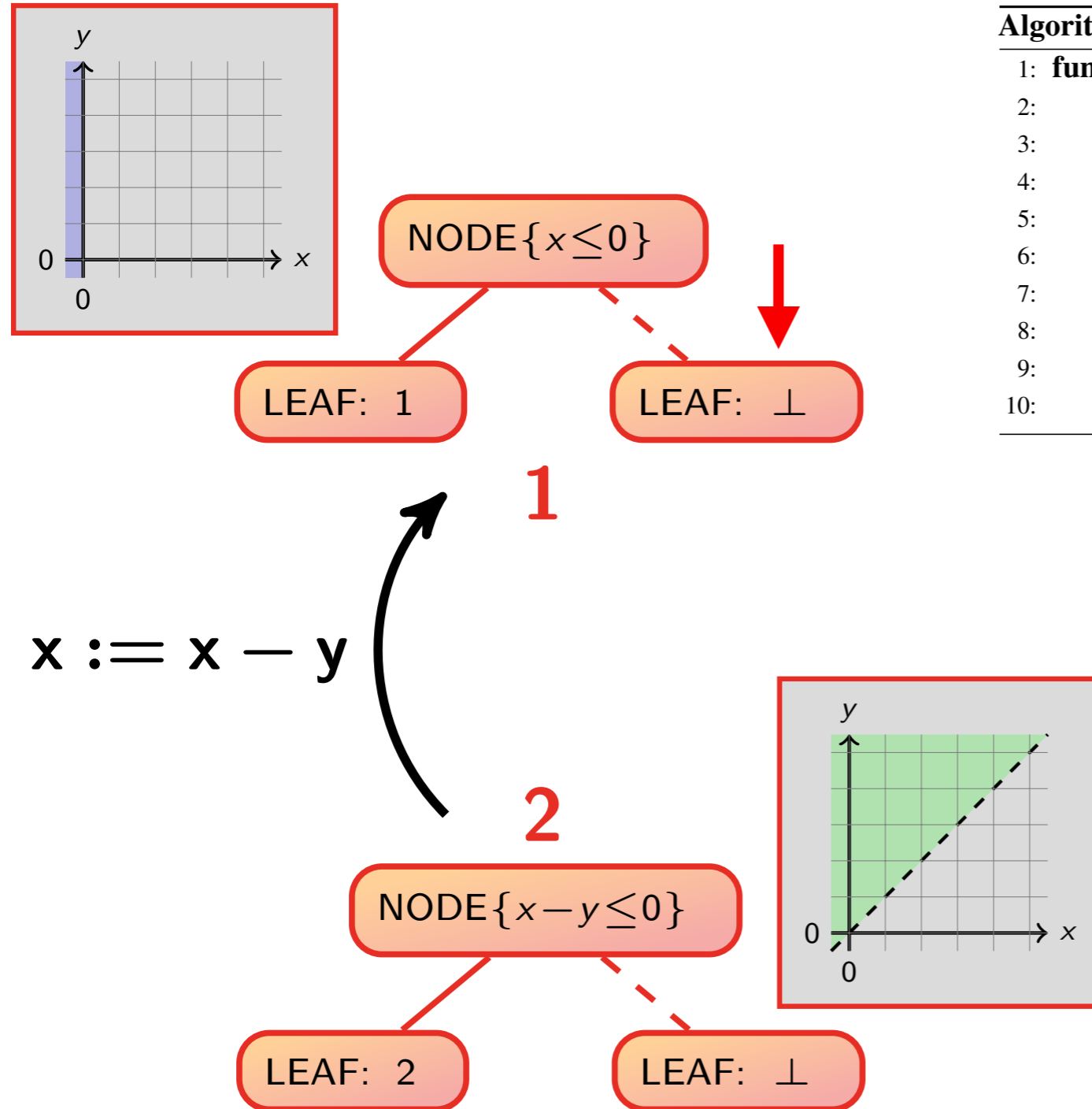
Algorithm 3 : Tree Assign

```

1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )      /*  $t \triangleq$  LEAF :  $f$  */
3:   else
4:      $C \leftarrow$  ASSIGNL( $t.c, x := a$ )
5:     if ISEMPY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_{\top}$  ASSIGN( $t.r, x := a$ )
6:     else if ISUNSAT( $C$ ) then return ASSIGN( $t.r, x := a$ )
7:     else
8:        $l \leftarrow$  AUGMENT(ASSIGN( $t.l, x := a$ ),  $C$ )
9:        $r \leftarrow$  AUGMENT(ASSIGN( $t.r, x := a$ ),  $C$ )
10:      return NODE{ $l.c$ } :  $l; r$ 

```

Assignments



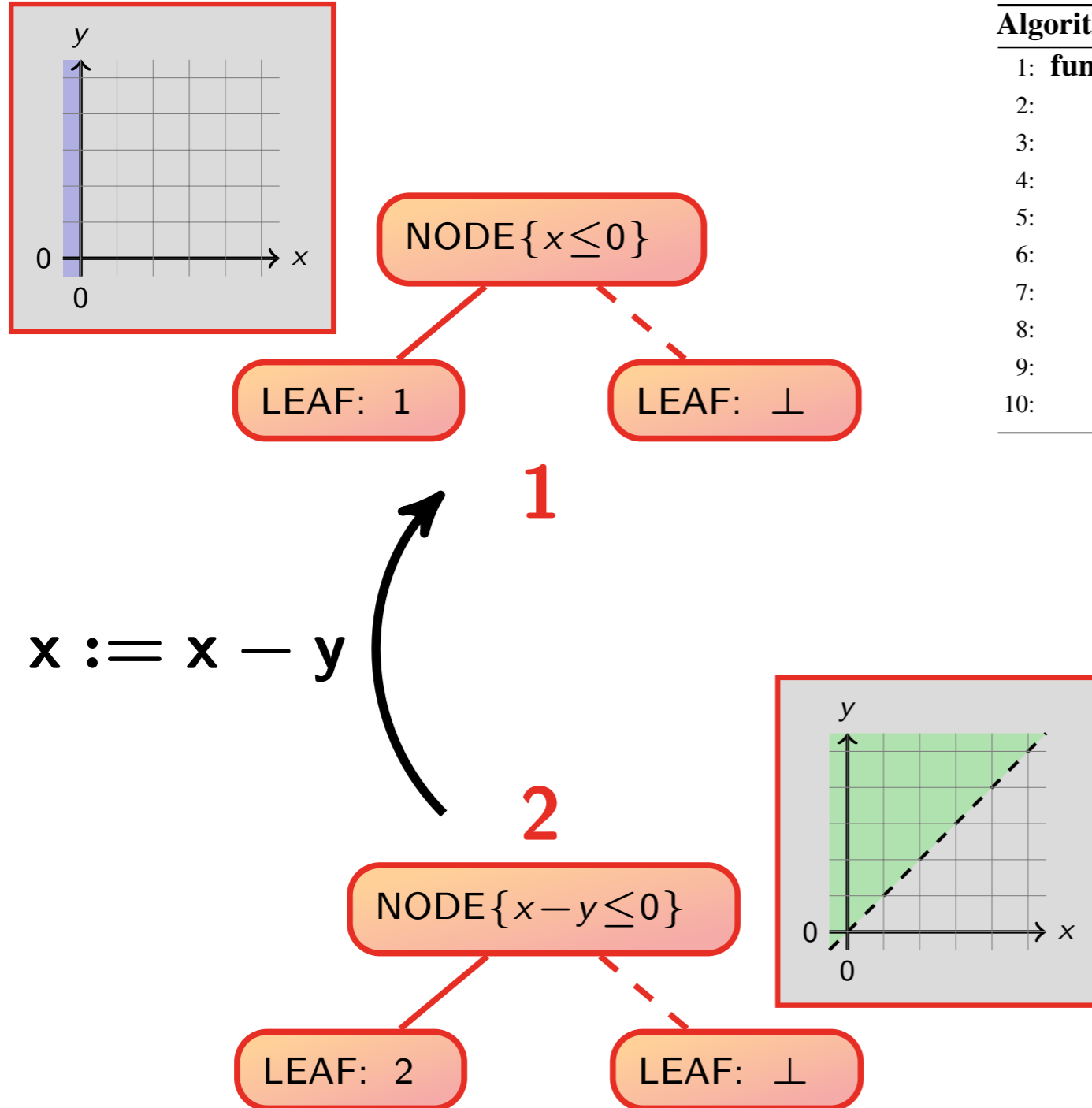
Algorithm 3 : Tree Assign

```

1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )      /*  $t \triangleq$  LEAF :  $f$  */
3:   else
4:      $C \leftarrow$  ASSIGNL( $t.c, x := a$ )
5:     if ISEMPY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_{\top}$  ASSIGN( $t.r, x := a$ )
6:     else if ISUNSAT( $C$ ) then return ASSIGN( $t.r, x := a$ )
7:     else
8:        $l \leftarrow$  AUGMENT(ASSIGN( $t.l, x := a$ ),  $C$ )
9:        $r \leftarrow$  AUGMENT(ASSIGN( $t.r, x := a$ ),  $C$ )
10:      return NODE $\{l.c\} : l; r$ 

```

Assignments



Algorithm 3 : Tree Assign

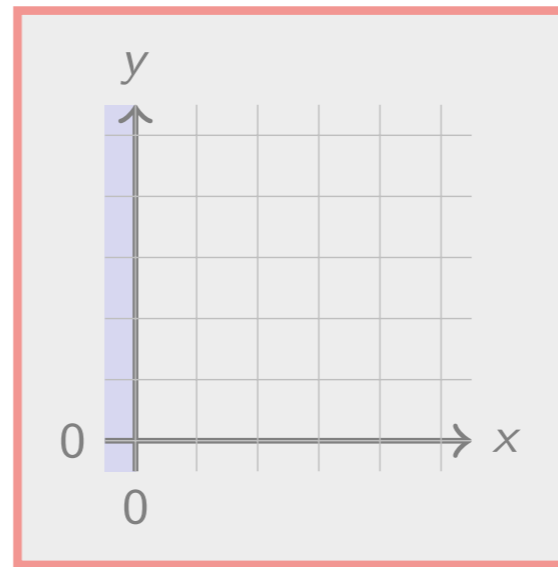
```

1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )      /*  $t \triangleq$  LEAF :  $f$  */
3:   else
4:      $C \leftarrow$  ASSIGNL( $t.c, x := a$ )
5:     if ISEMPY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_{\top}$  ASSIGN( $t.r, x := a$ )
6:     else if ISUNSAT( $C$ ) then return ASSIGN( $t.r, x := a$ )
7:     else
8:        $l \leftarrow$  AUGMENT(ASSIGN( $t.l, x := a$ ),  $C$ )
9:        $r \leftarrow$  AUGMENT(ASSIGN( $t.r, x := a$ ),  $C$ )
10:    return NODE{ $l.c$ } :  $l; r$ 

```

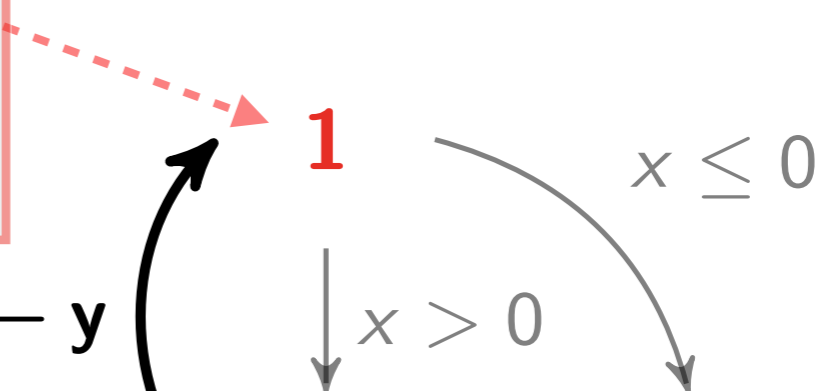
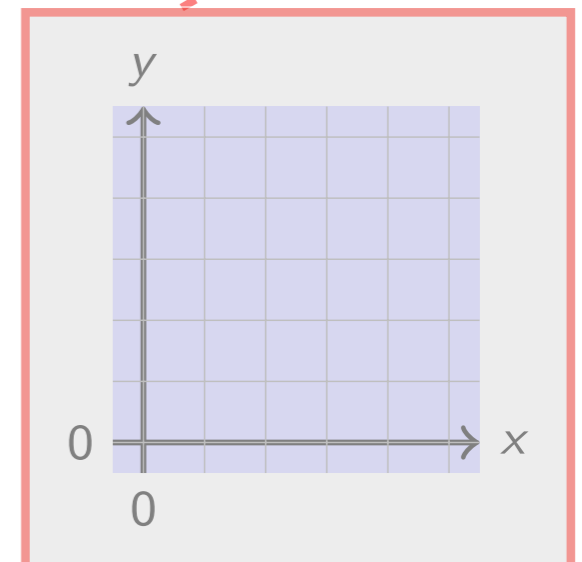
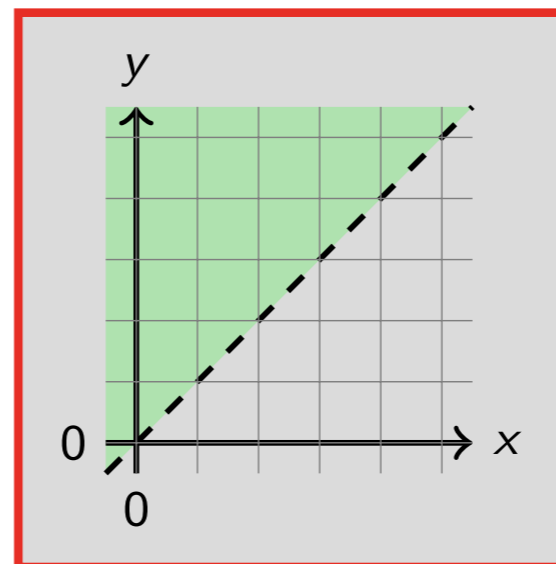
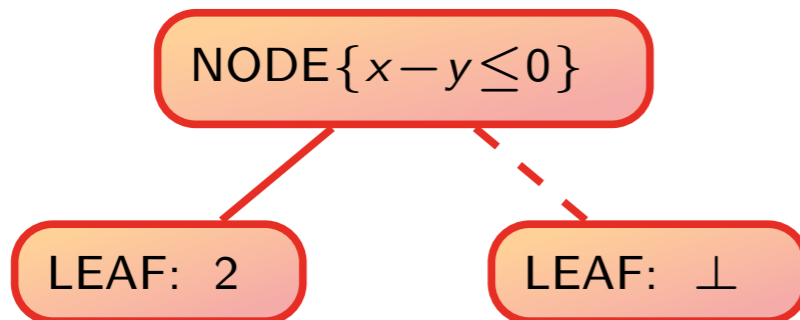
Example

```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



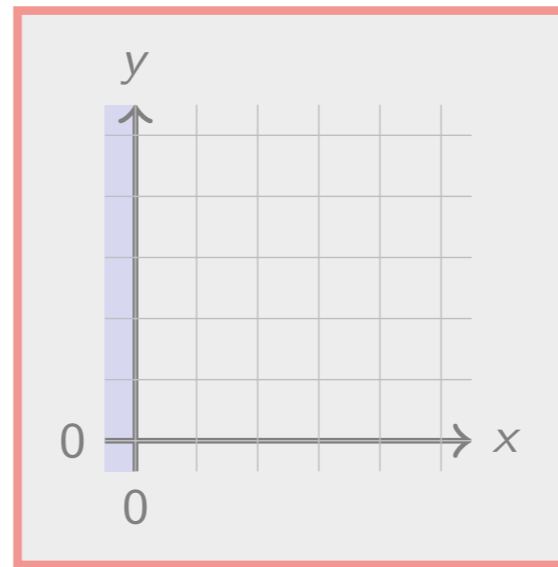
$$x := x - y$$

we have taken $x := x - y$ into account and we have 2 steps to termination



Example

```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



$$x := x - y$$

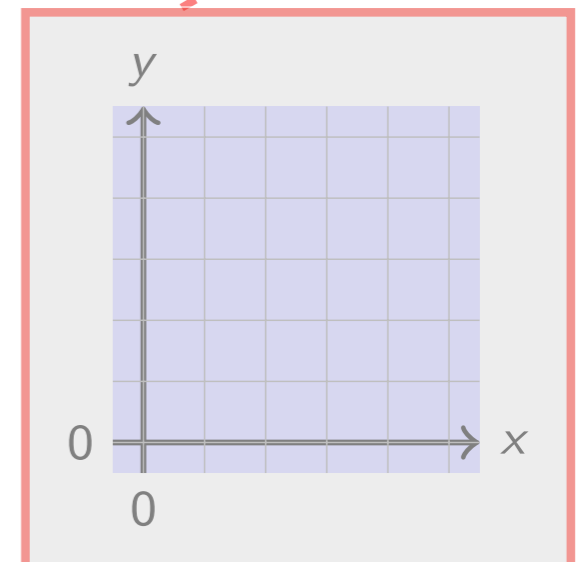
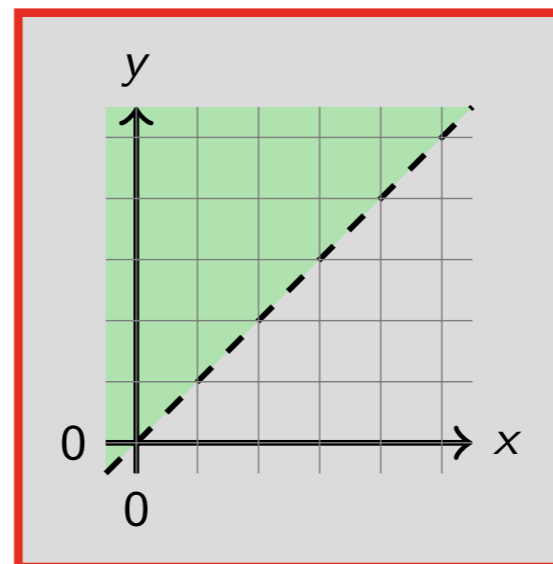
1

$$x \leq 0$$

x > 0

2

3



NODE { $x - y \leq 0$ }

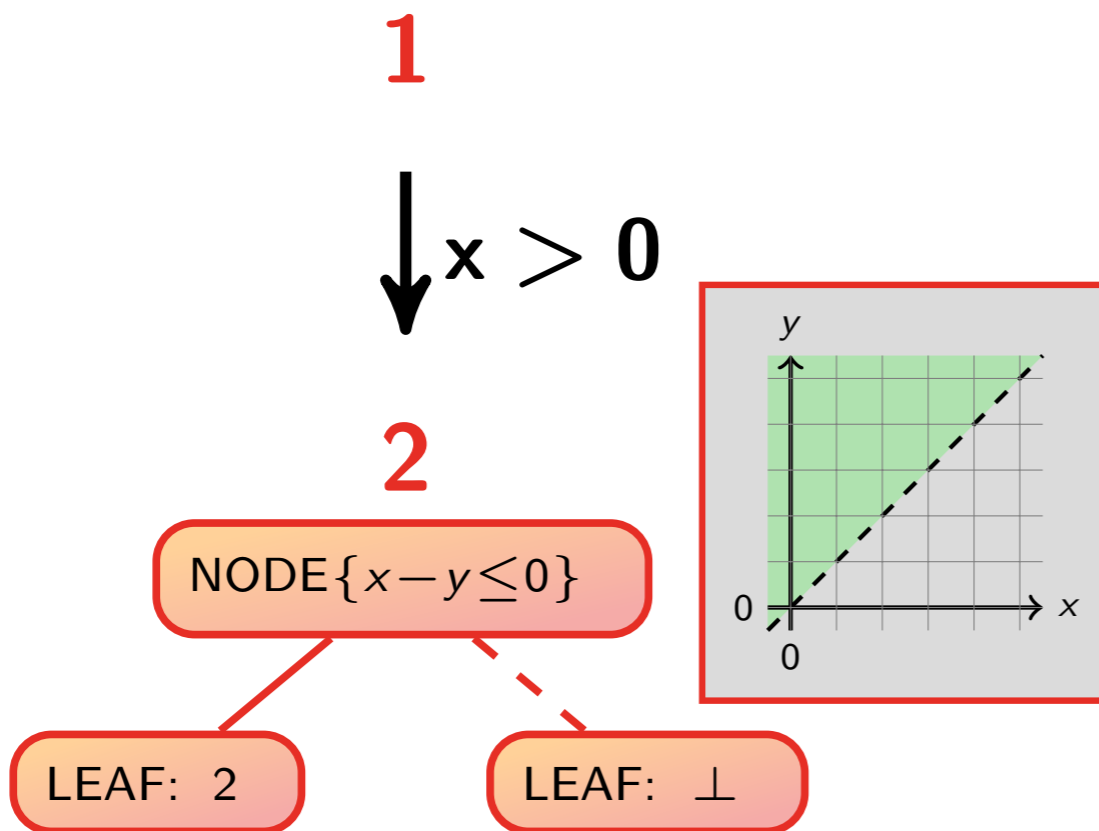
LEAF: 2

LEAF: \perp

Tests

Algorithm 4 : Tree Filter

```
1: function FILTER-AUX( $t, c$ )  
2:   if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq$  LEAF :  $f$  */  
3:   else return NODE{ $t.c$ } : FILTER-AUX( $t.l, c$ ); FILTER-AUX( $t.r, c$ )  
  
4: function FILTER( $t, c$ )  
5:    $C \leftarrow$  FILTERL( $c$ )  
6:   return AUGMENT(FILTER-AUX( $t, c$ ),  $C$ )
```



Tests

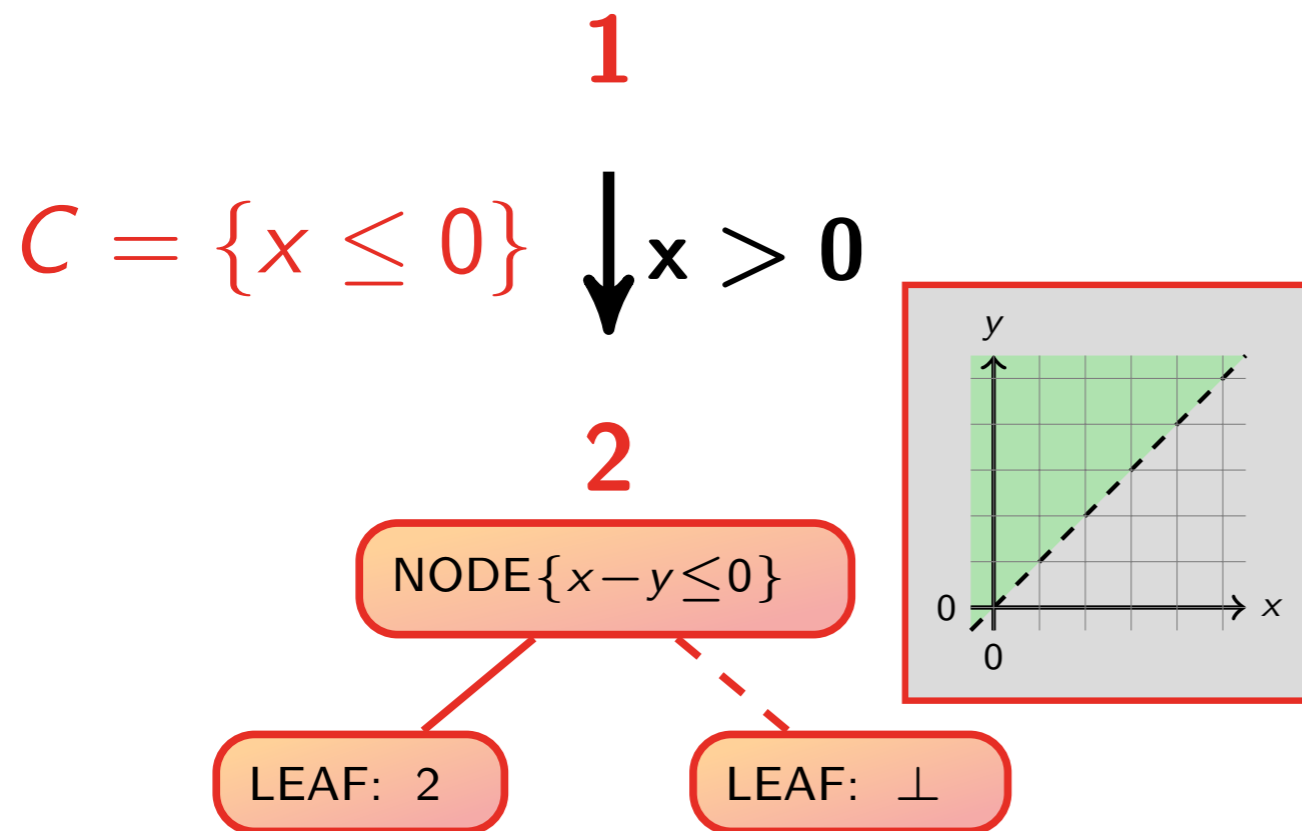
Algorithm 4 : Tree Filter

```

1: function FILTER-AUX( $t, c$ )
2:   if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq$  LEAF :  $f$  */
3:   else return NODE{ $t.c$ } : FILTER-AUX( $t.l, c$ ); FILTER-AUX( $t.r, c$ )

4: function FILTER( $t, c$ )
5:    $C \leftarrow$  FILTERL( $c$ )
6:   return AUGMENT(FILTER-AUX( $t, c$ ),  $C$ )

```



Tests

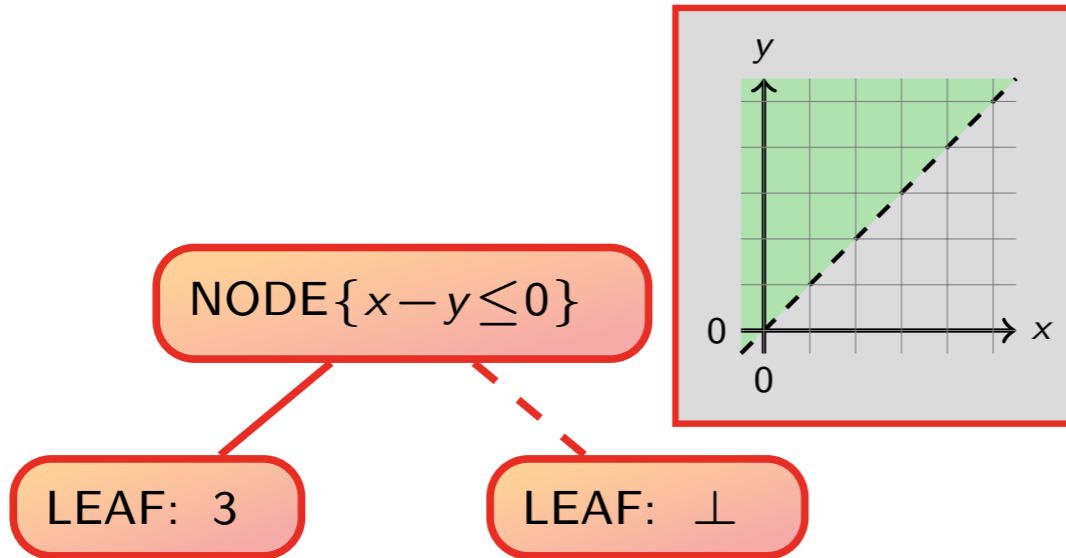
thm 4 : Tree Filter

```

function FILTER-AUX( $t, c$ )
  if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq$  LEAF :  $f$  */
  else return NODE{ $t.c$ } : FILTER-AUX( $t.l, c$ ); FILTER-AUX( $t.r, c$ )
  
```

```

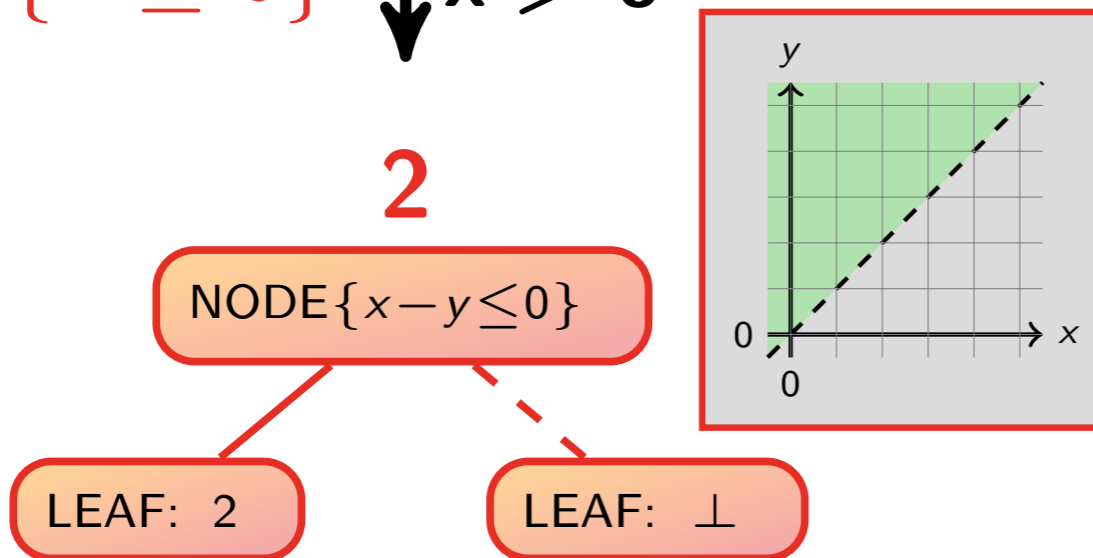
function FILTER( $t, c$ )
   $C \leftarrow$  FILTERL( $c$ )
  return AUGMENT(FILTER-AUX( $t, c$ ),  $C$ )
  
```



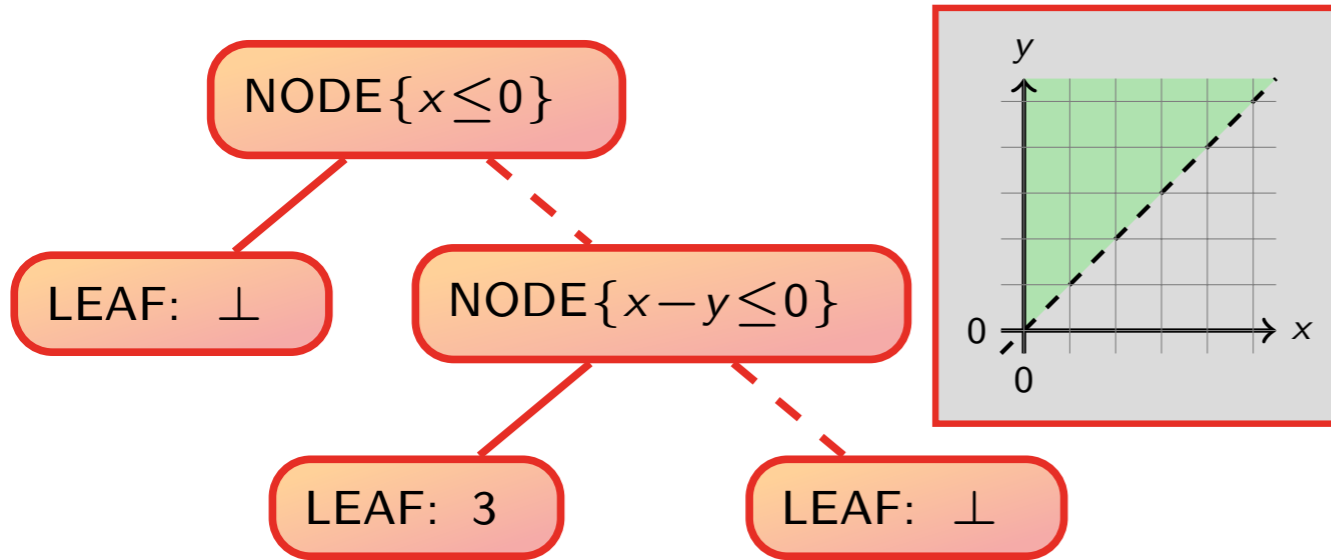
1

$C = \{x \leq 0\}$ $\downarrow x > 0$

2



Tests



thm 4 : Tree Filter

```

function FILTER-AUX(t,c)
  if ISLEAF(t) then return LEAF : FILTERF(f, c)           /* t ≜ LEAF : f */
  else return NODE{t.c} : FILTER-AUX(t.l, c); FILTER-AUX(t.r, c)
  
```

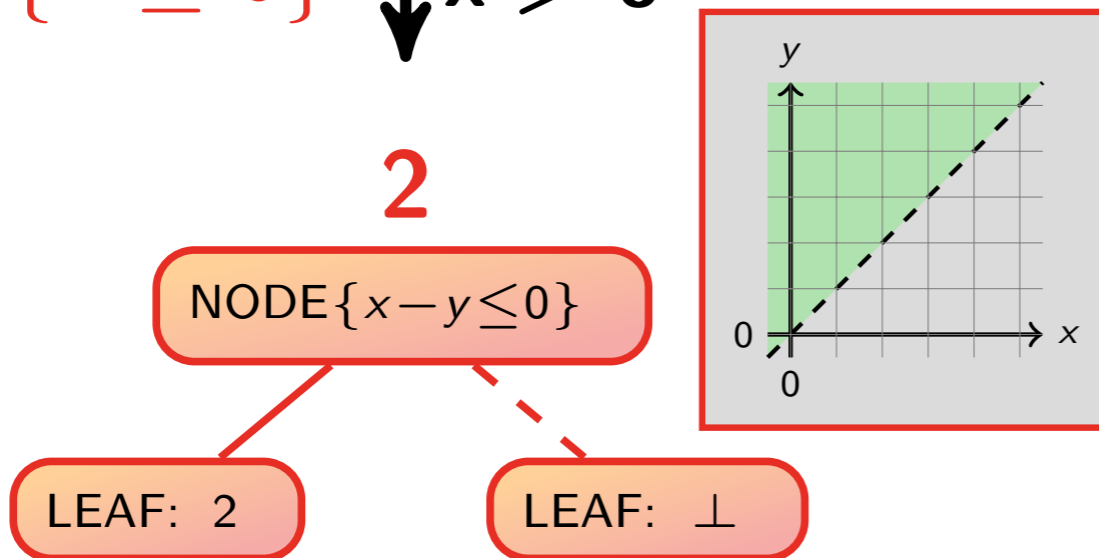
```

function FILTER(t,c)
  C ← FILTERL(c)
  return AUGMENT(FILTER-AUX(t, c), C)
  
```

1

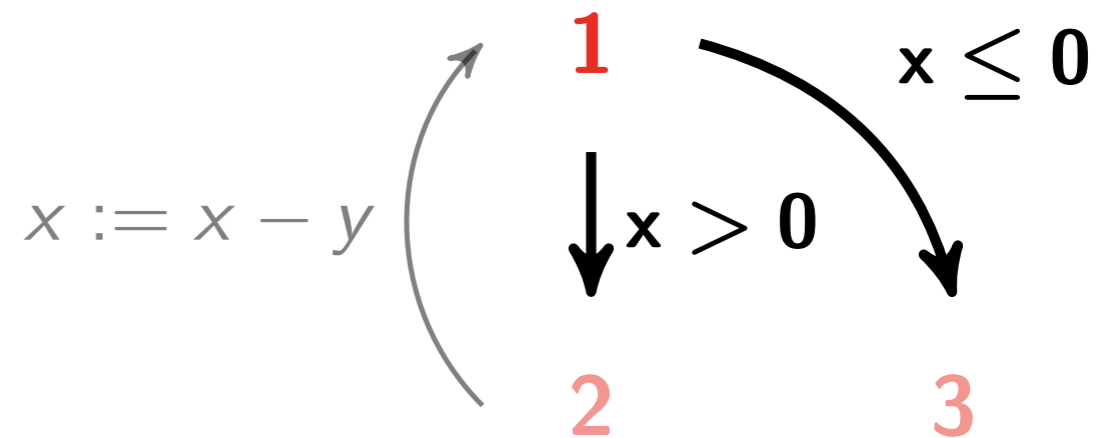
$C = \{x \leq 0\}$ $\downarrow x > 0$

2



Example

```
int : x, y  
while 1(x > 0) do  
  2x := x - y  
od3
```



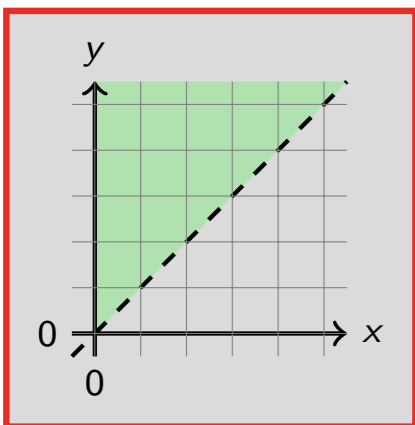
Join

Algorithm 1 : Tree Unification

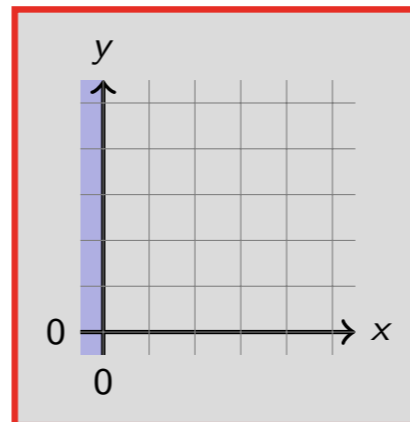
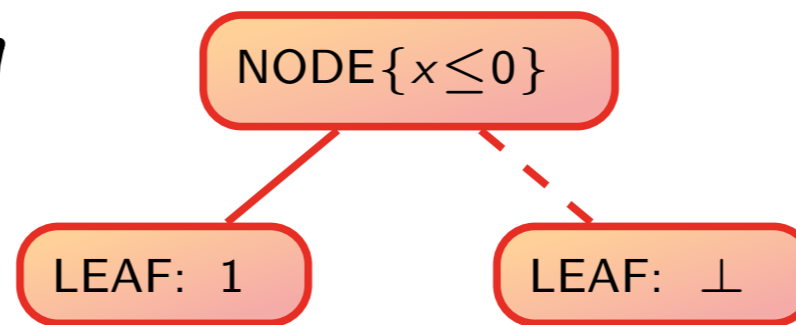
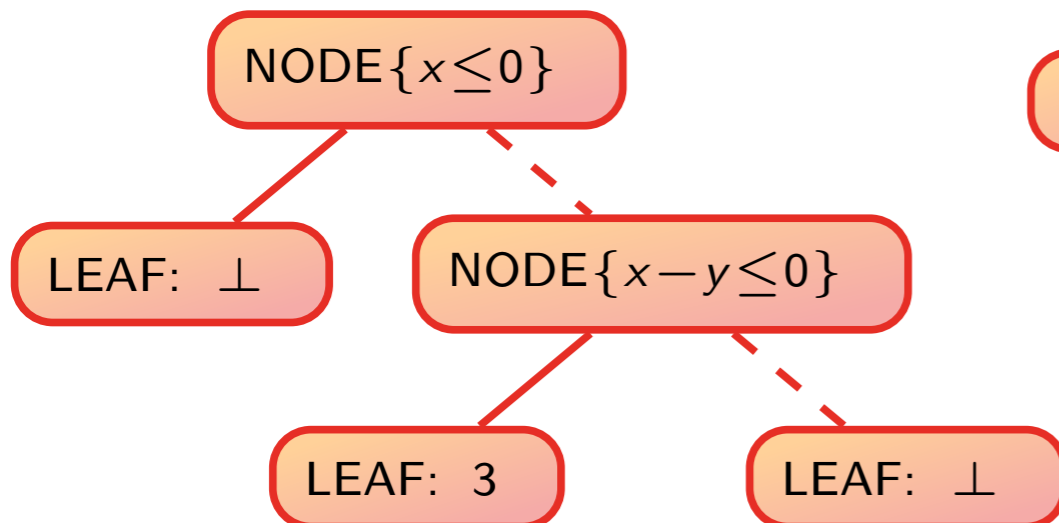
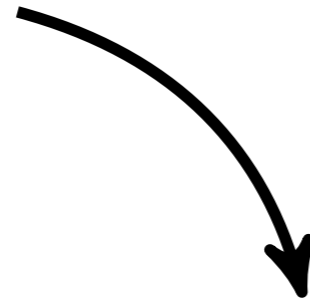
```

1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE{ $t_2.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_1.c$ } :  $l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )

```



1



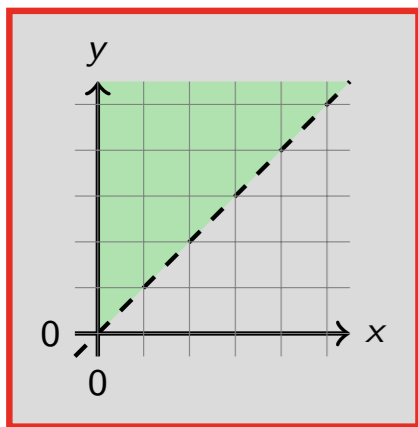
Join

Algorithm 1 : Tree Unification

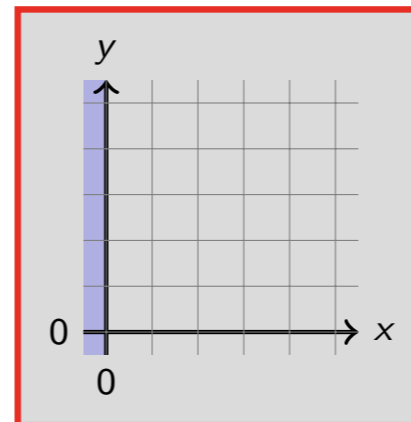
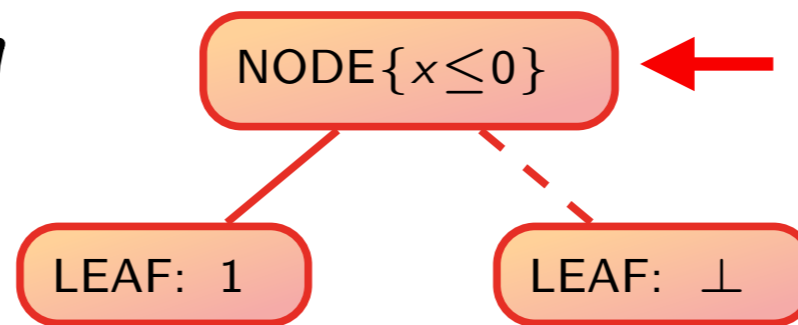
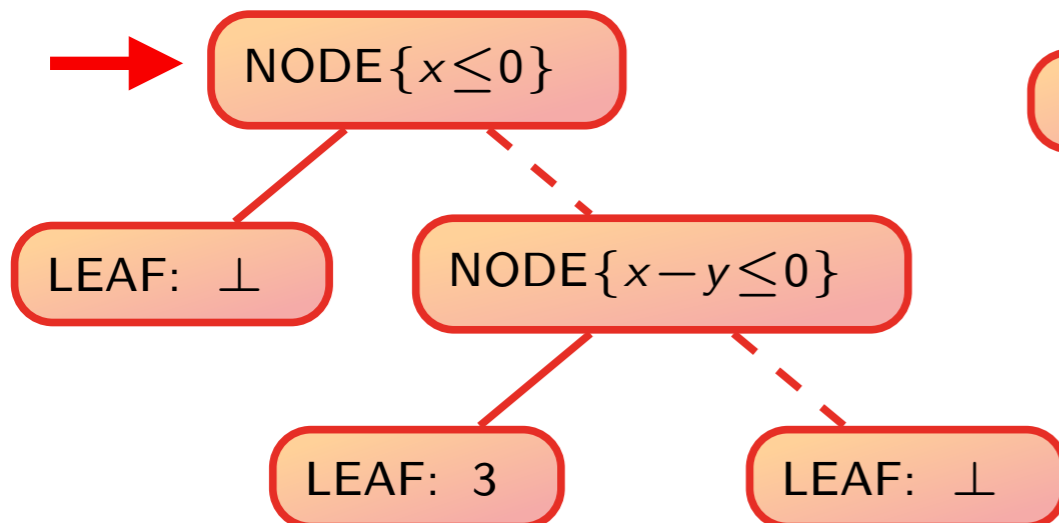
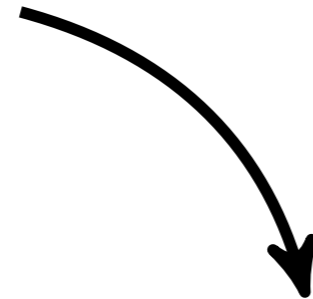
```

1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE{ $t_2.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_1.c$ } :  $l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )

```

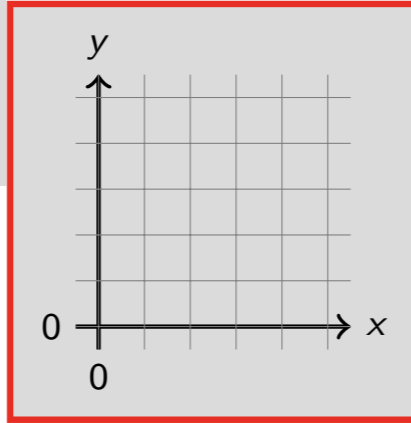


1



Join

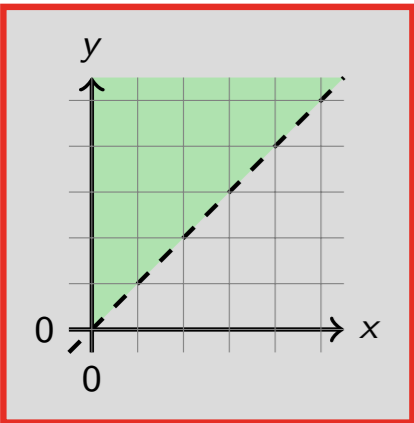
NODE $\{x \leq 0\}$



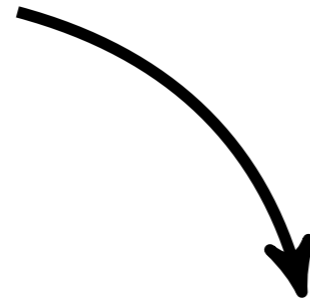
Algorithm 1 : Tree Unification

```

1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE $\{t_2.c\} : l_1; r_1$ , NODE $\{t_2.c\} : l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE $\{t_1.c\} : l_1; r_1$ , NODE $\{t_1.c\} : l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE $\{t_1.c\} : l_1; r_1$ , NODE $\{t_2.c\} : l_2; r_2$ )
  
```



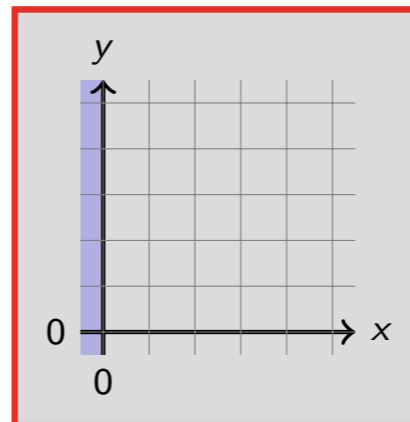
1



NODE $\{x \leq 0\}$

LEAF: 1

LEAF: \perp



NODE $\{x \leq 0\}$

LEAF: \perp

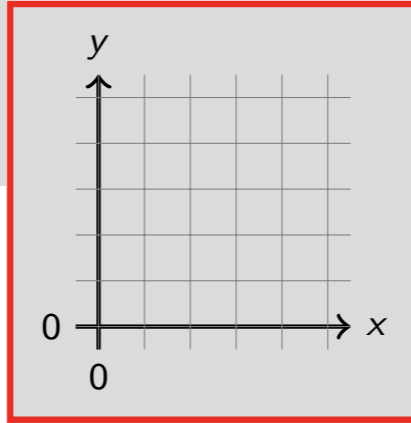
NODE $\{x - y \leq 0\}$

LEAF: 3

LEAF: \perp

Join

NODE $\{x \leq 0\}$



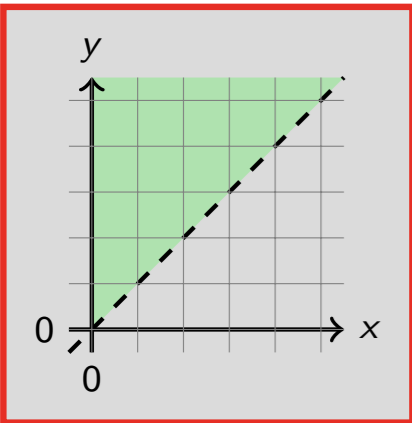
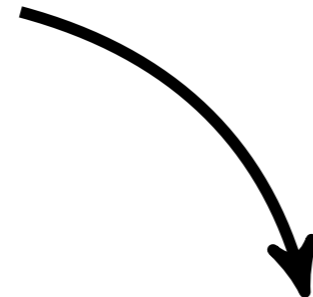
Algorithm 1 : Tree Unification

```

1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE $\{t_2.c\} : l_1; r_1$ , NODE $\{t_2.c\} : l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE $\{t_1.c\} : l_1; r_1$ , NODE $\{t_1.c\} : l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE $\{t_1.c\} : l_1; r_1$ , NODE $\{t_2.c\} : l_2; r_2$ )

```

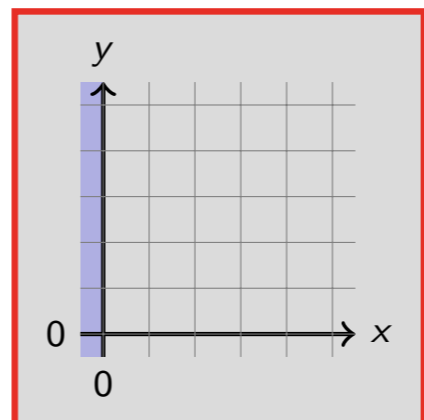
1



NODE $\{x \leq 0\}$

LEAF: 1

LEAF: \perp



NODE $\{x \leq 0\}$

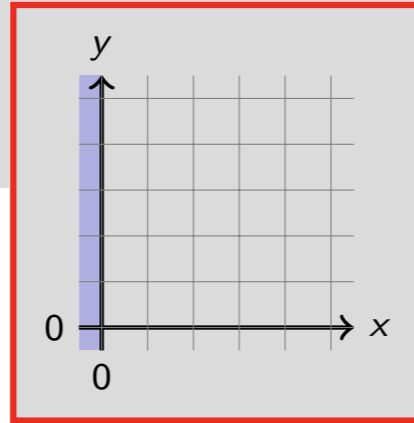
LEAF: \perp

NODE $\{x - y \leq 0\}$

LEAF: 3

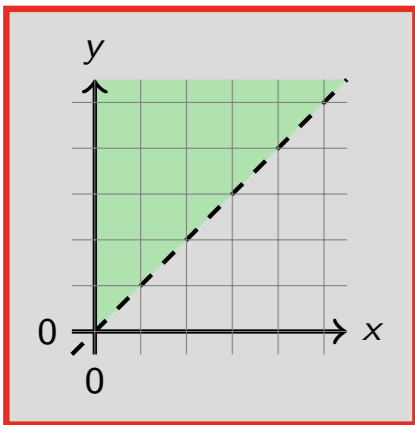
LEAF: \perp

Join

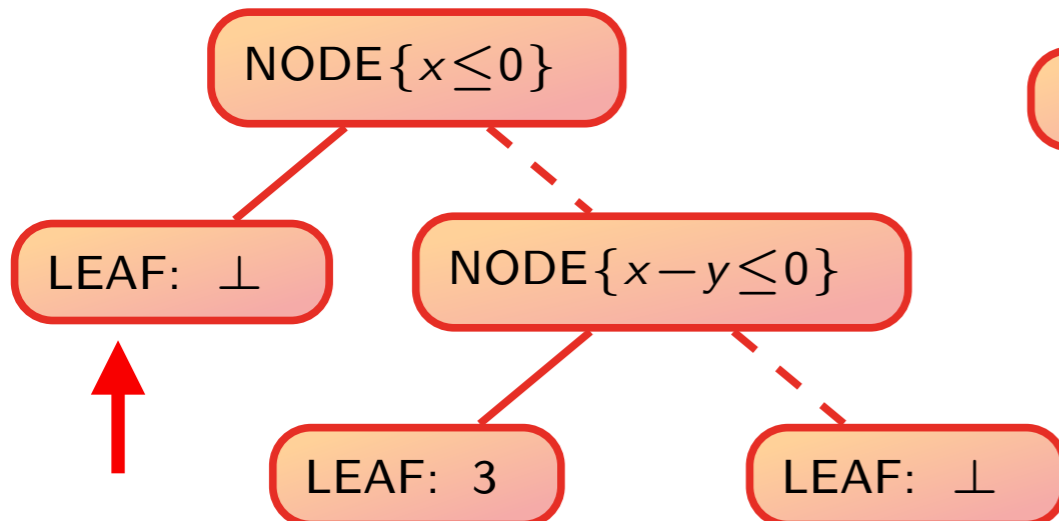
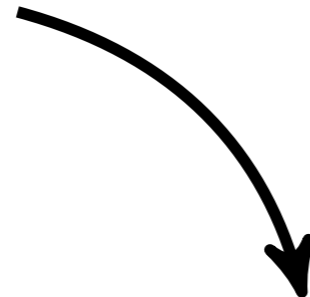


NODE $\{x \leq 0\}$

LEAF: 1



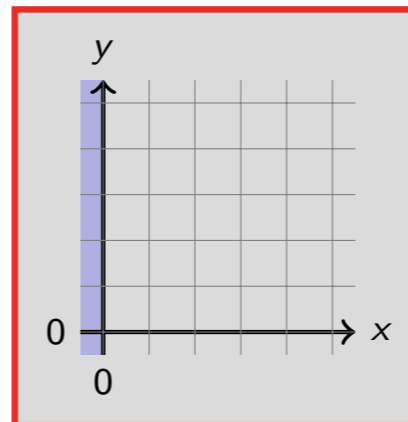
1



NODE $\{x \leq 0\}$

LEAF: 1

LEAF: \perp



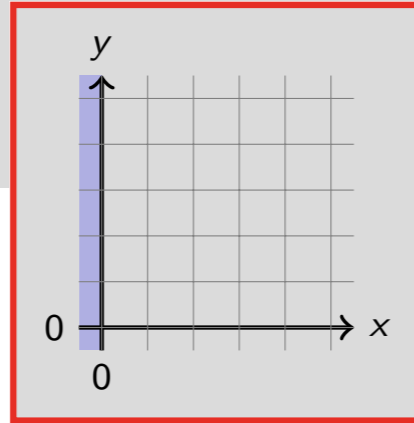
Algorithm 1 : Tree Unification

```

1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE $\{t_2.c\} : l_1; r_1$ , NODE $\{t_2.c\} : l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE $\{t_1.c\} : l_1; r_1$ , NODE $\{t_1.c\} : l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE $\{t_1.c\} : l_1; r_1$ , NODE $\{t_2.c\} : l_2; r_2$ )

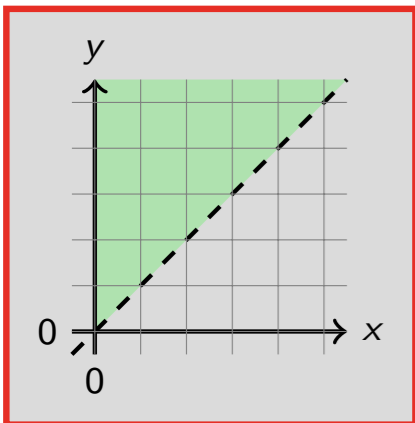
```

Join

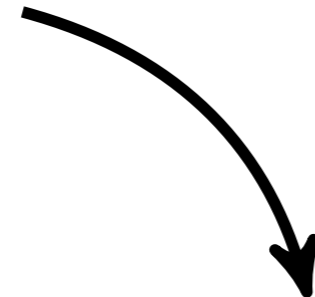


NODE $\{x \leq 0\}$

LEAF: 1



1



NODE $\{x \leq 0\}$

LEAF: \perp

NODE $\{x - y \leq 0\}$

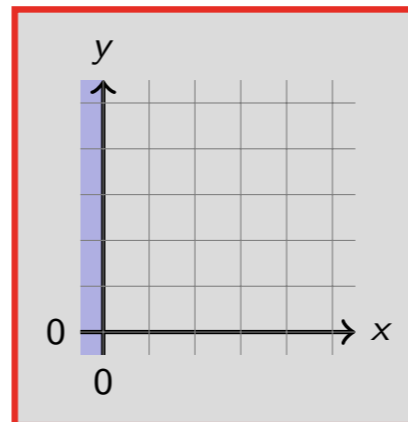
LEAF: 3

LEAF: \perp

NODE $\{x \leq 0\}$

LEAF: 1

LEAF: \perp



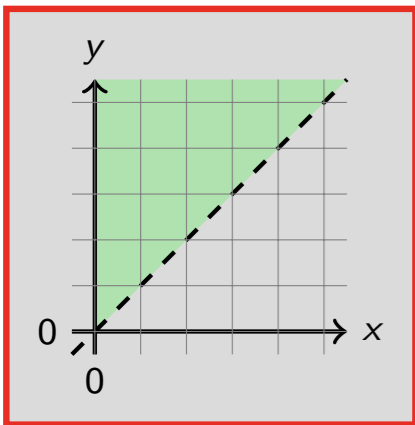
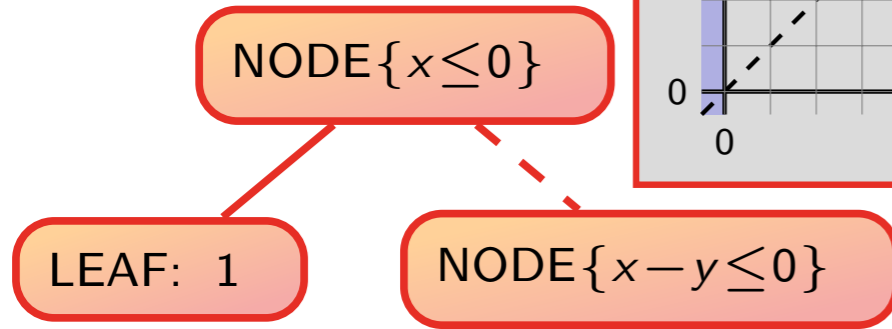
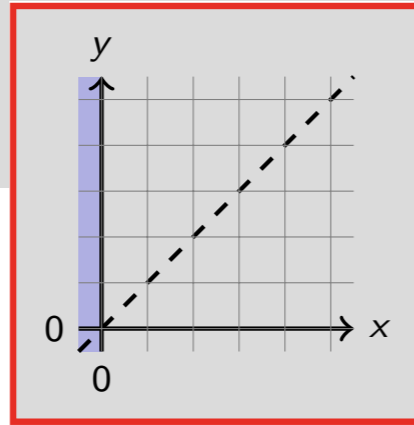
Algorithm 1 : Tree Unification

```

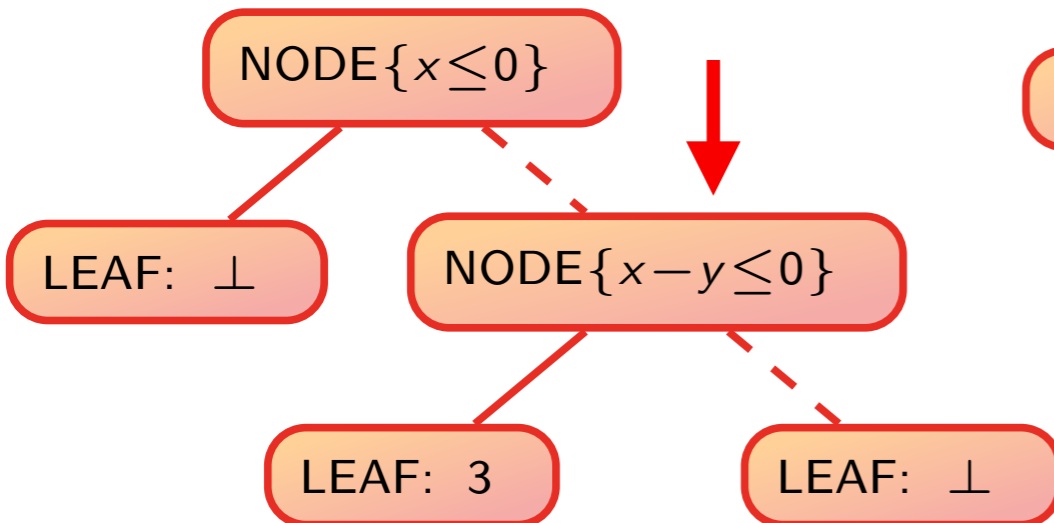
1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE $\{t_2.c\} : l_1; r_1$ , NODE $\{t_2.c\} : l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE $\{t_1.c\} : l_1; r_1$ , NODE $\{t_1.c\} : l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE $\{t_1.c\} : l_1; r_1$ , NODE $\{t_2.c\} : l_2; r_2$ )

```

Join



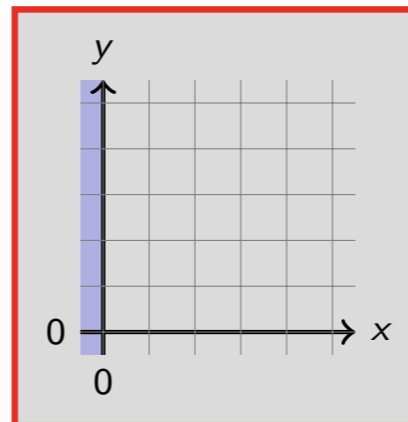
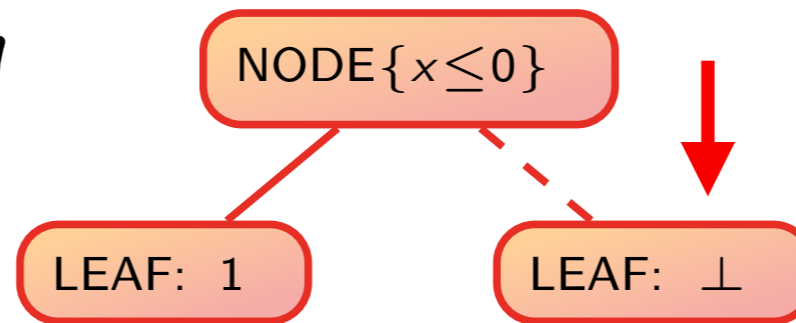
1



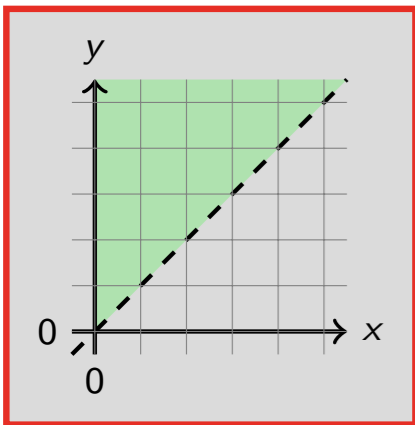
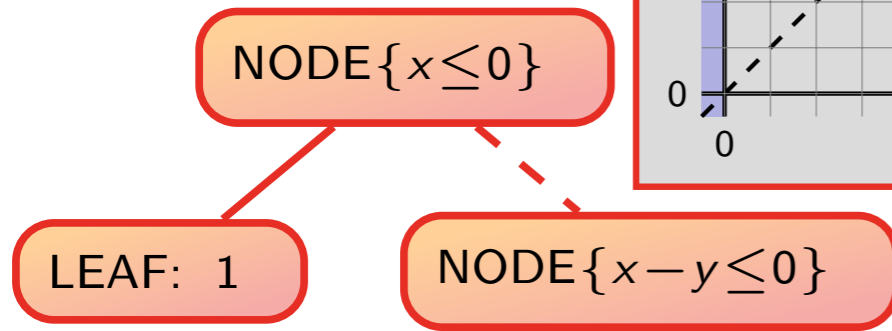
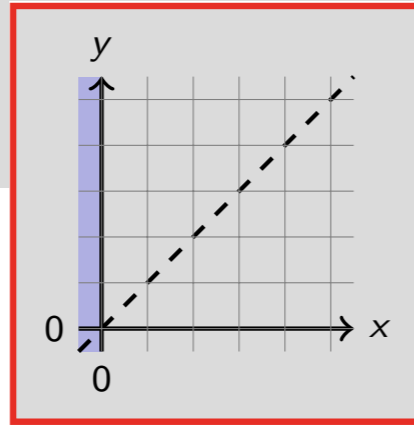
Algorithm 1 : Tree Unification

```

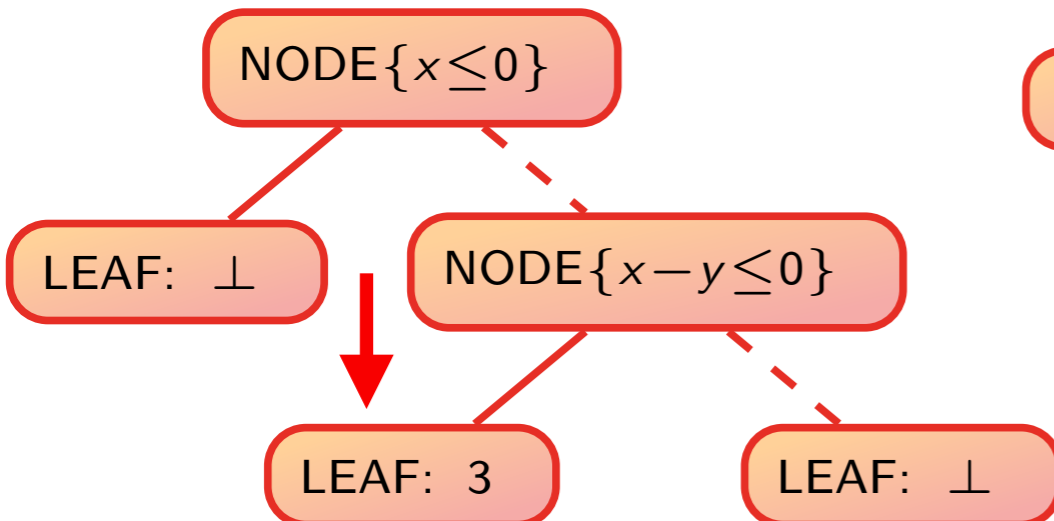
1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE{ $t_2.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_1.c$ } :  $l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
  
```



Join



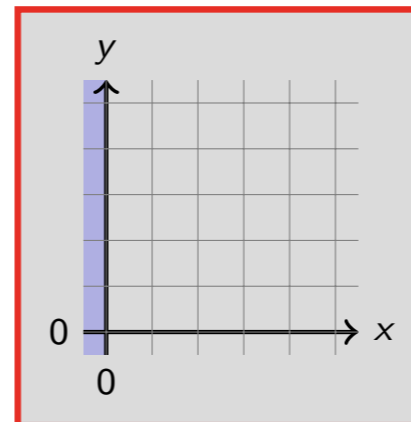
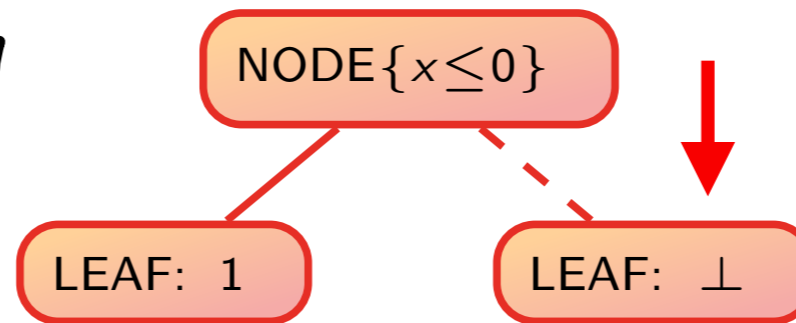
1



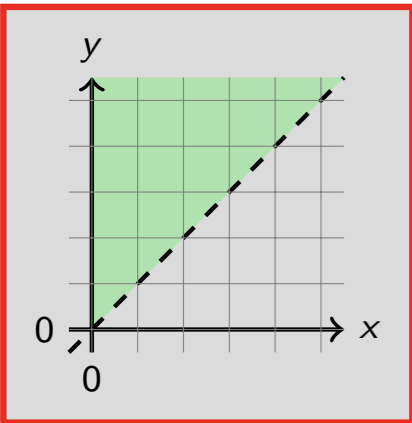
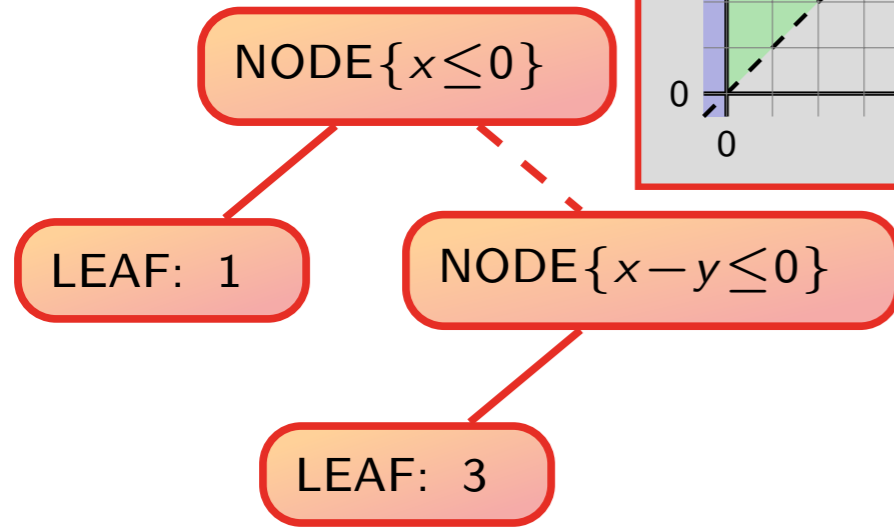
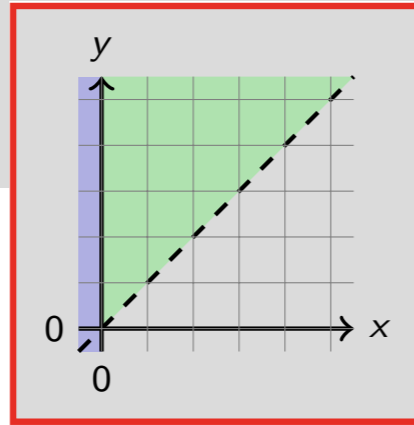
Algorithm 1 : Tree Unification

```

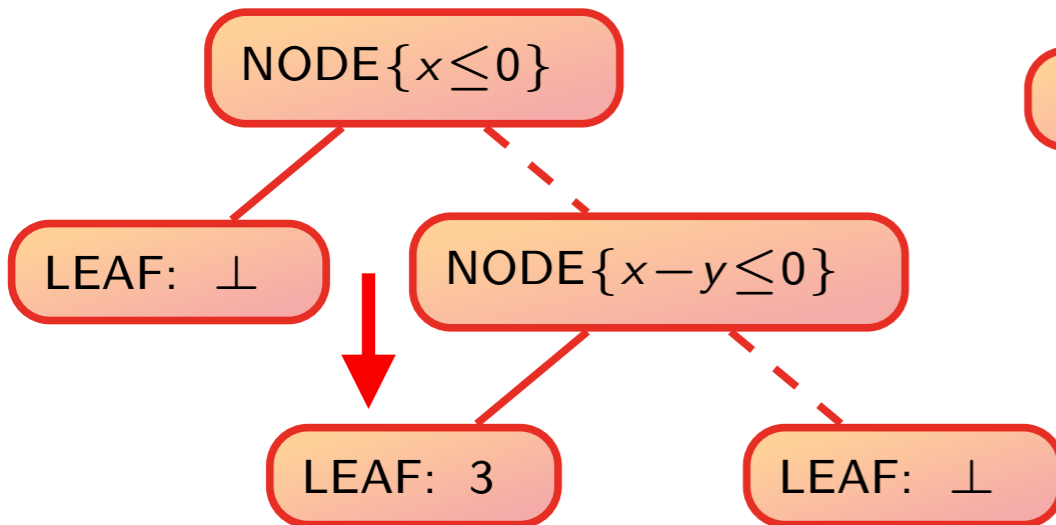
1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE{ $t_2.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_1.c$ } :  $l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
  
```



Join



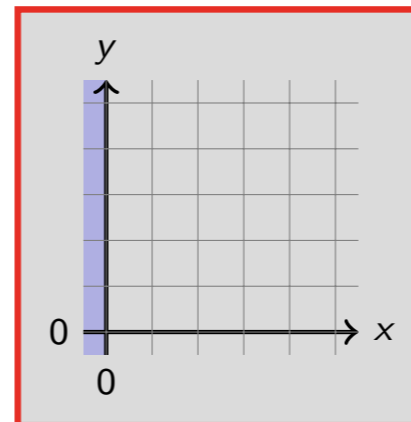
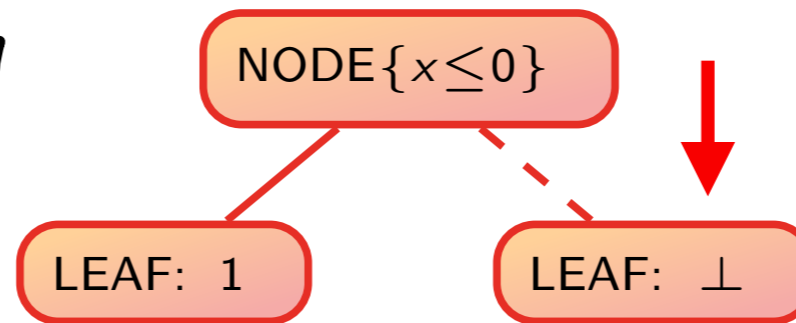
1



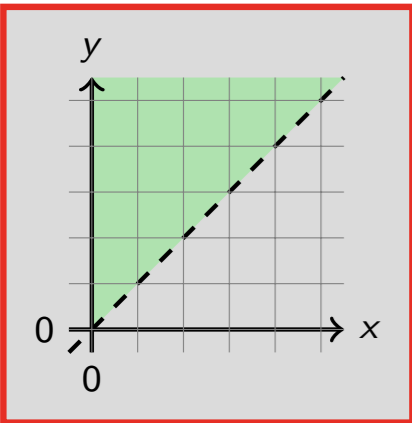
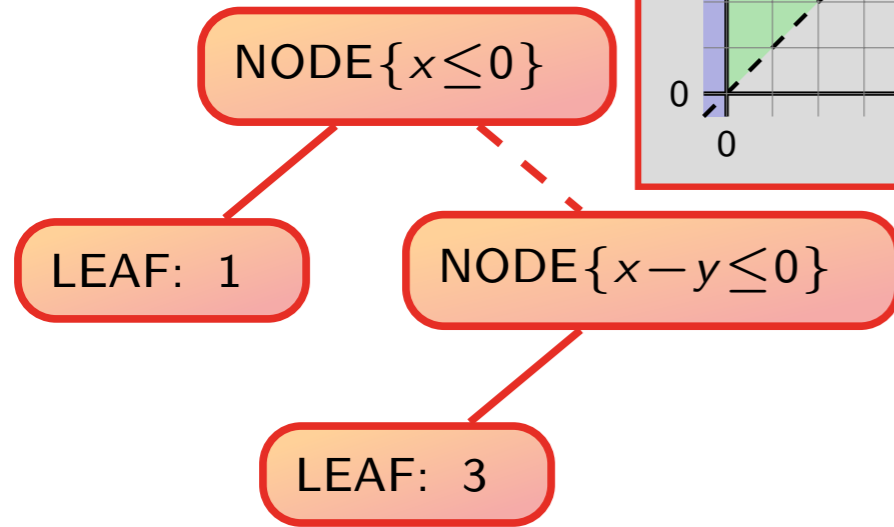
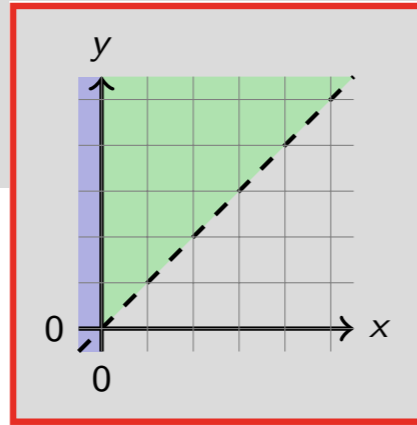
Algorithm 1 : Tree Unification

```

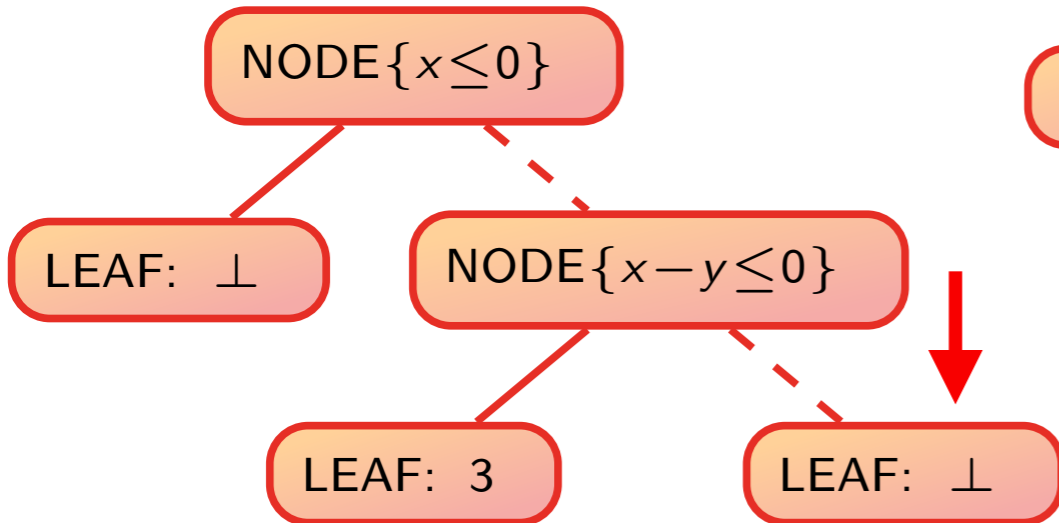
1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE{ $t_2.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_1.c$ } :  $l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
  
```



Join



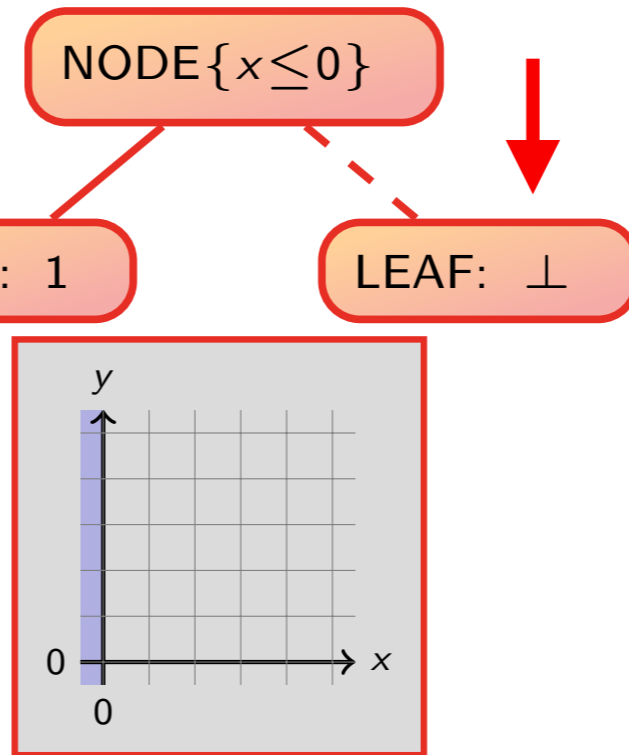
1



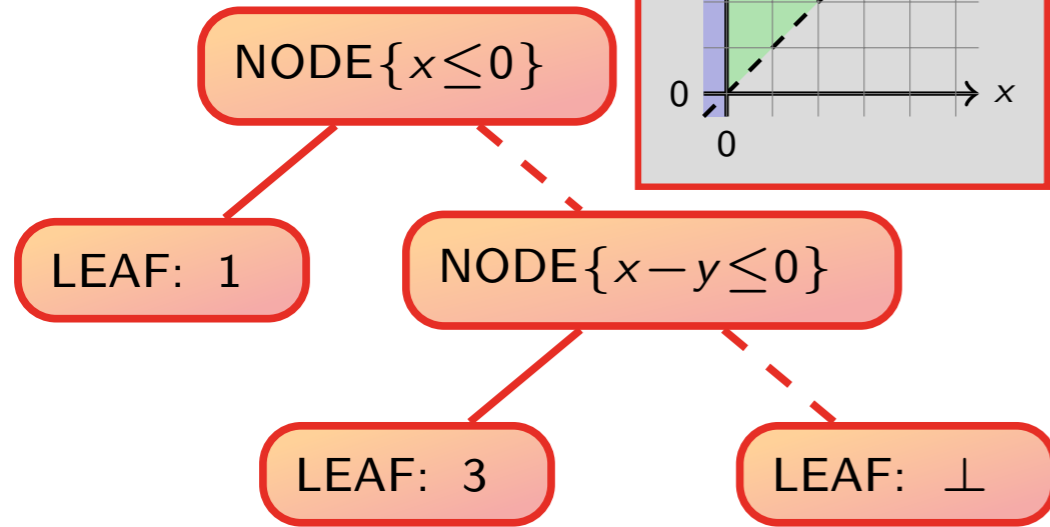
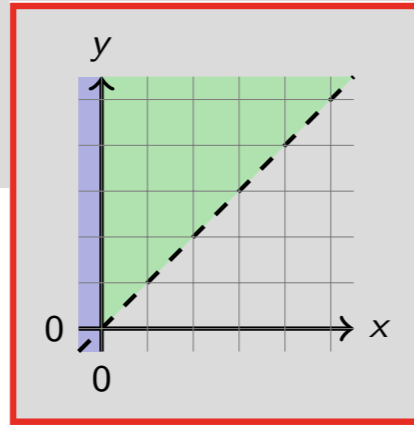
Algorithm 1 : Tree Unification

```

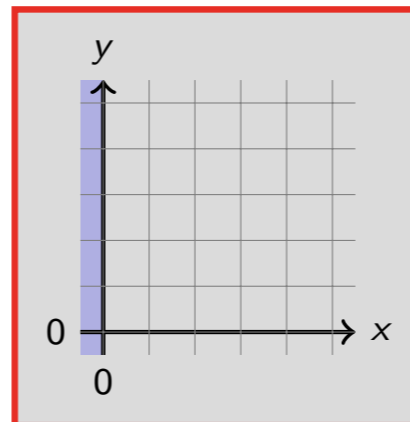
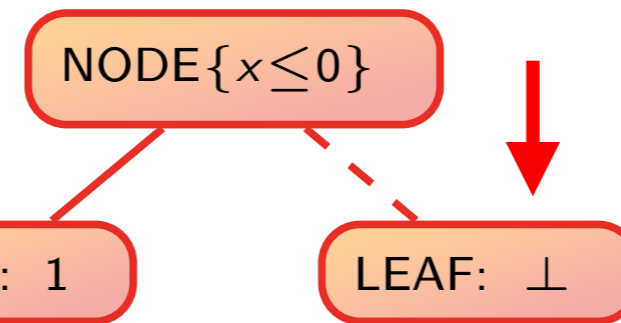
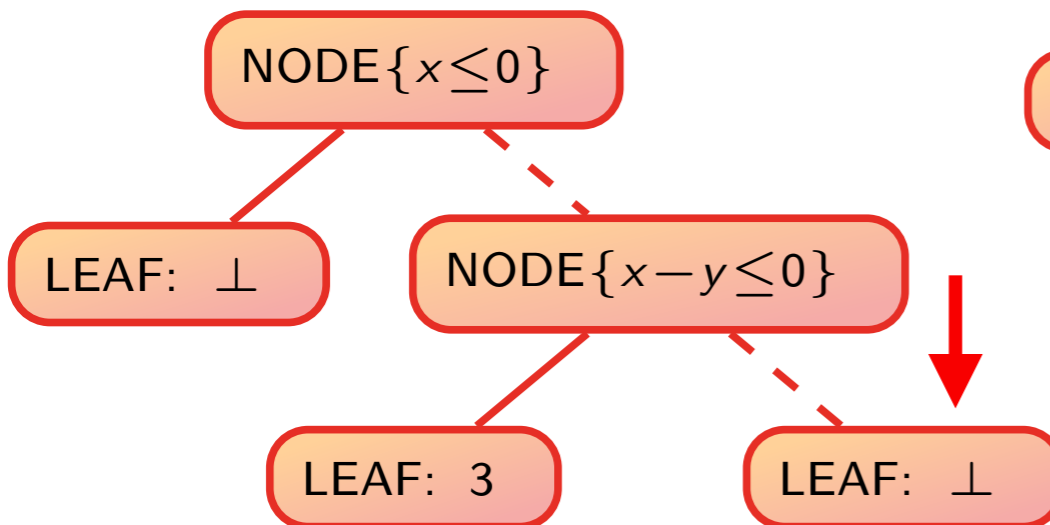
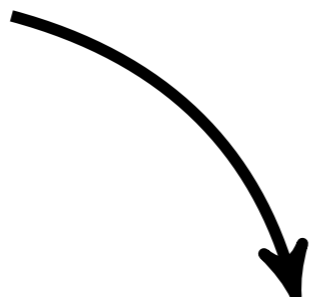
1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE{ $t_2.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_1.c$ } :  $l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
  
```



Join



1

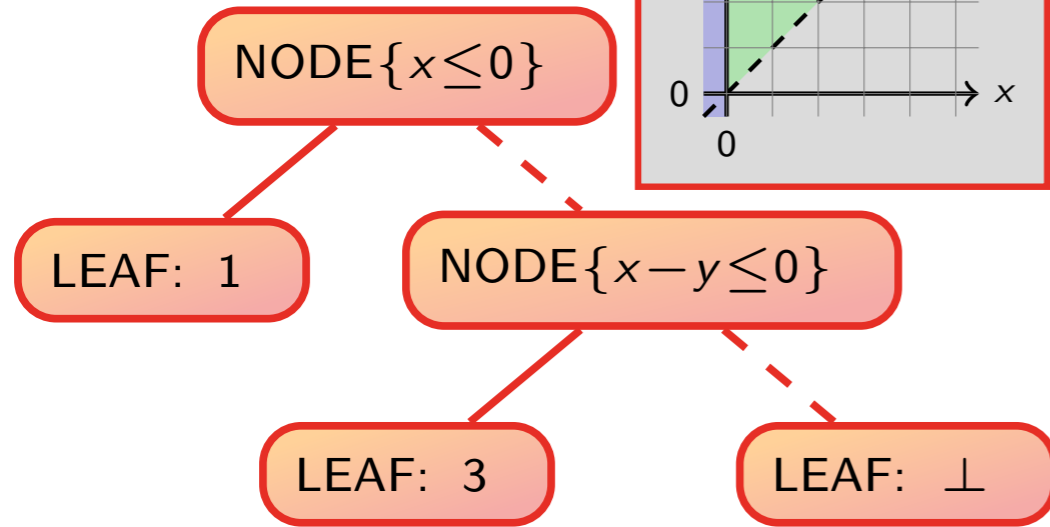
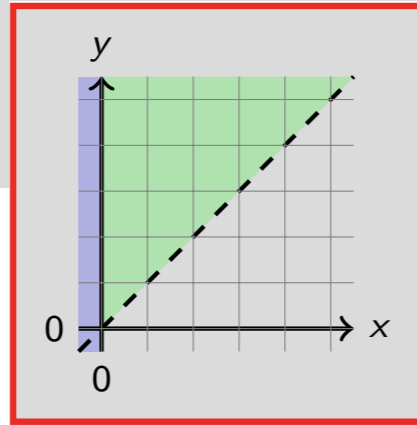


Algorithm 1 : Tree Unification

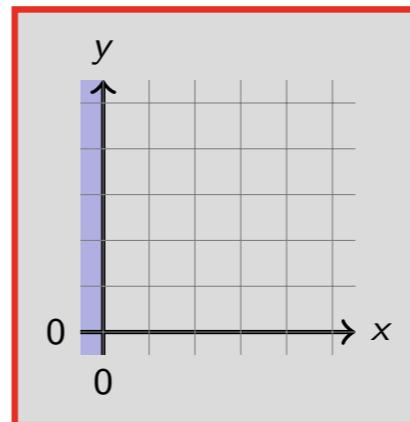
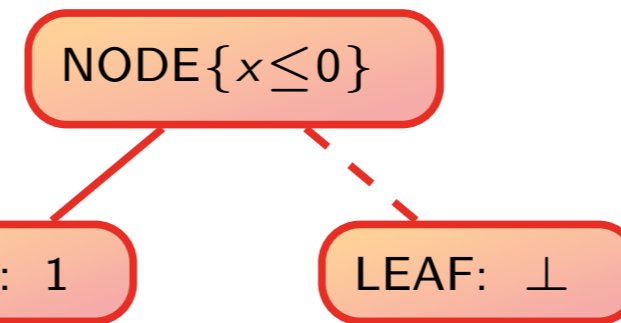
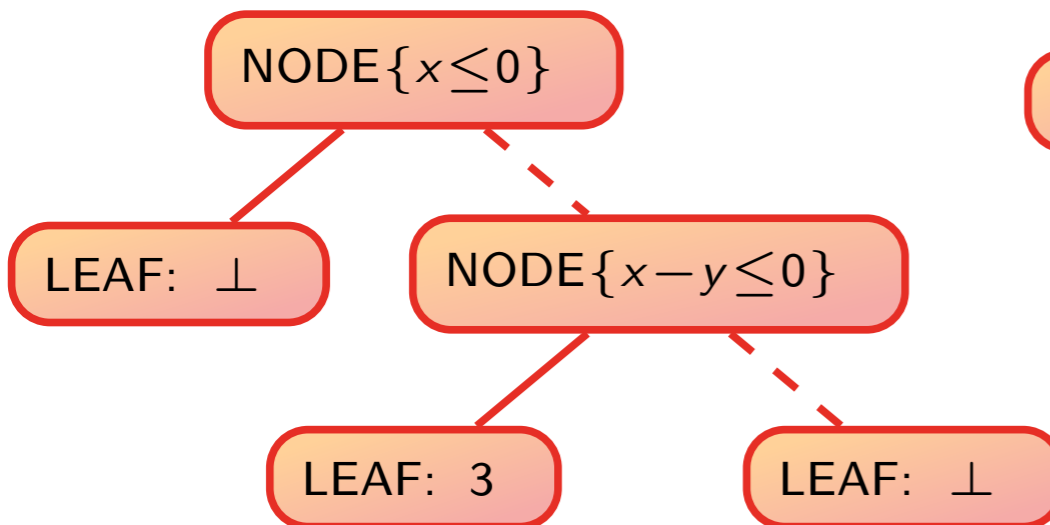
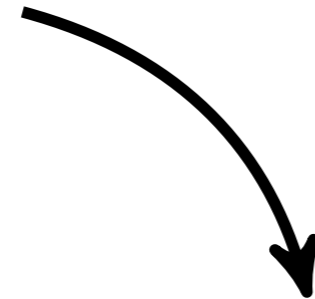
```

1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE{ $t_2.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_1.c$ } :  $l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
  
```

Join



1



Algorithm 1 : Tree Unification

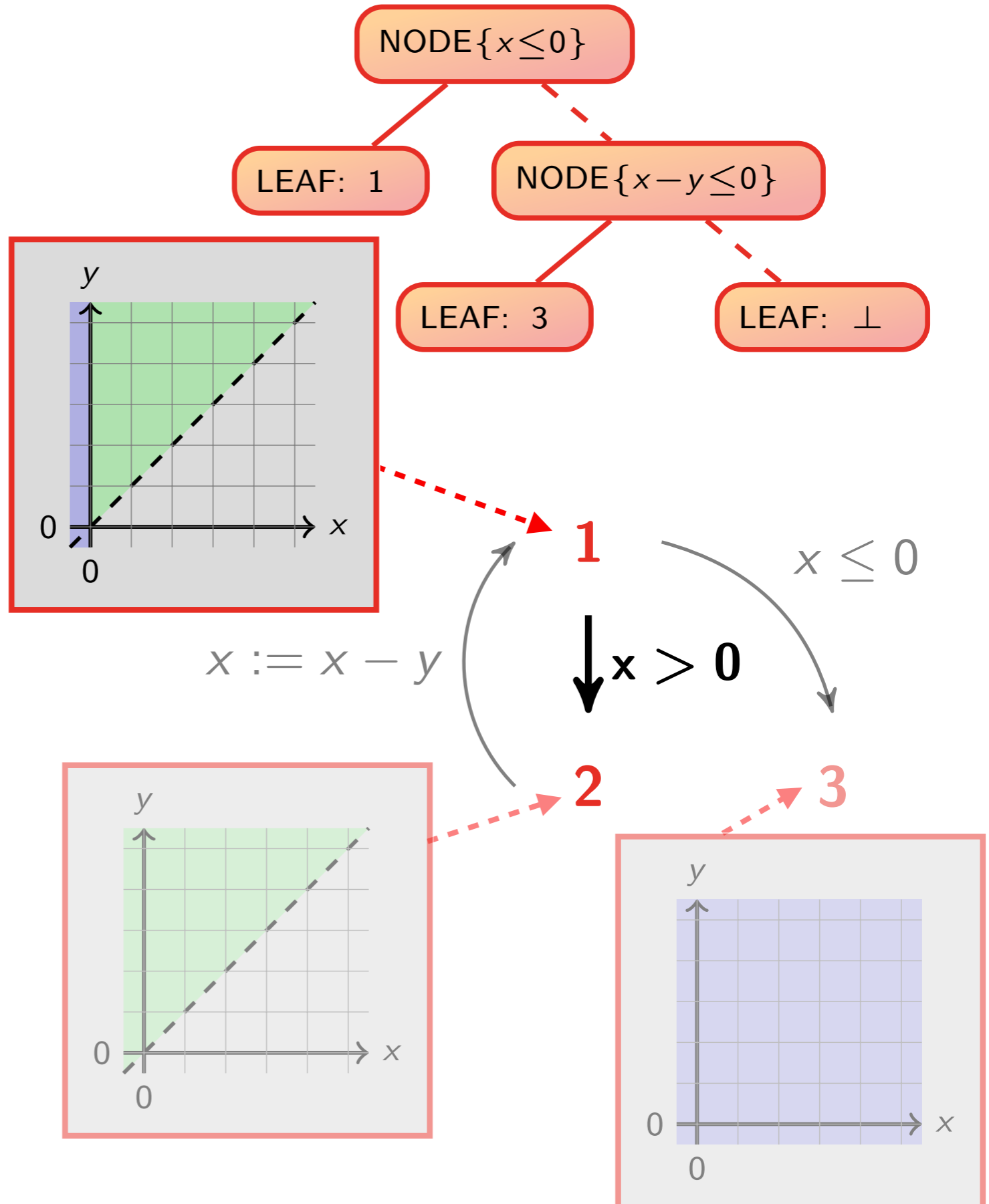
```

1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
6:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.r$ )
7:     return (NODE{ $t_2.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_L t_2.c$ ) then
9:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2$ )
10:    ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2$ )
11:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_1.c$ } :  $l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_1, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
  
```

Example

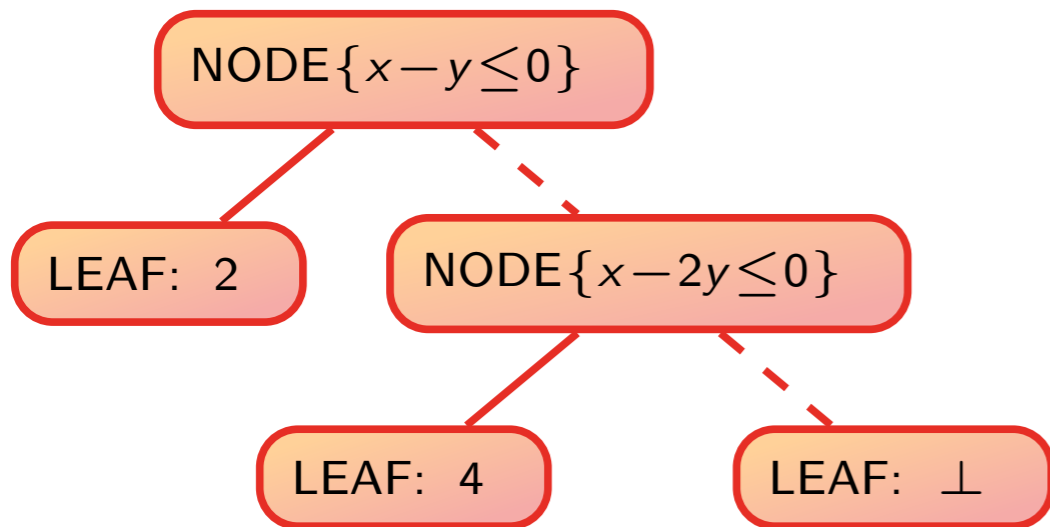
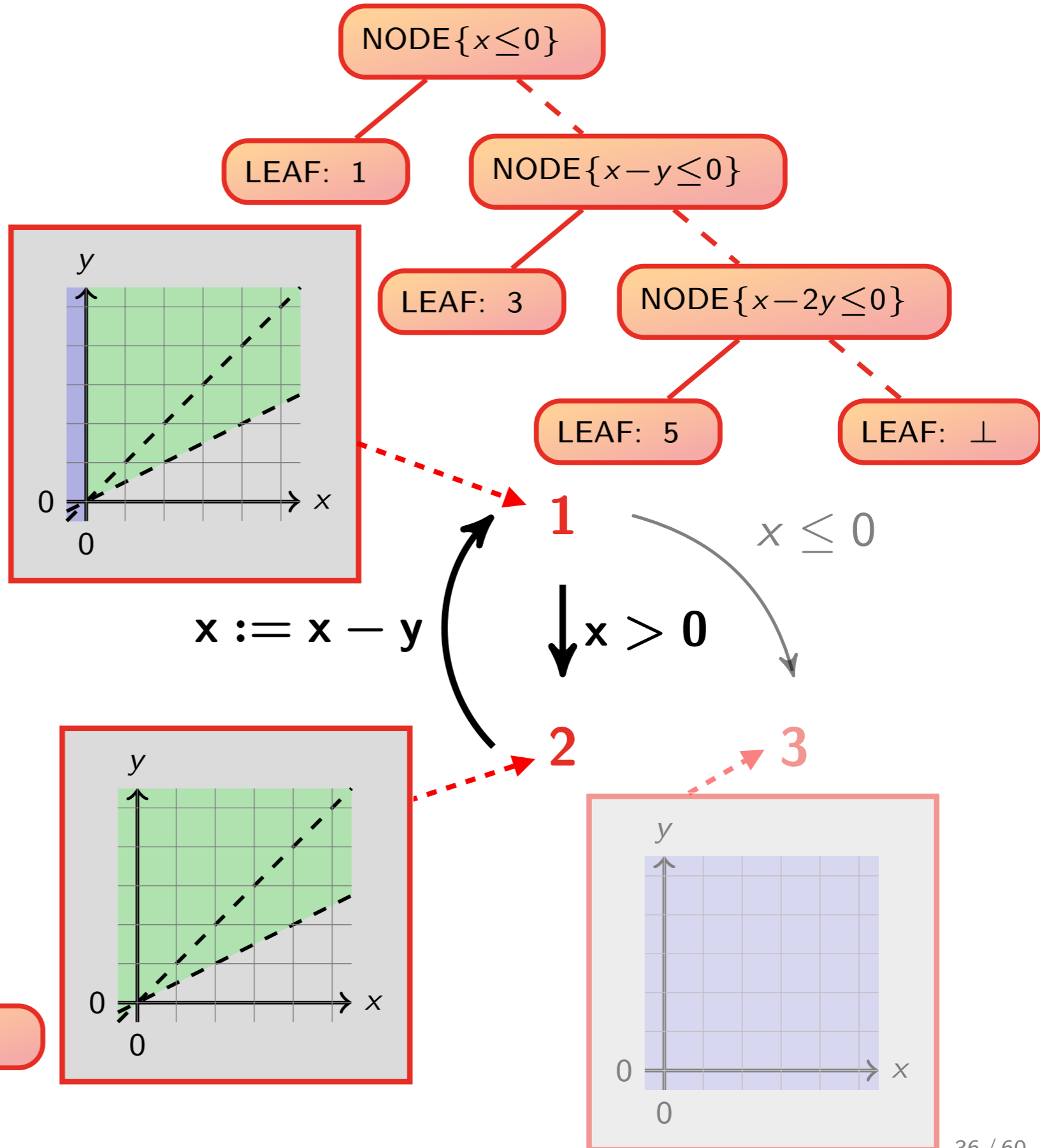
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

we have taken $x > 0$
 into account and we
 have done the join



Example

```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

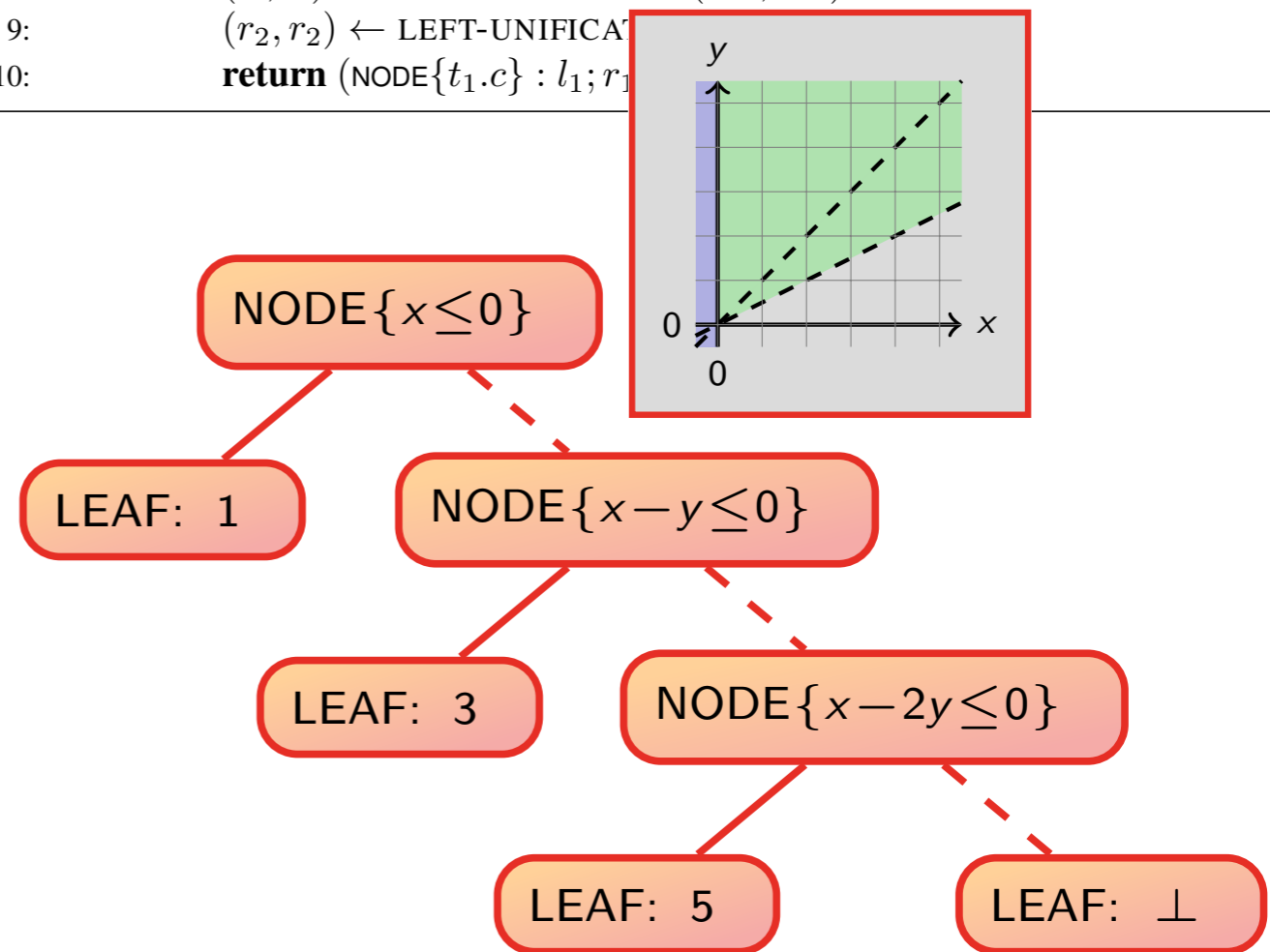
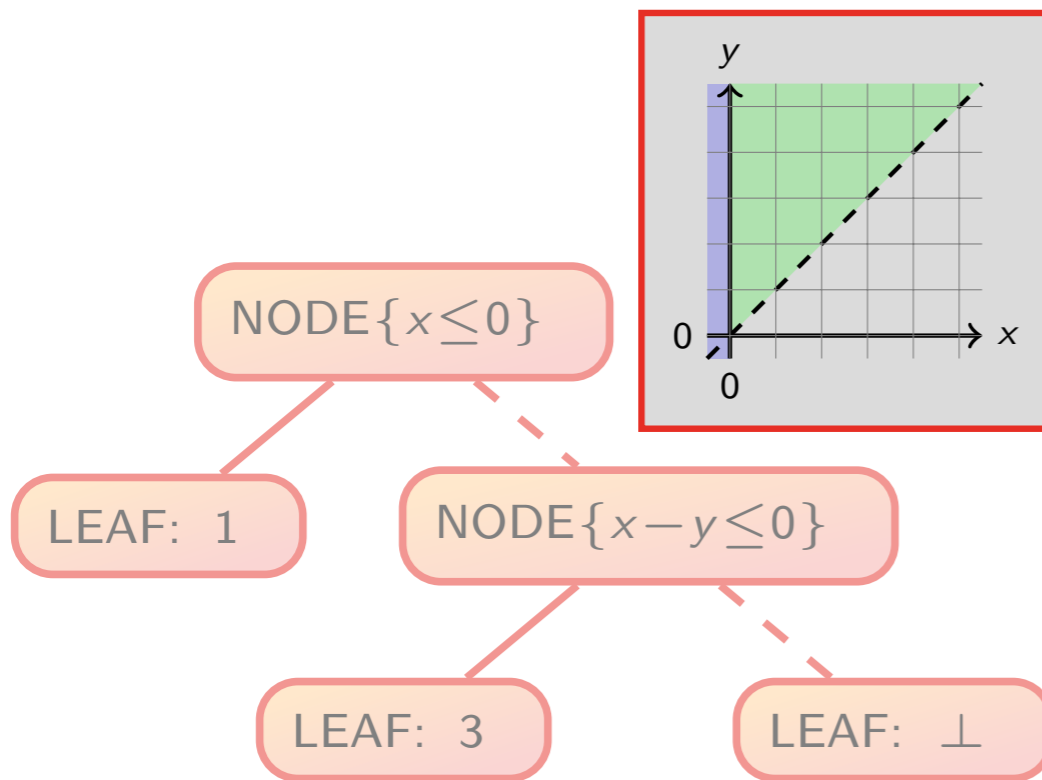


Widening: Left Unification

Algorithm 5 : Tree Left Unification

```

1: function LEFT-UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else
5:     if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\top} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_1, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:    return (NODE( $t_1.c$ ) :  $l_1; r_1$ )
    
```

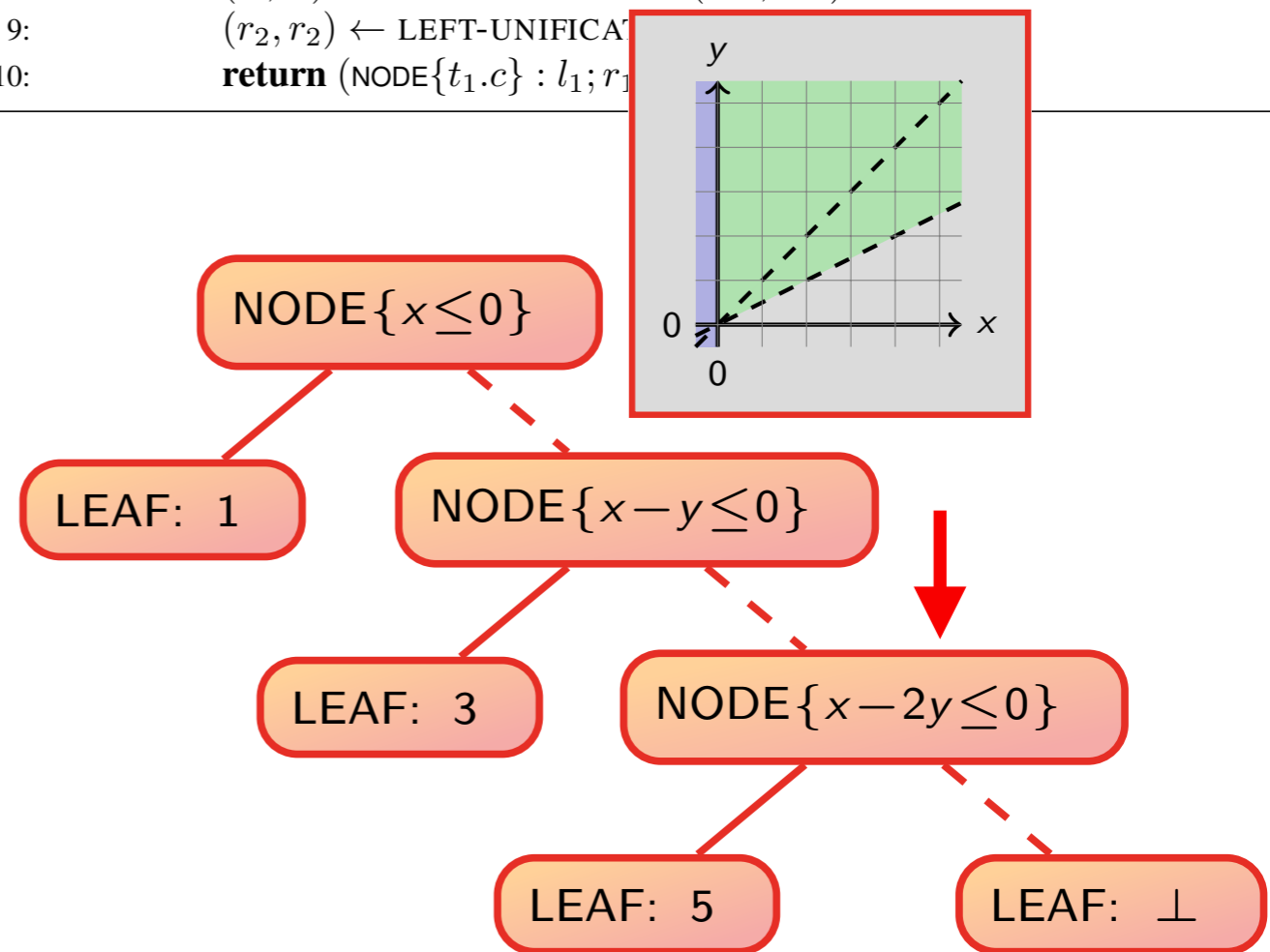
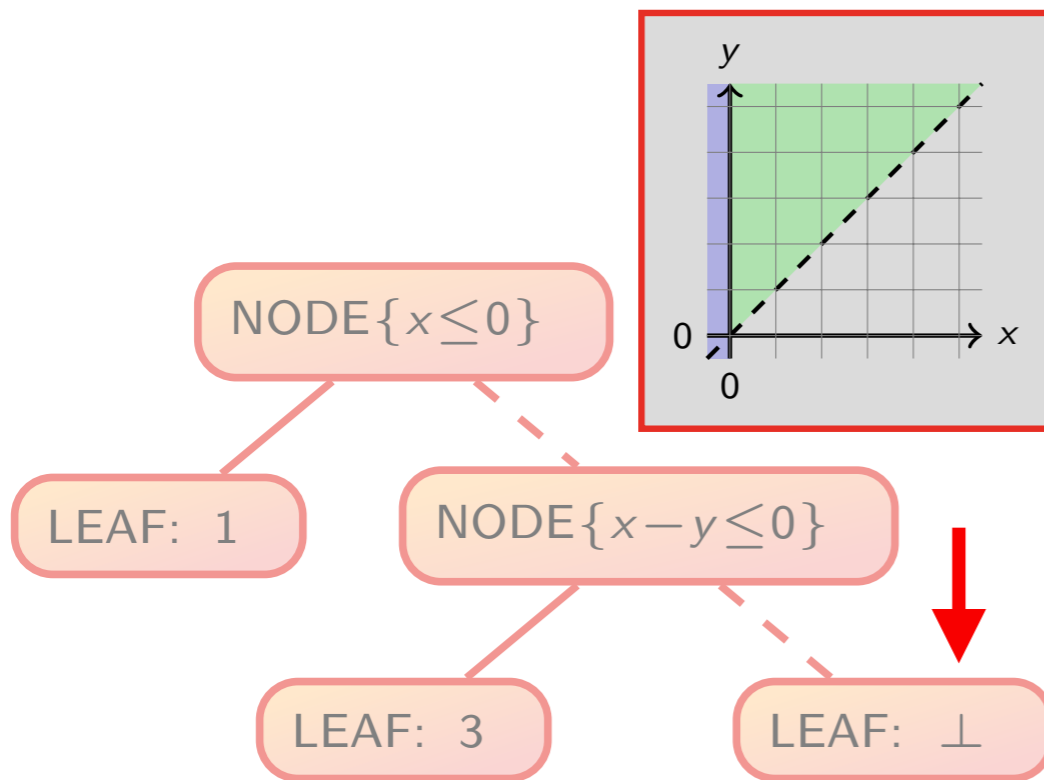


Widening: Left Unification

Algorithm 5 : Tree Left Unification

```

1: function LEFT-UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else
5:     if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\top} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_1, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:      return (NODE( $t_1.c$ ) :  $l_1; r_1$ )
    
```

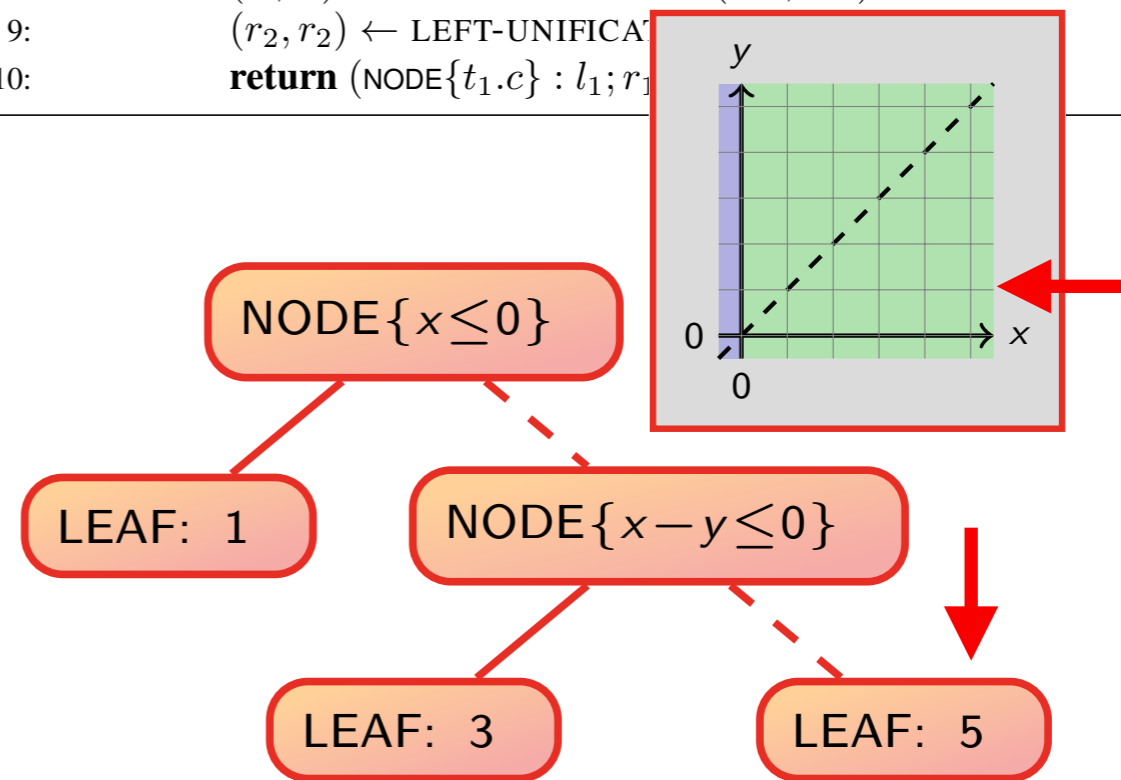
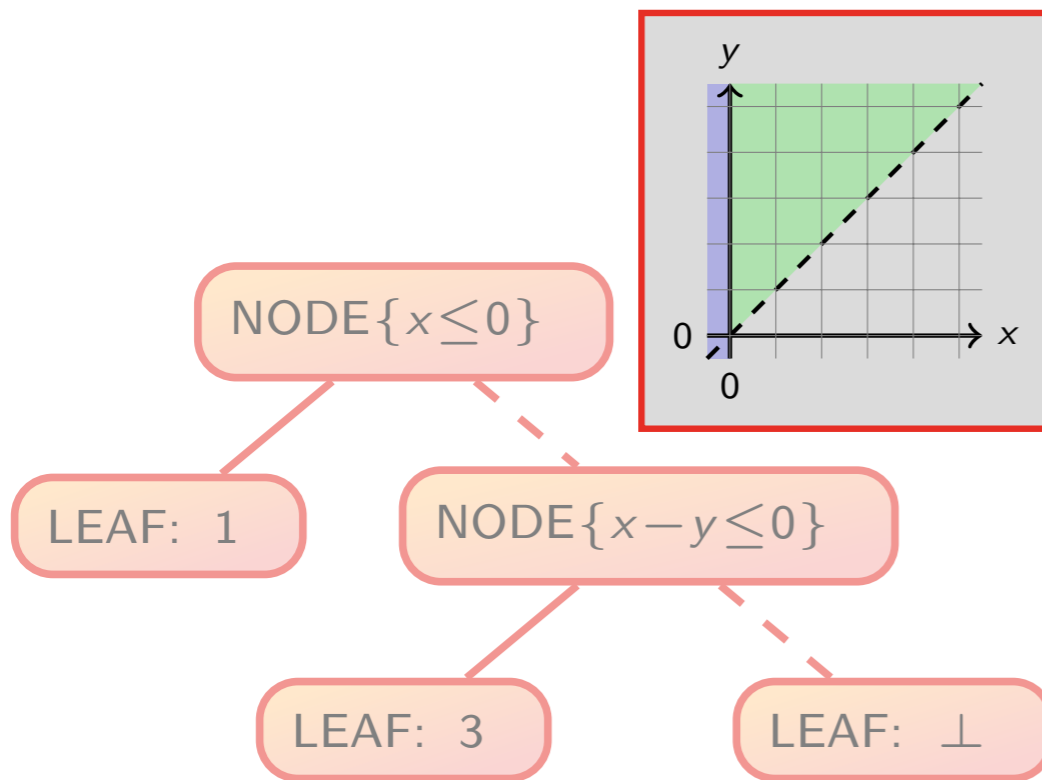


Widening: Left Unification

Algorithm 5 : Tree Left Unification

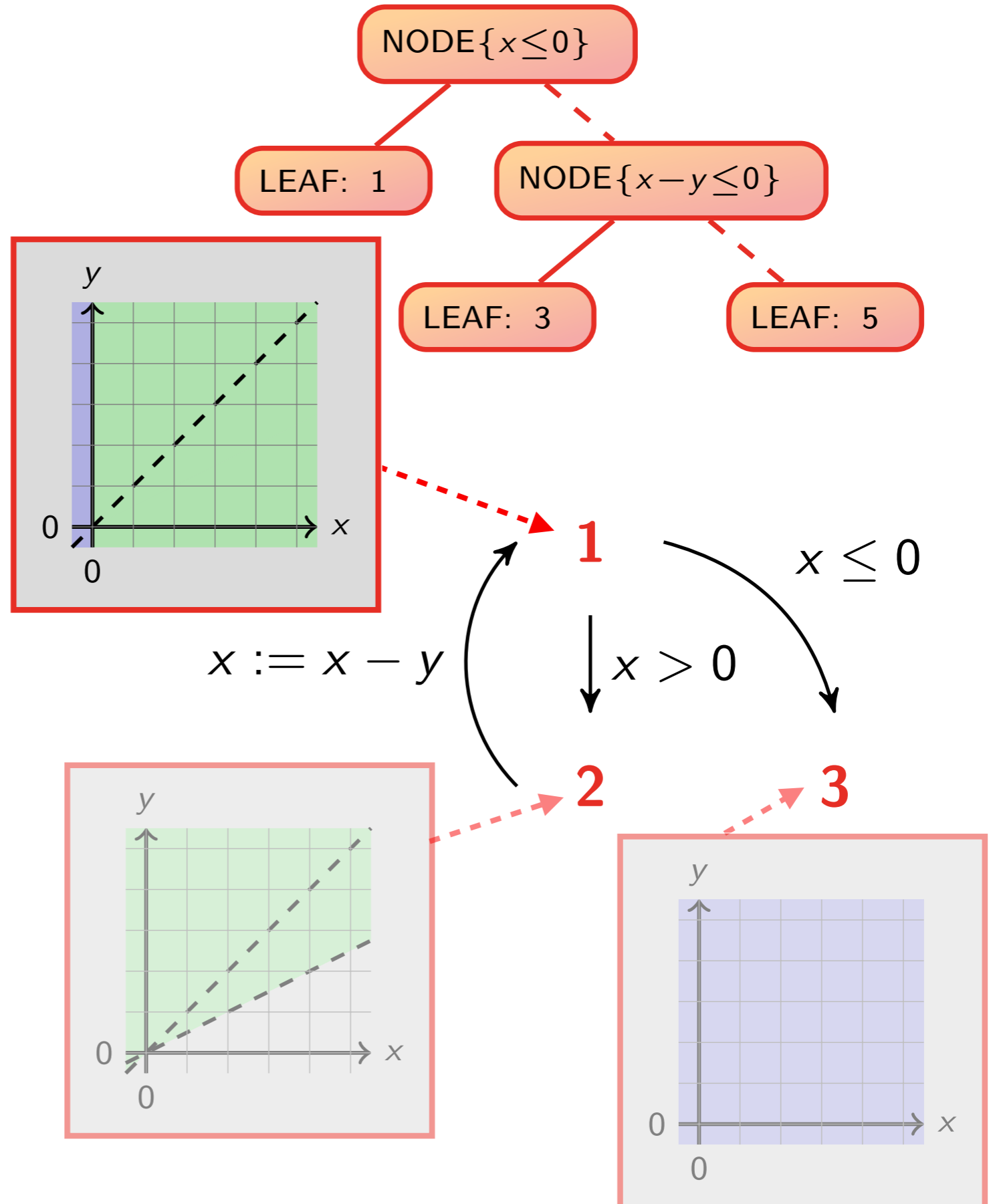
```

1: function LEFT-UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else
5:     if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\top} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_1, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:    return (NODE( $t_1.c$ ) :  $l_1; r_1$ )
    
```



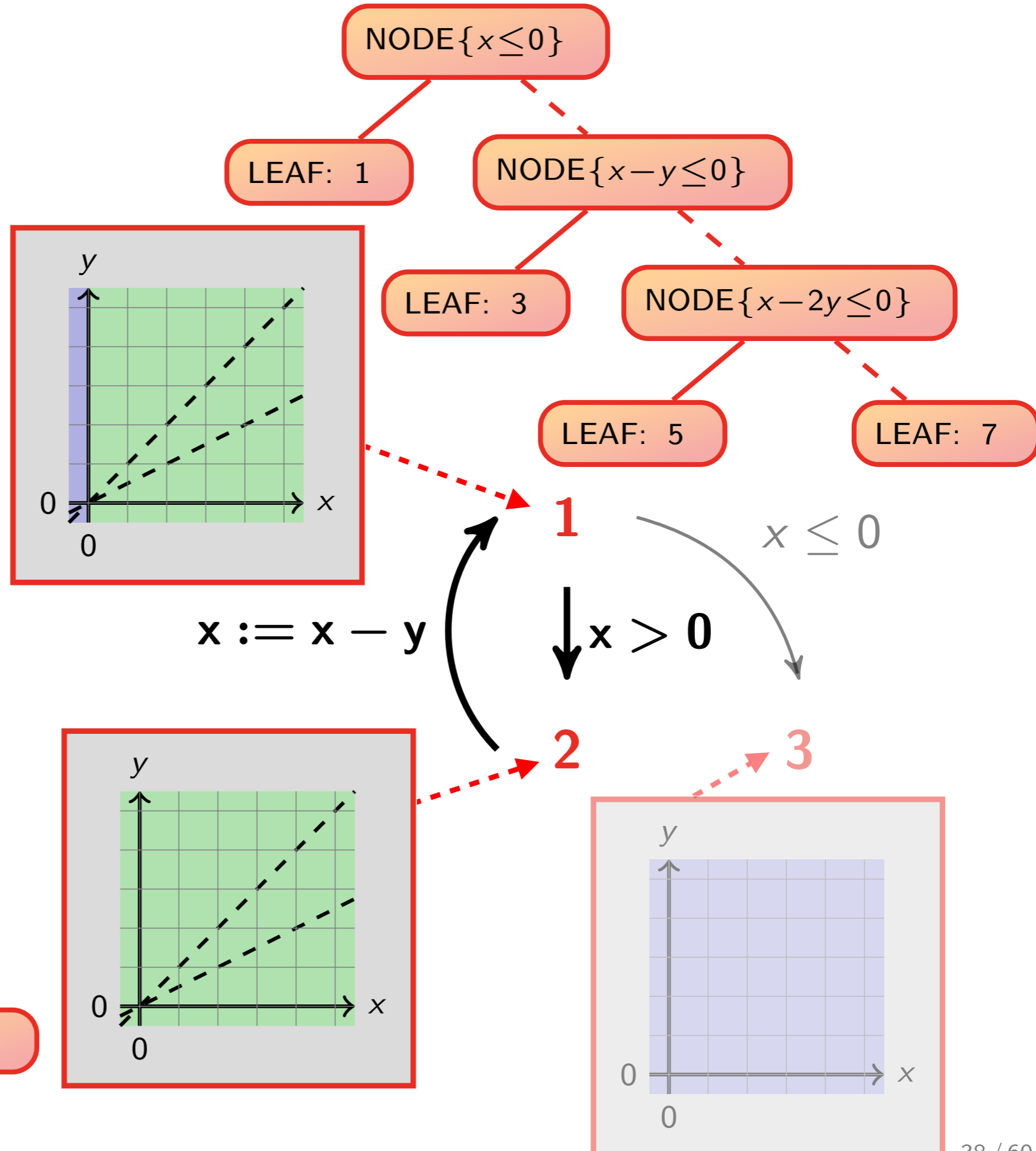
Example

```
int : x, y
while 1(x > 0) do
    2x := x - y
od3
```



Example

```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

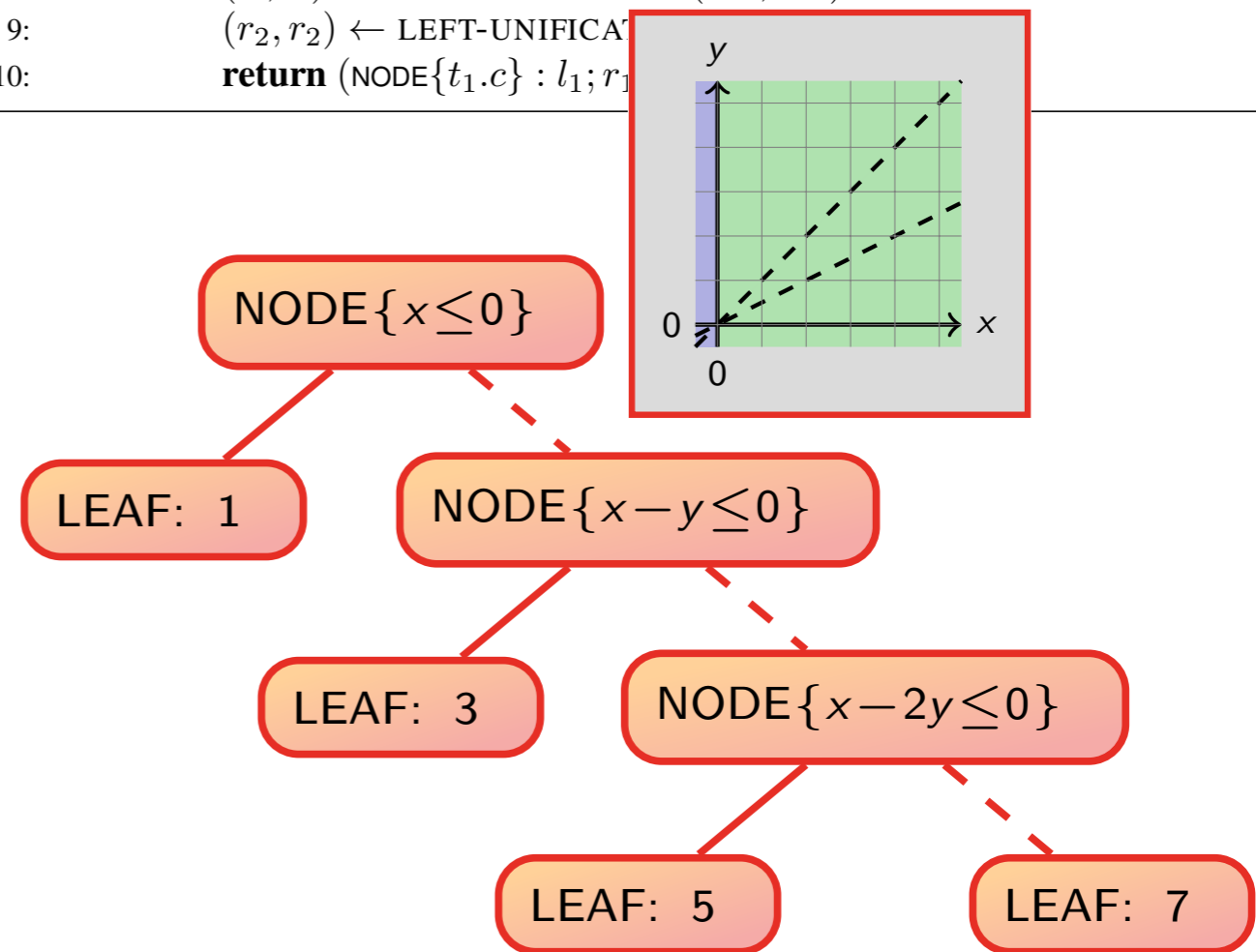
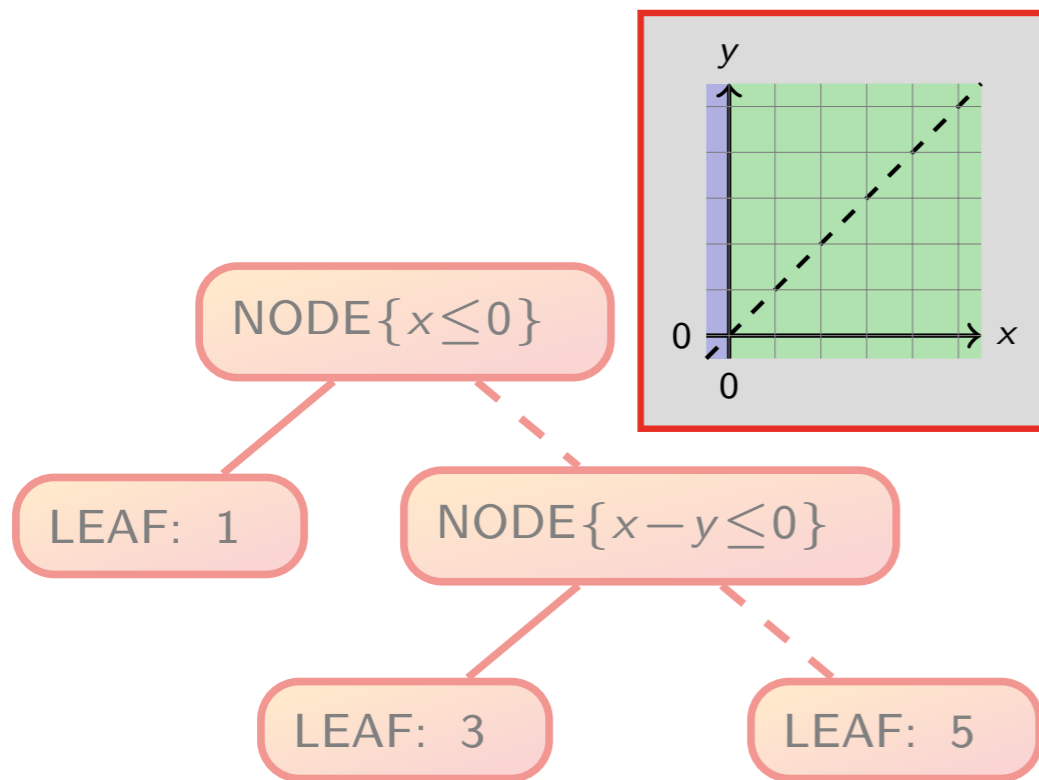


Widening: Domain Over-Approximation

Algorithm 5 : Tree Left Unification

```

1: function LEFT-UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else
5:     if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\top} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_1, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:    return (NODE( $t_1.c$ ) :  $l_1; r_1$ )
    
```

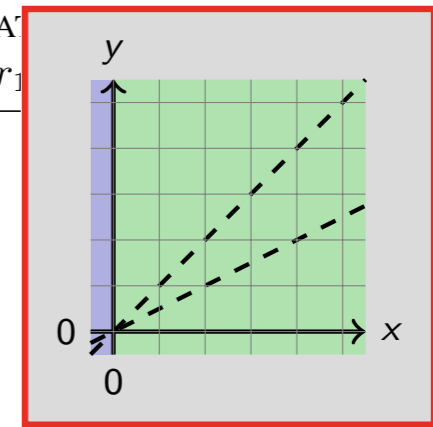
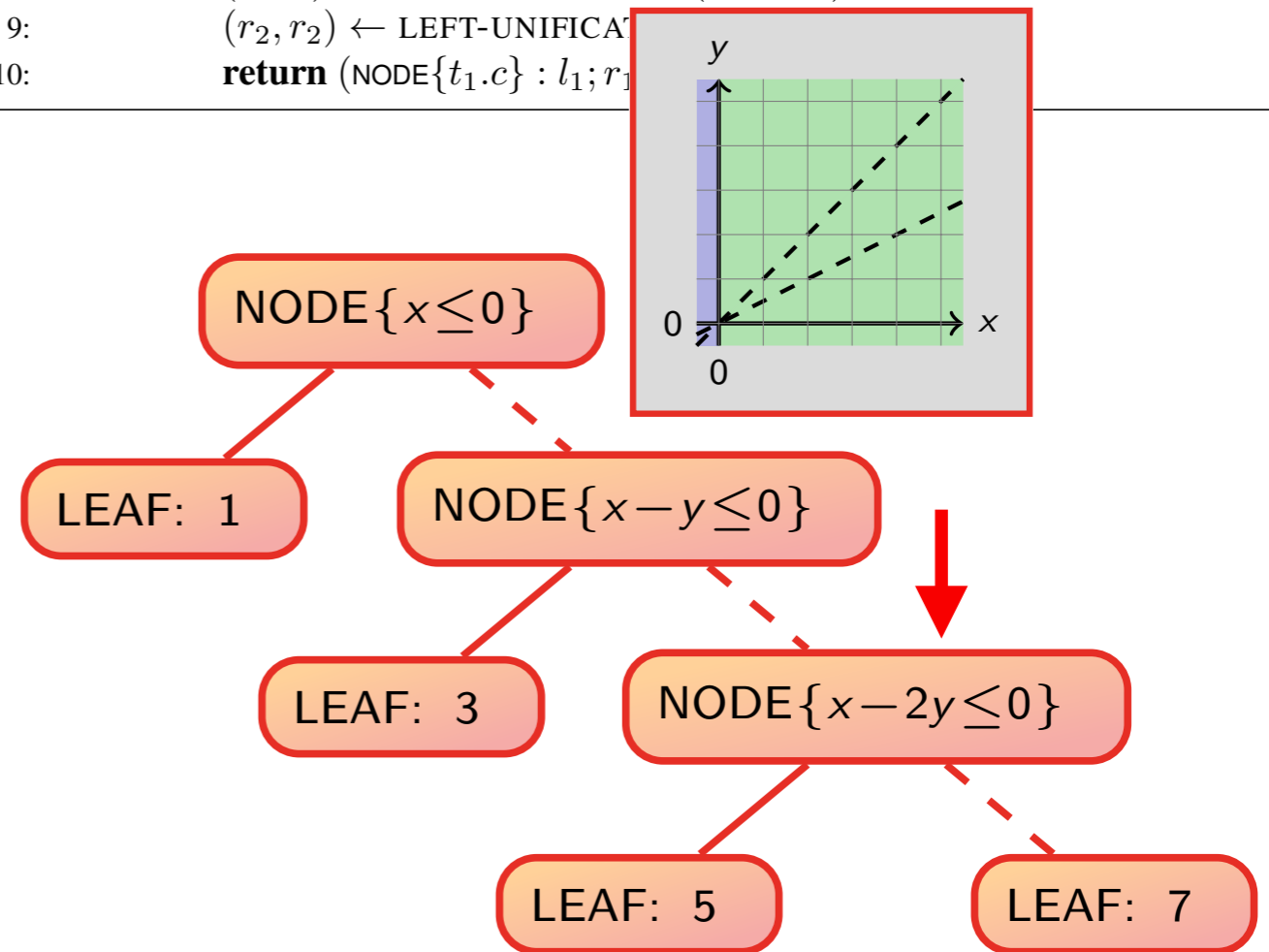
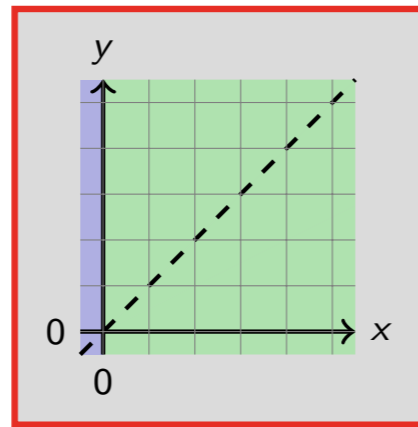
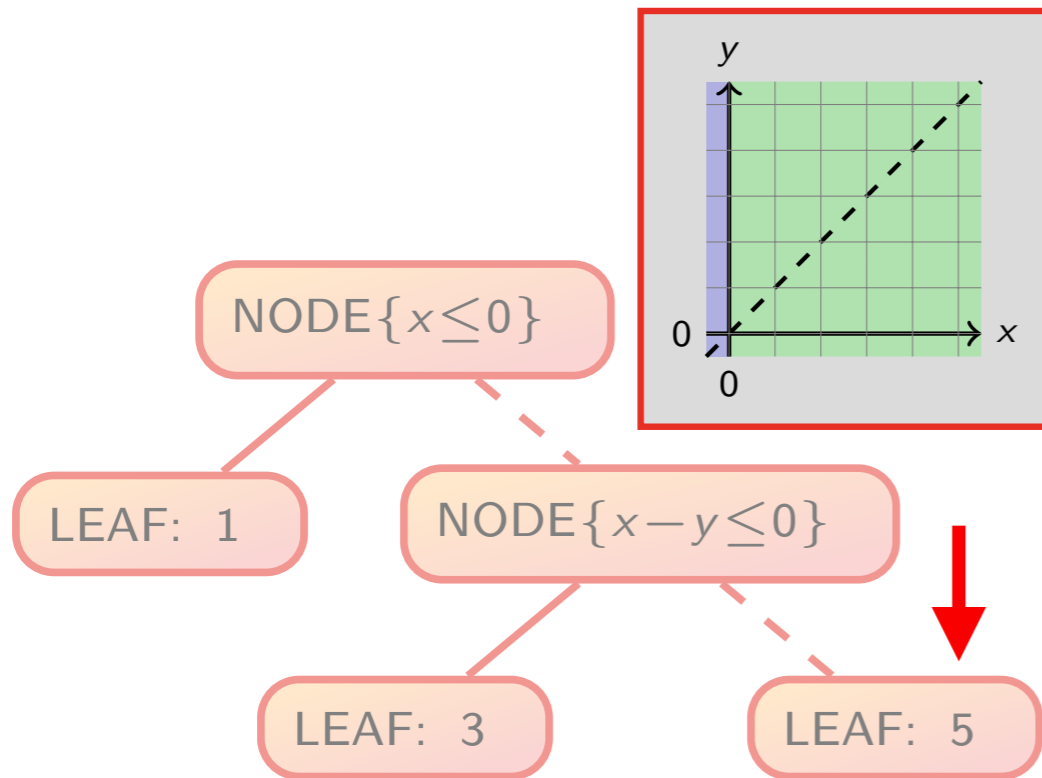


Widening: Domain Over-Approximation

Algorithm 5 : Tree Left Unification

```

1: function LEFT-UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else
5:     if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\top} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_1, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ )
    
```

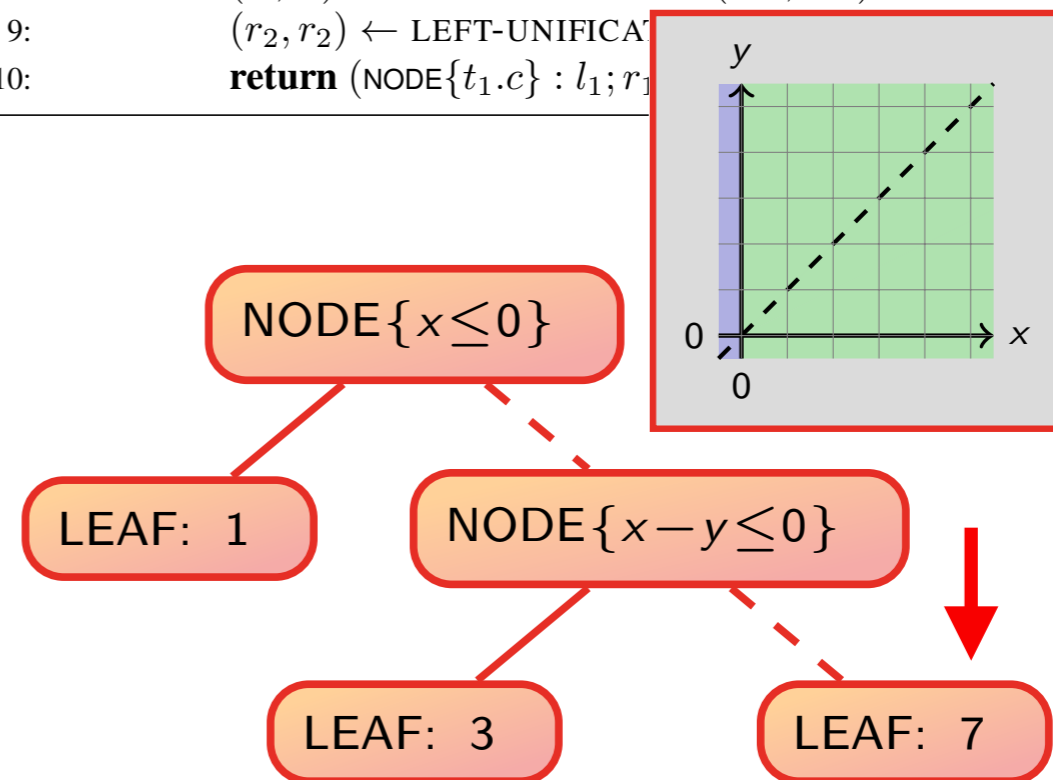
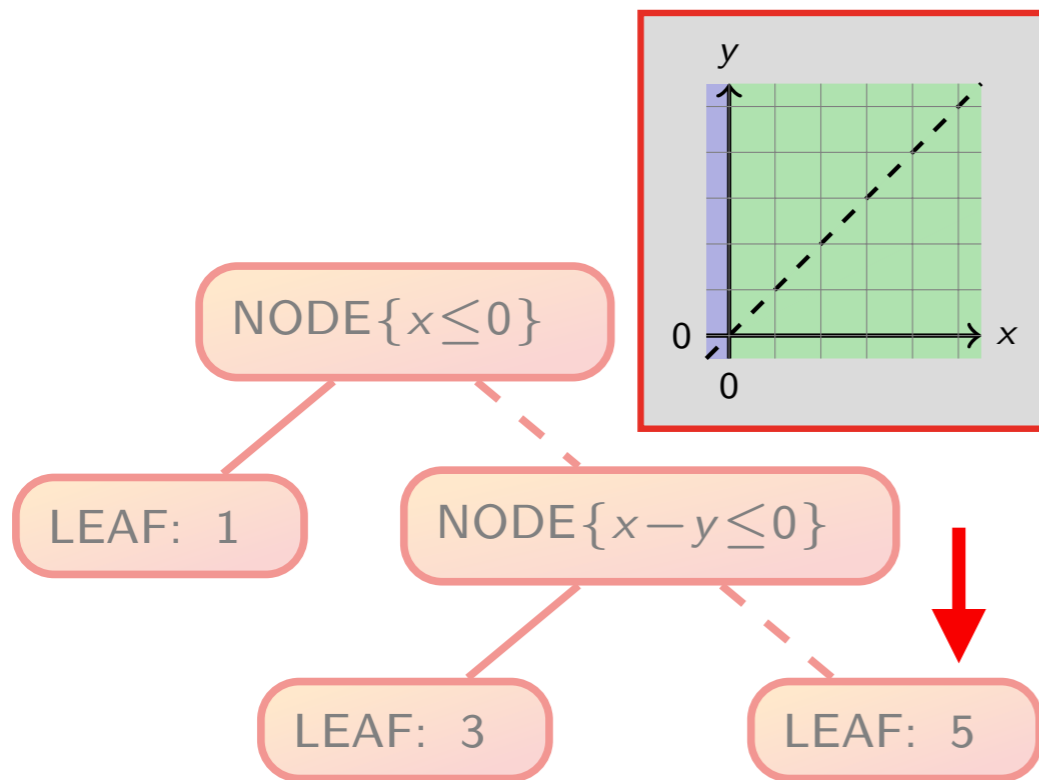


Widening: Domain Over-Approximation

Algorithm 5 : Tree Left Unification

```

1: function LEFT-UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else
5:     if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\top} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_1, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ )
    
```



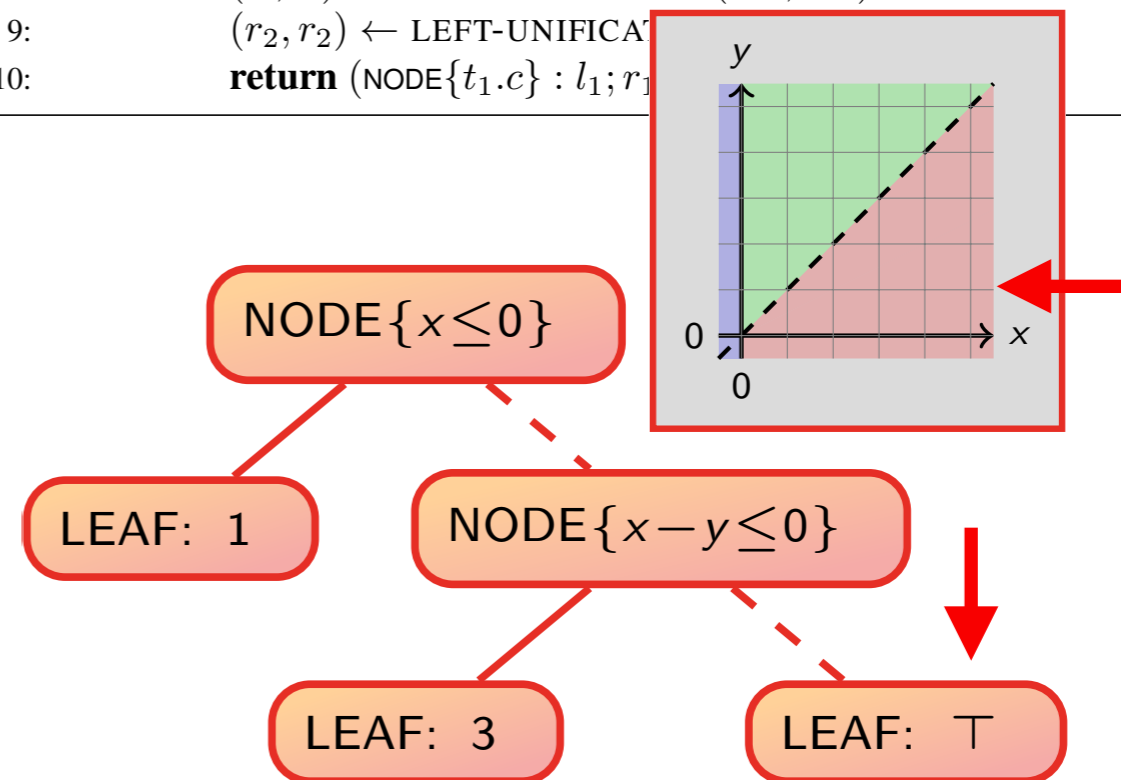
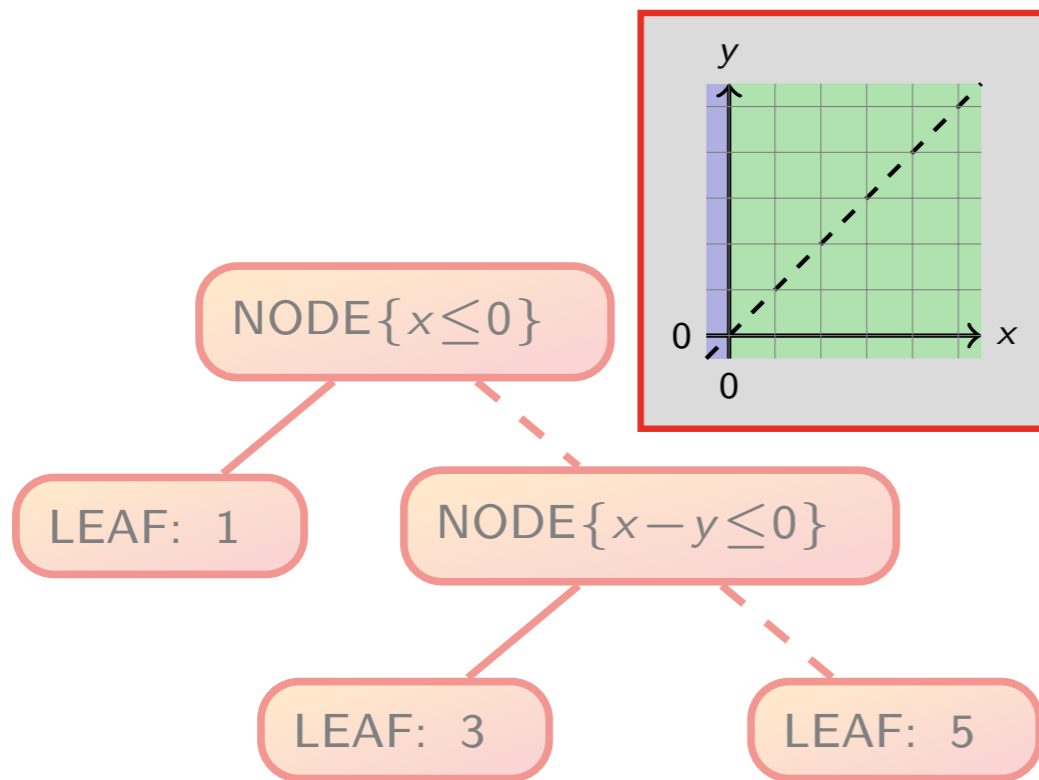
Widening: Domain Over-Approximation

Algorithm 5 : Tree Left Unification

```

1: function LEFT-UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else
5:     if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_L t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\top} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_1, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:    return (NODE{ $t_1.c$ } :  $l_1; r_1$ )

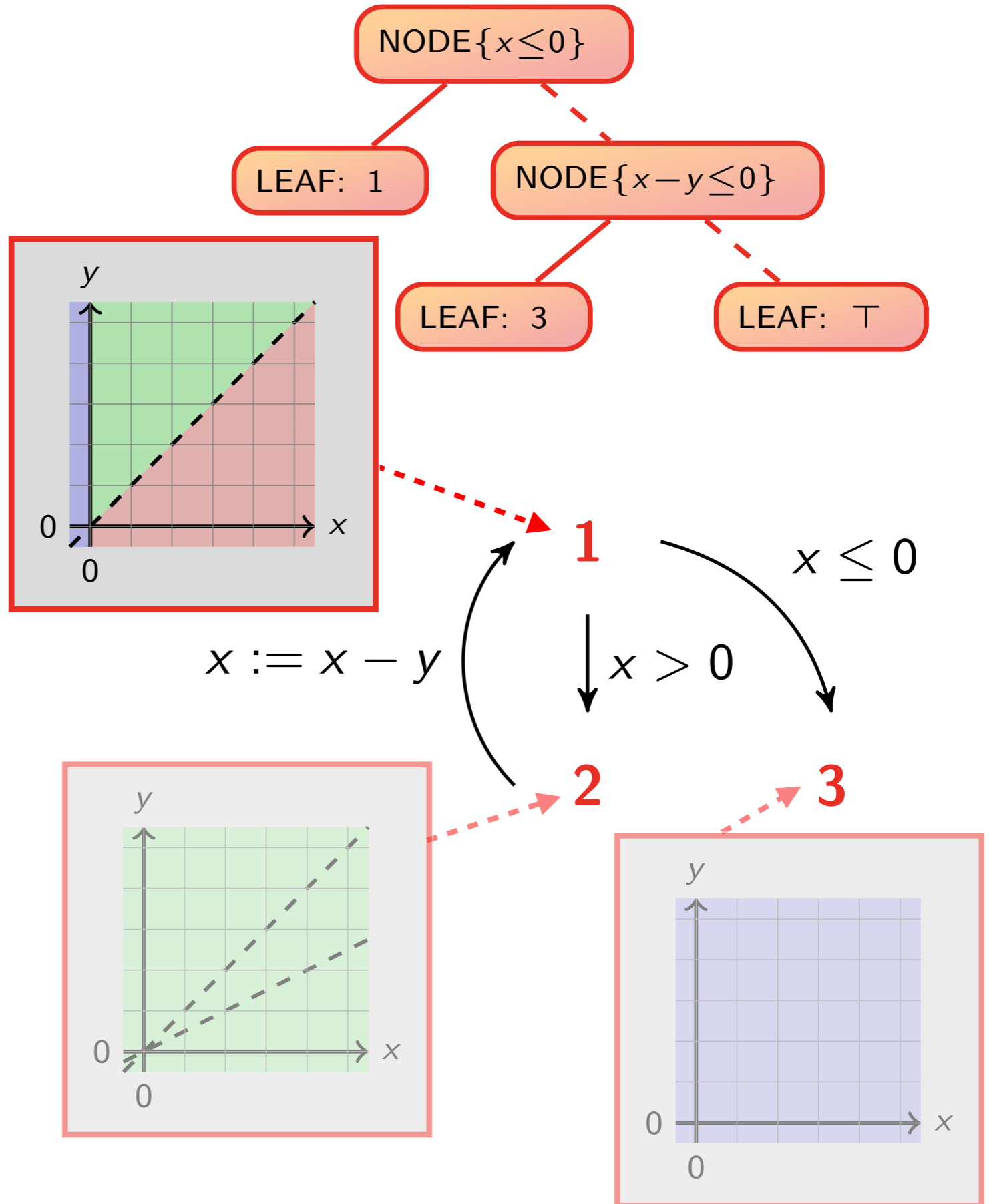
```



Example

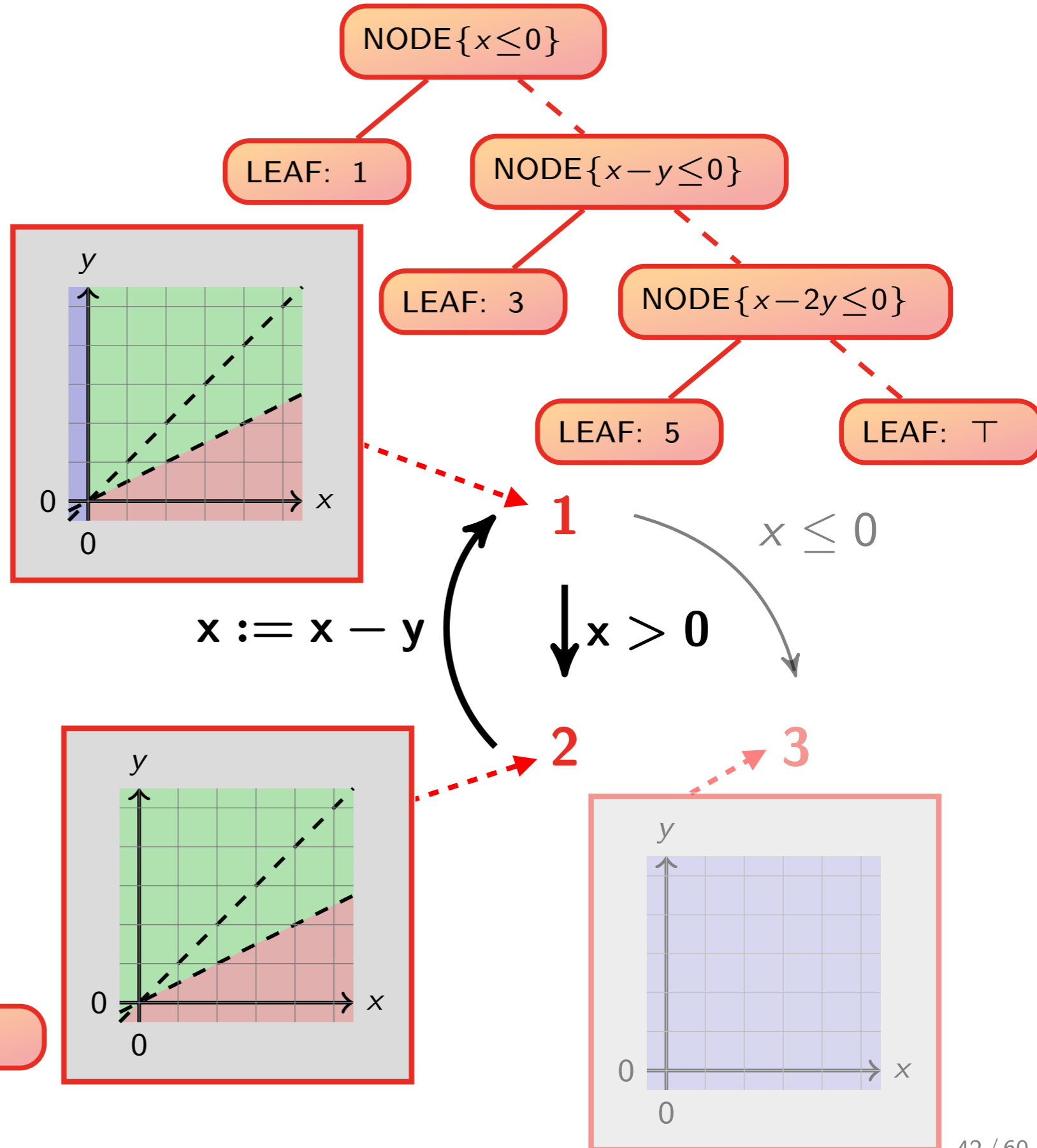
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

→ the widening is **sound!**



Example

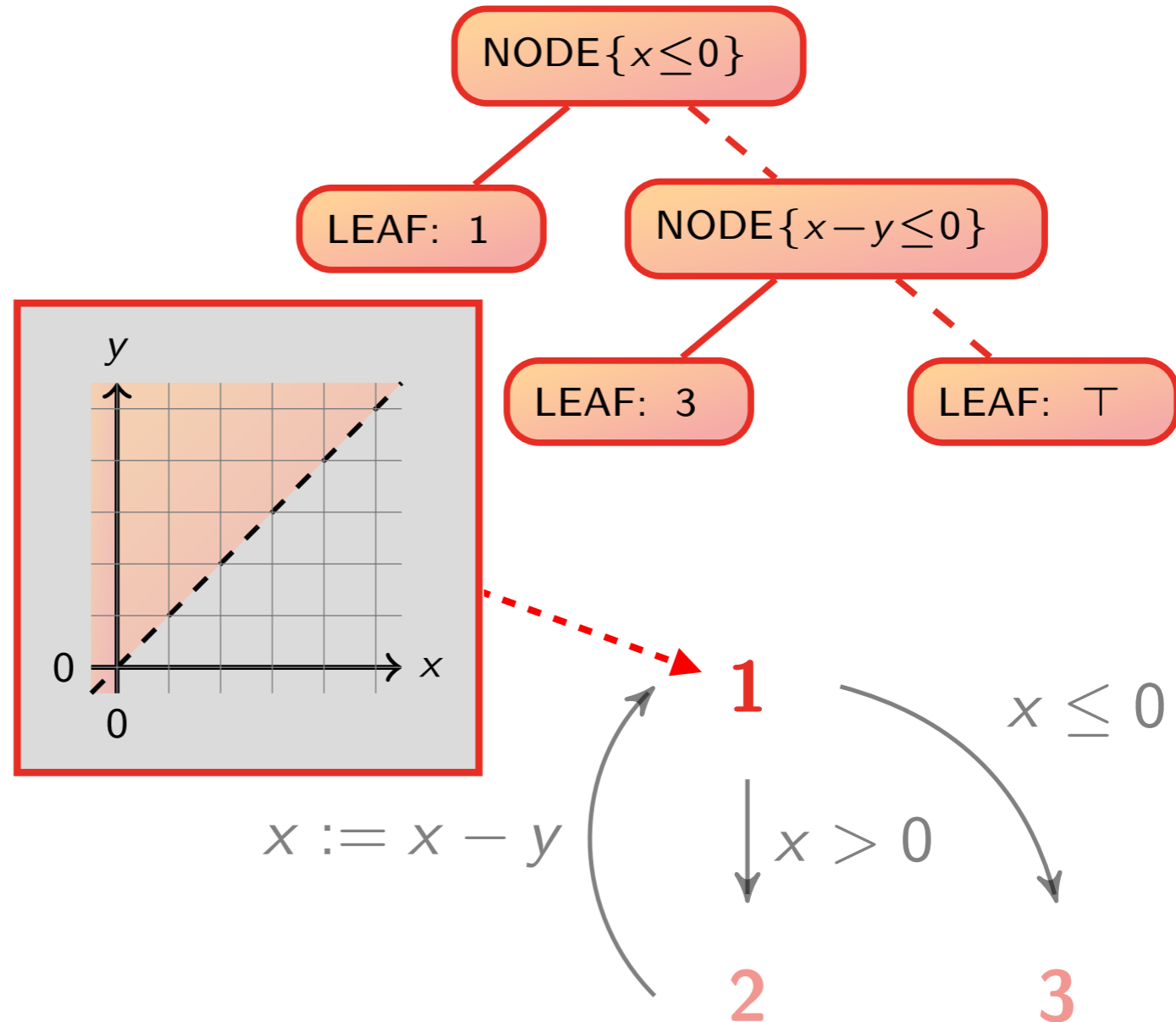
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



Example

```
int : x, y
while 1(x > 0) do
    2x := x - y
od3
```

the analysis gives $x \leq 0 \vee x \leq y$ as **sufficient precondition**

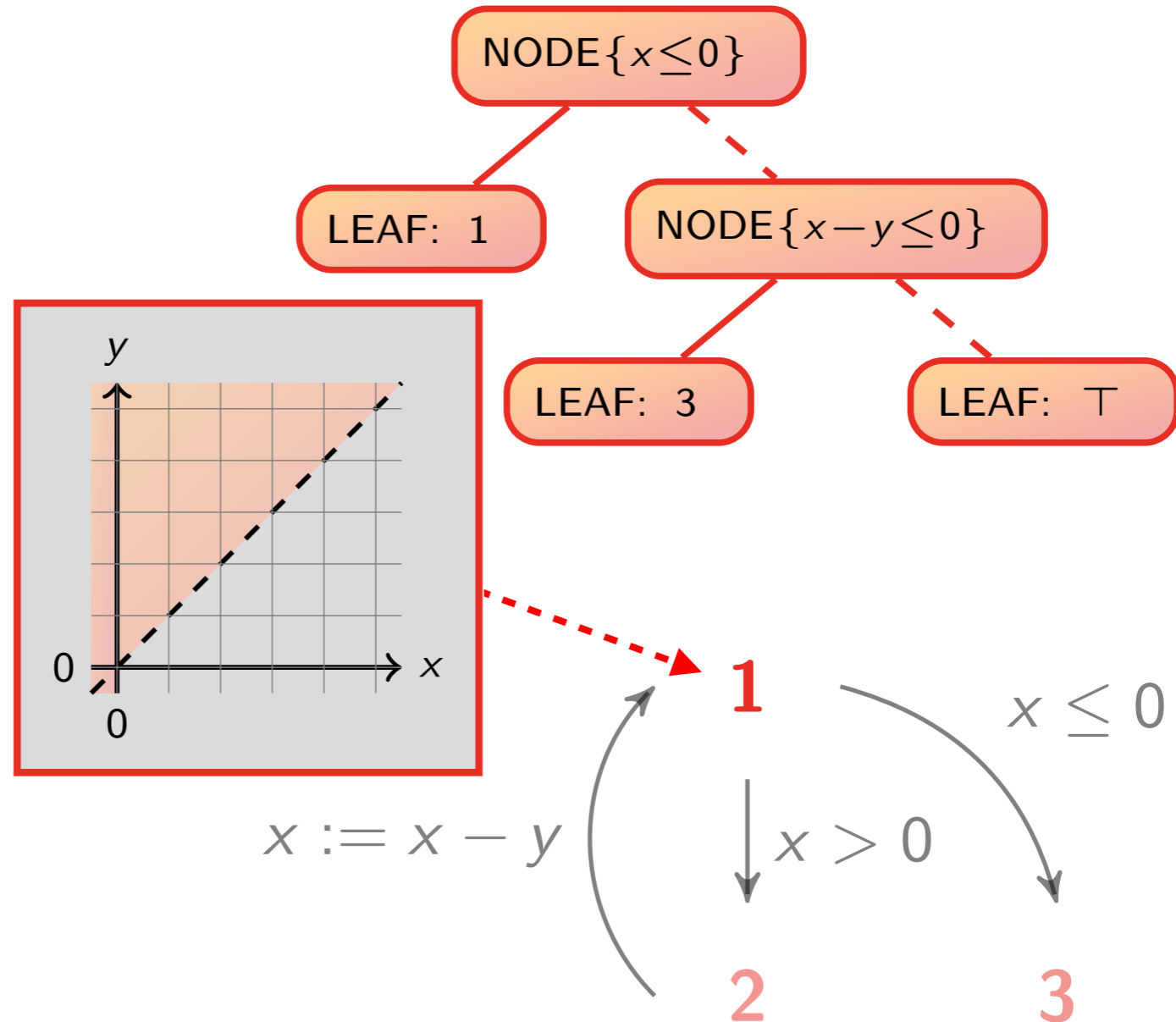


Example

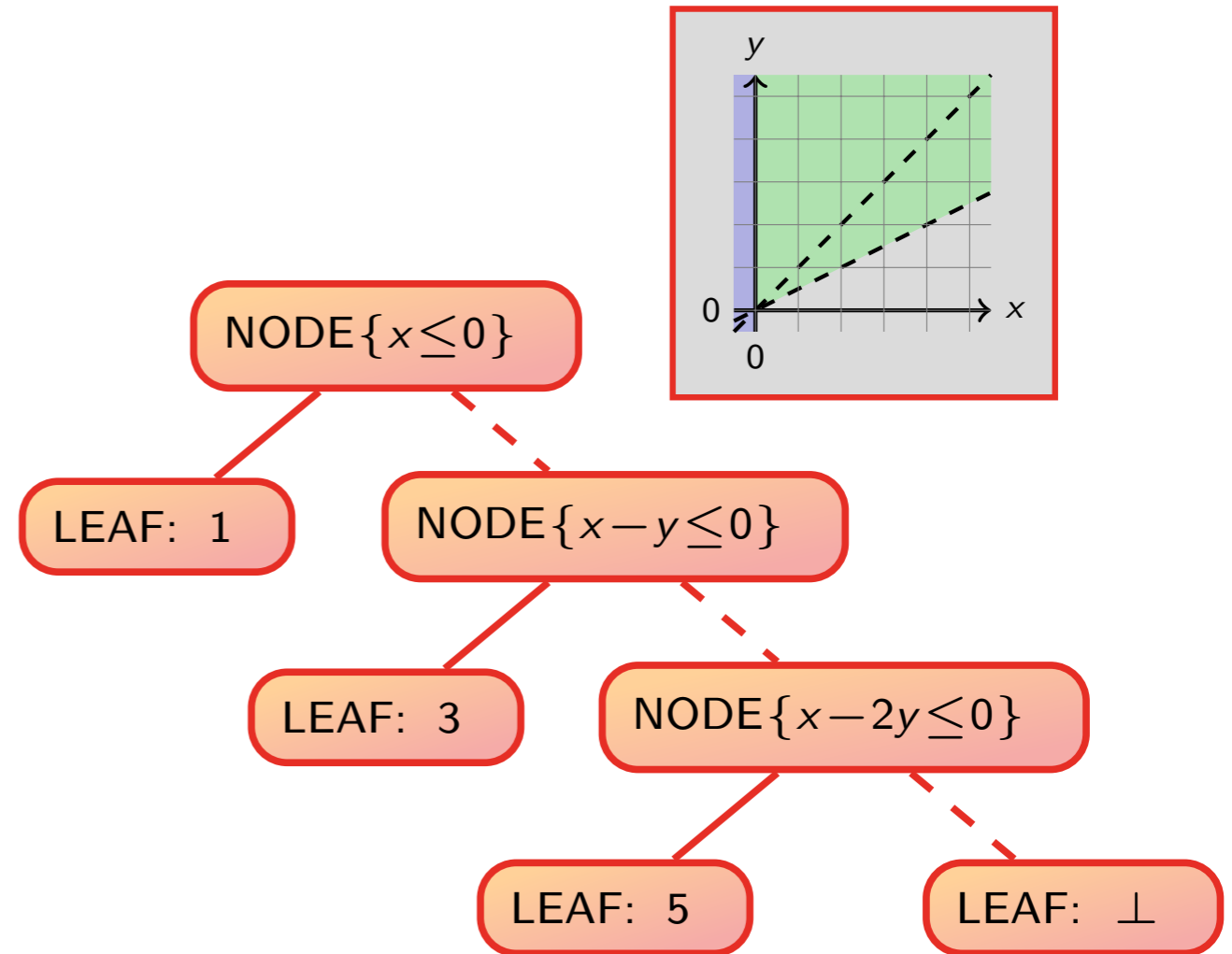
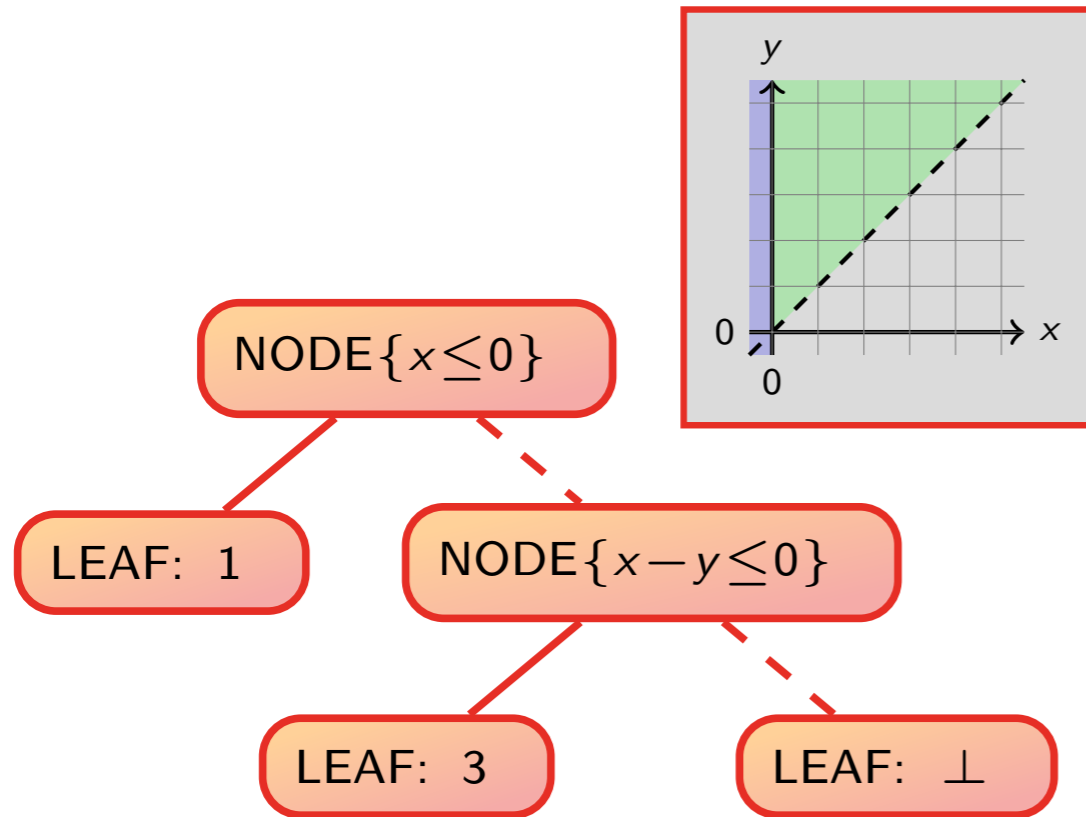
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

the analysis gives $x \leq 0 \vee x \leq y$
 as **sufficient precondition**

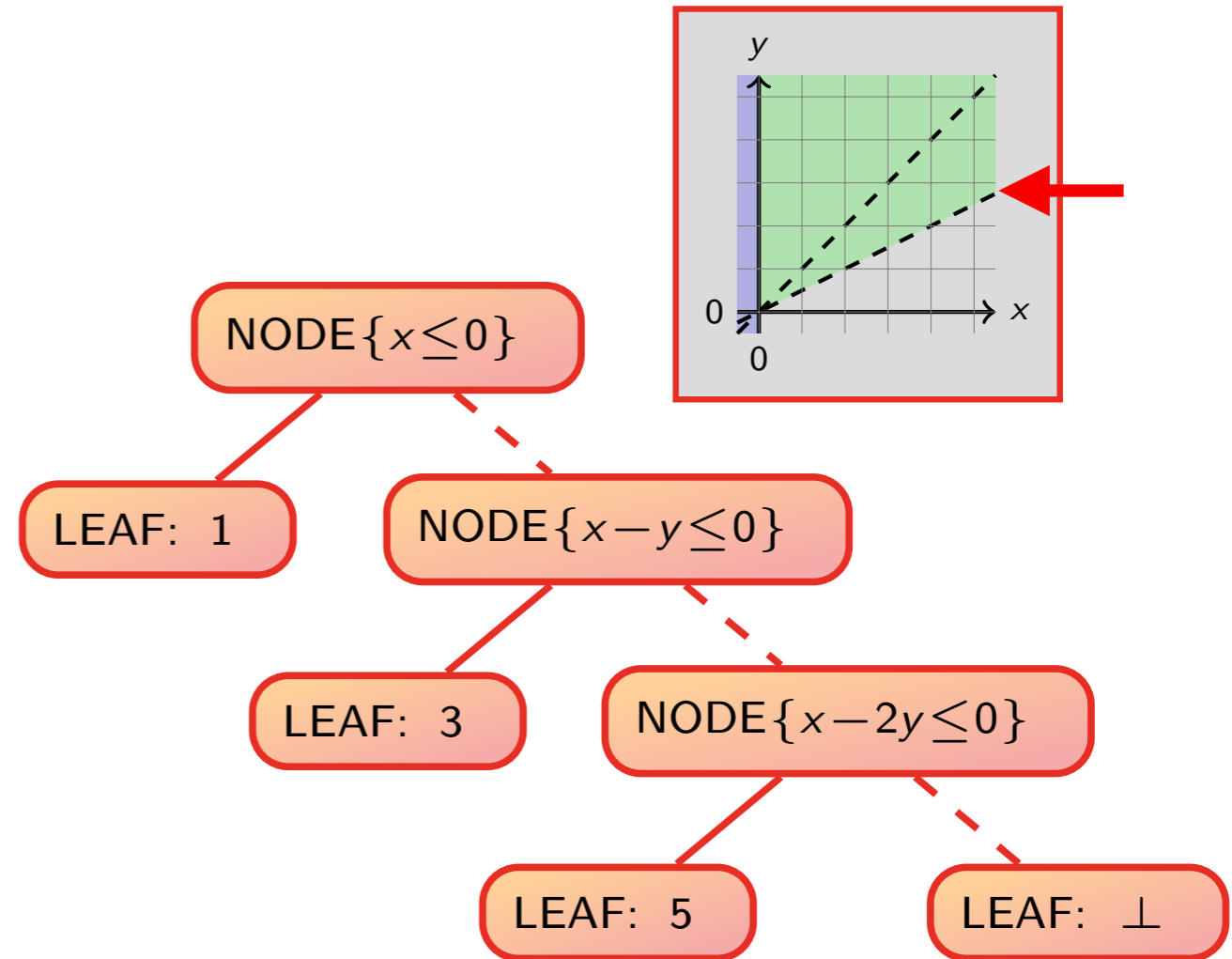
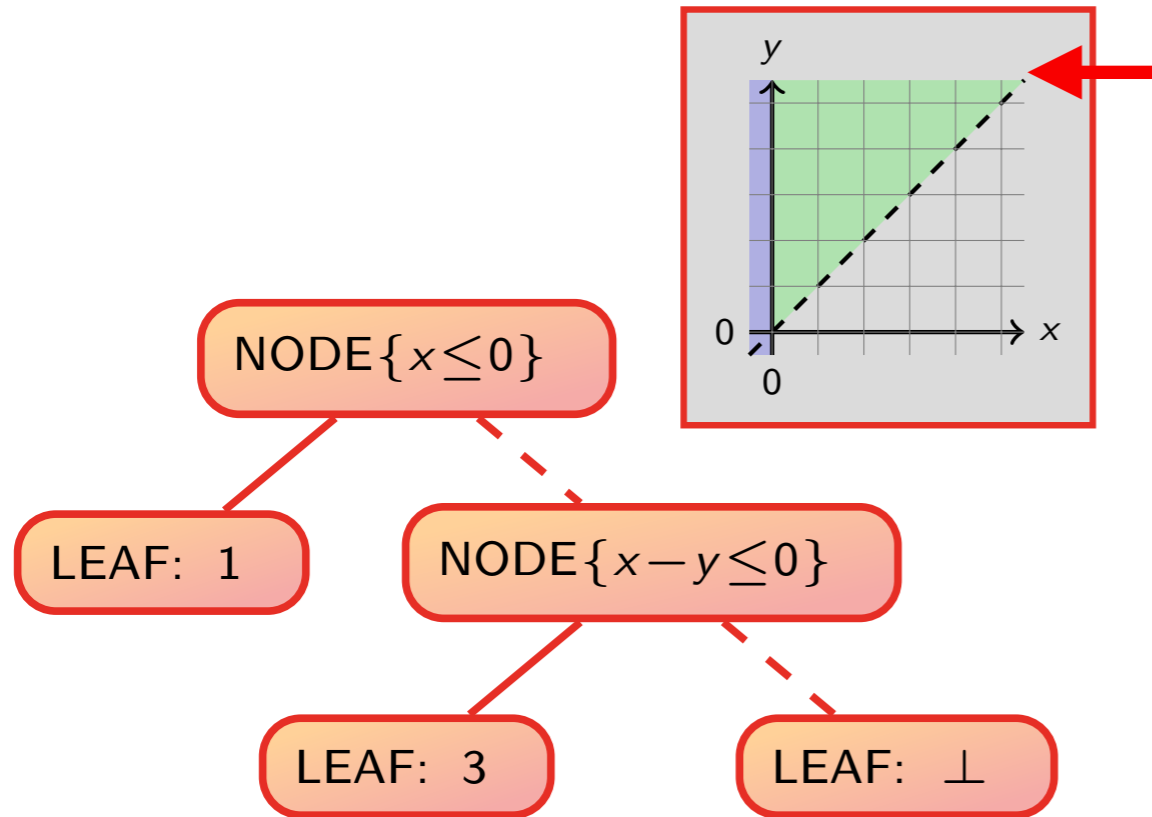
the **weakest precondition**
 is $x \leq 0 \vee y > 0$



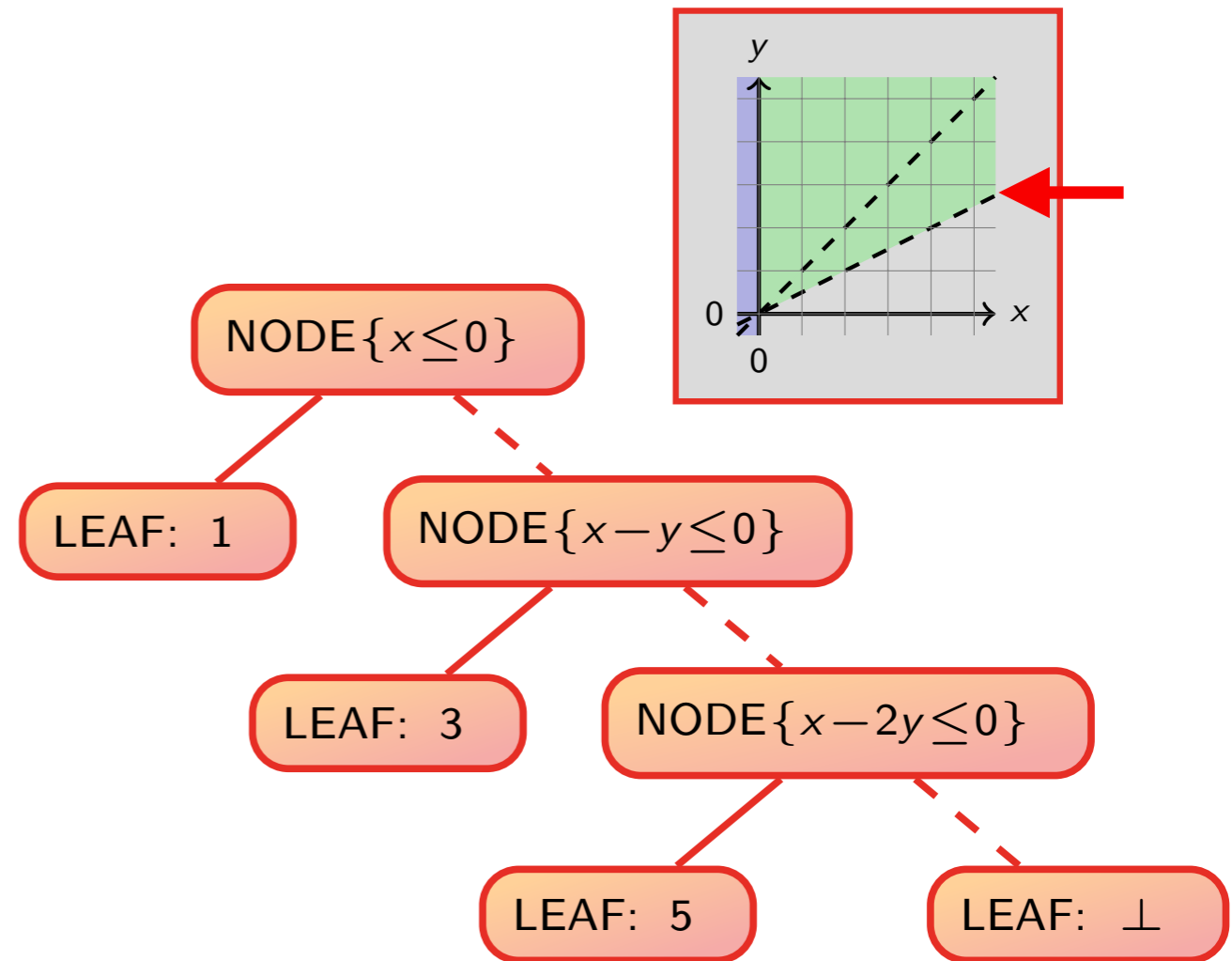
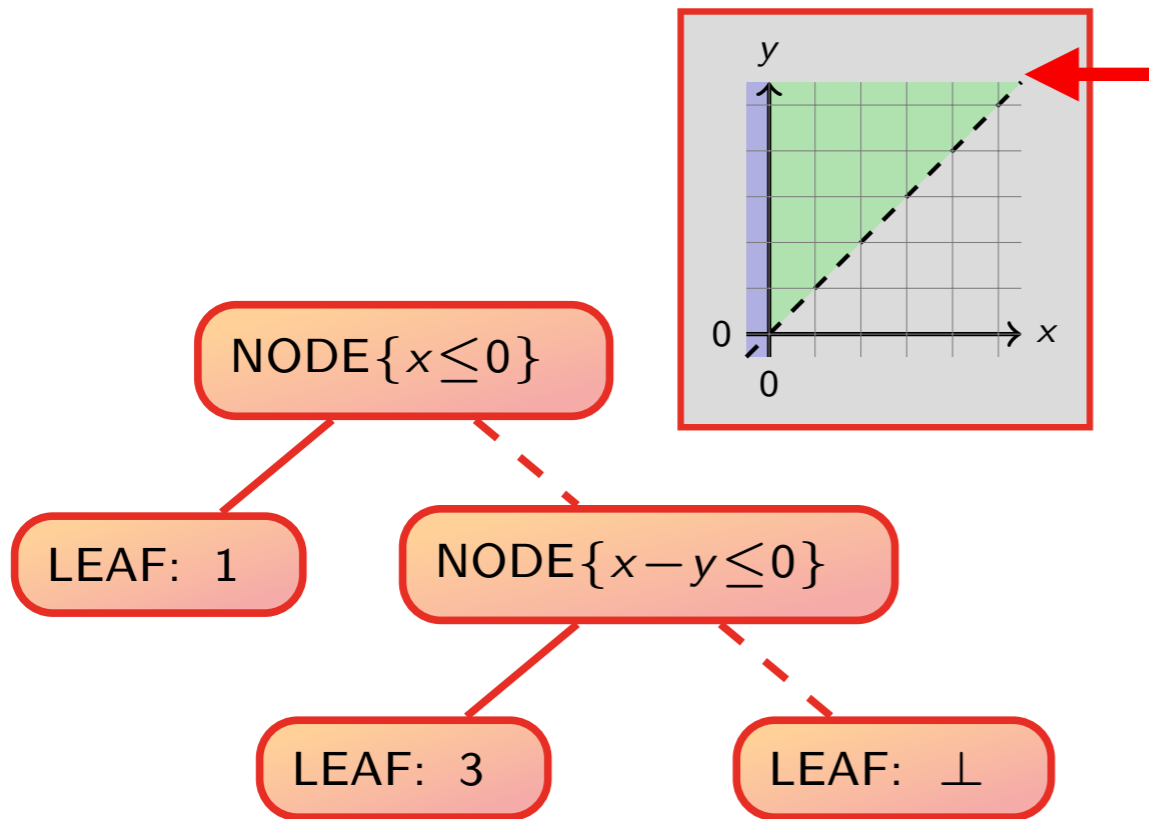
Widening



Widening



Widening



Precise Widening Operators for Convex Polyhedra*

Roberto Bagnara¹, Patricia M. Hill², Elisa Ricci¹, and Enea Zaffanella¹

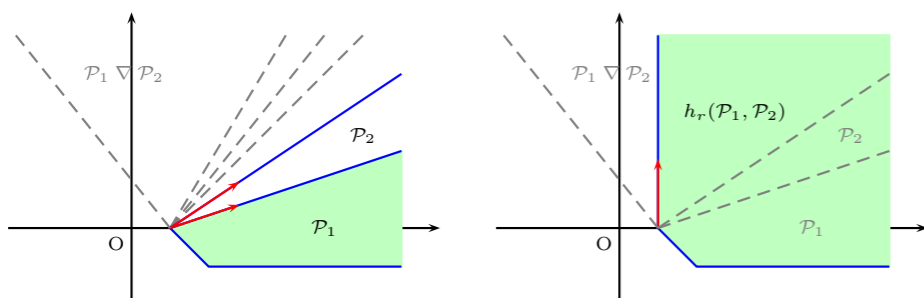
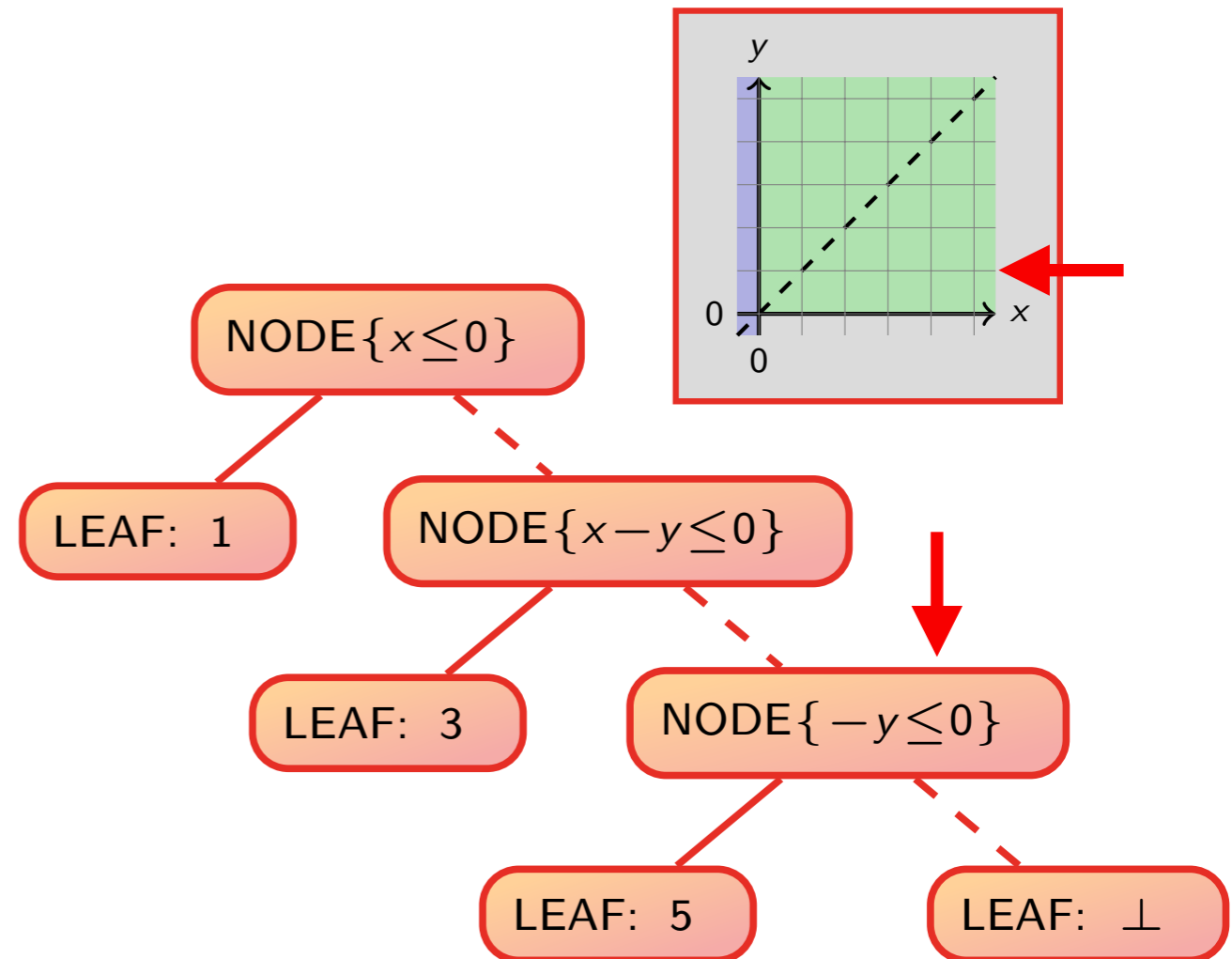
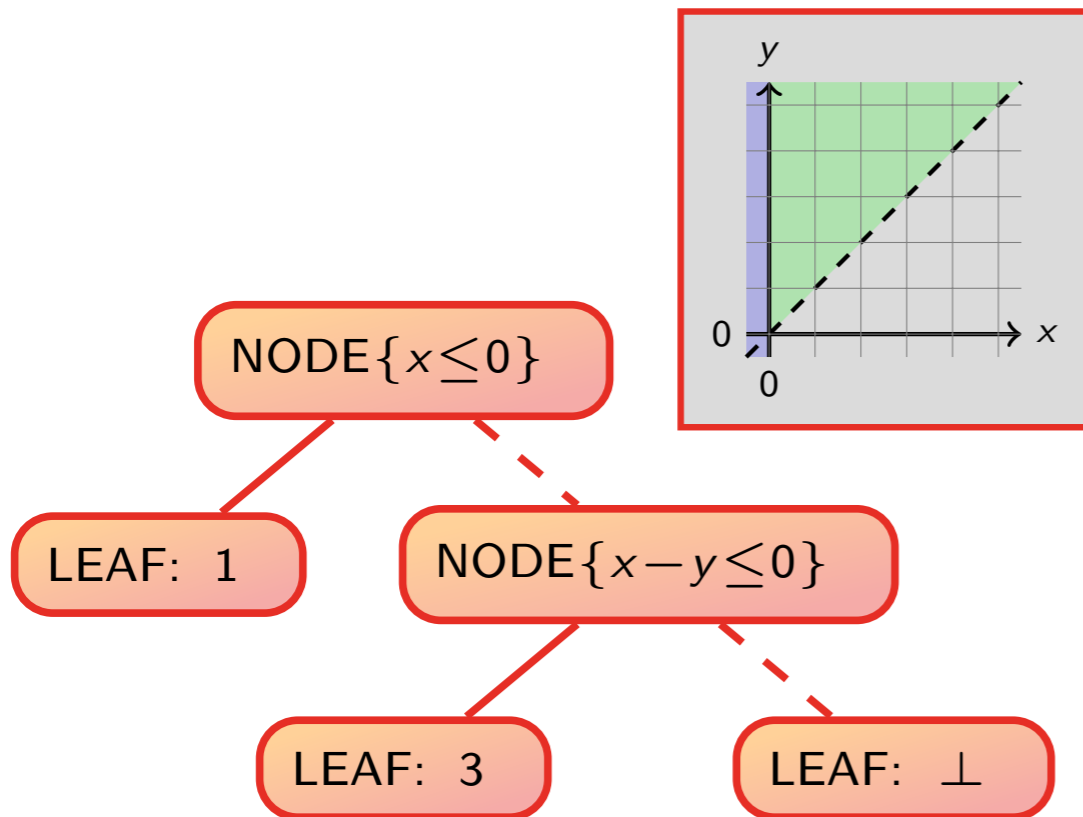


Fig. 2. The heuristics h_r improving on the standard widening.

Widening



Precise Widening Operators for Convex Polyhedra*

Roberto Bagnara¹, Patricia M. Hill², Elisa Ricci¹, and Enea Zaffanella¹

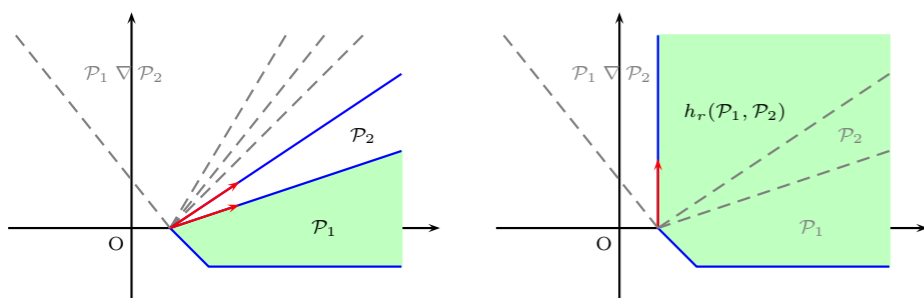
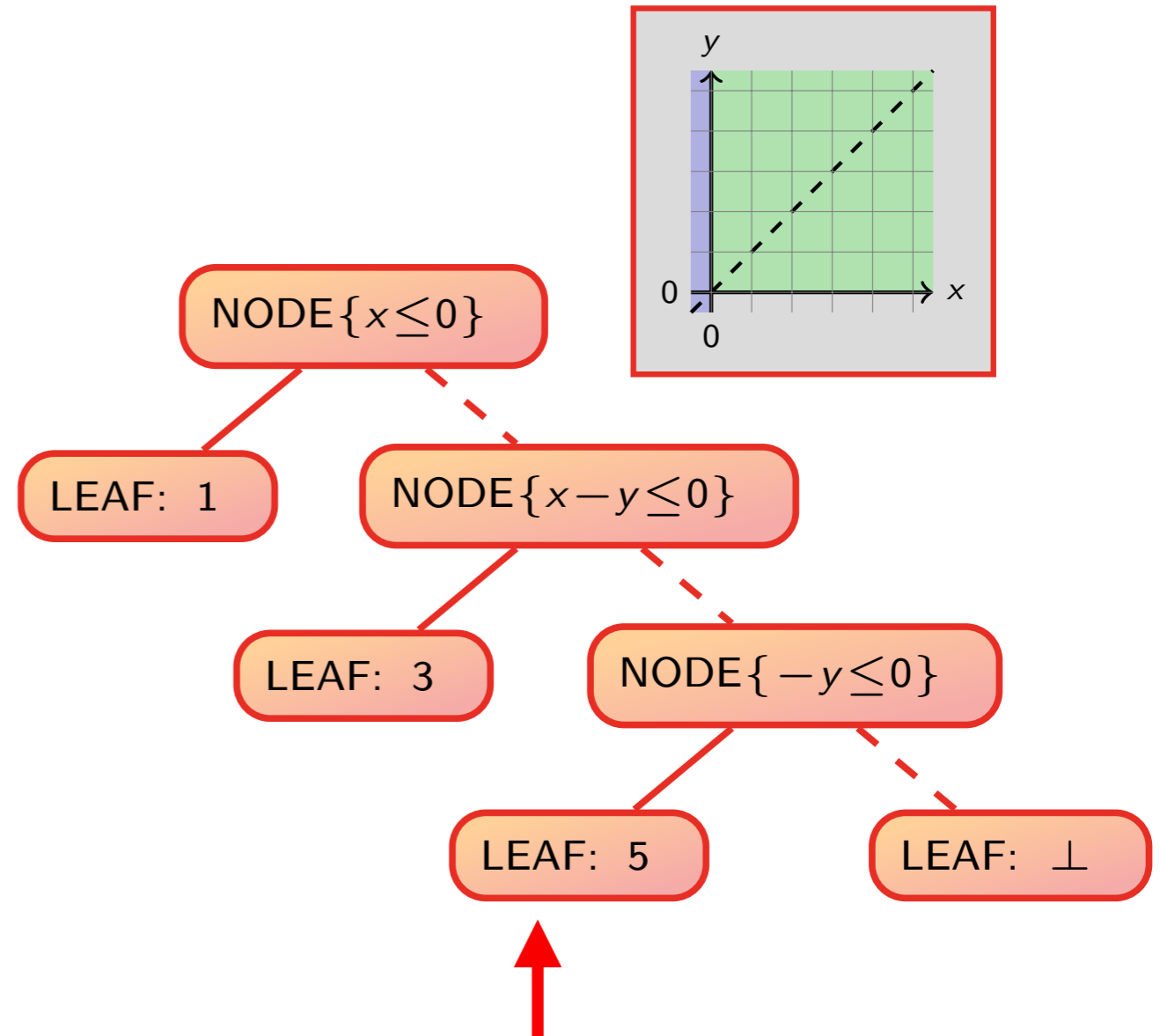
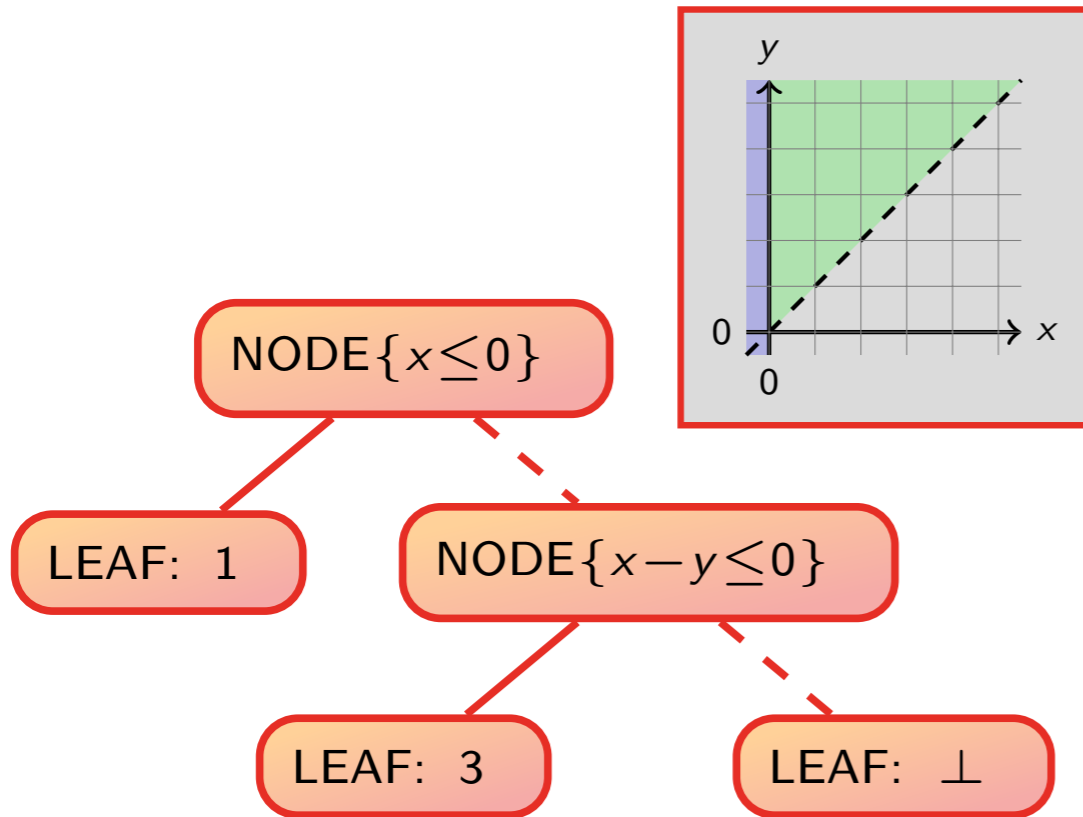
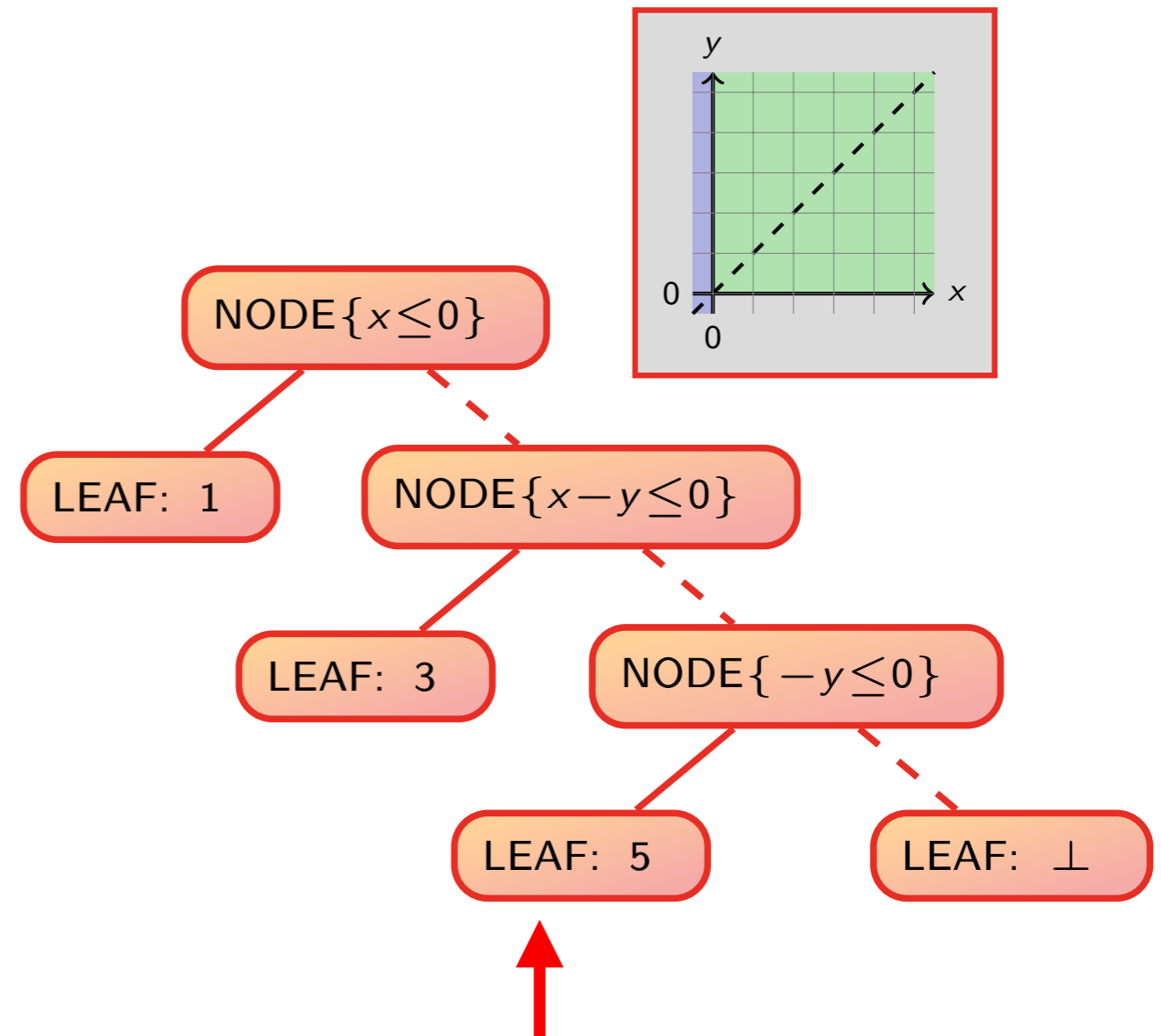
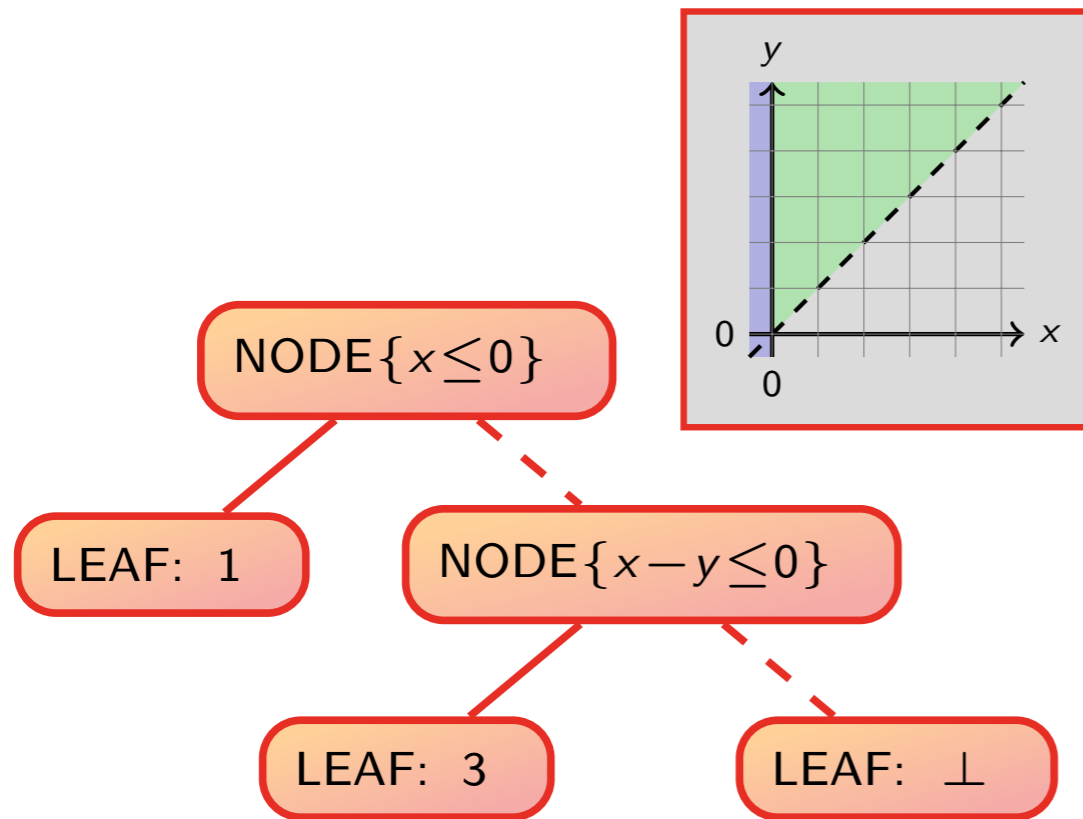


Fig. 2. The heuristics h_r improving on the standard widening.

Widening



Widening



The Abstract Domain of Segmented Ranking Functions

Caterina Urban

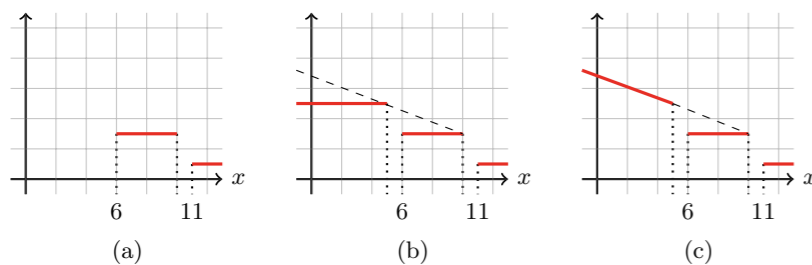
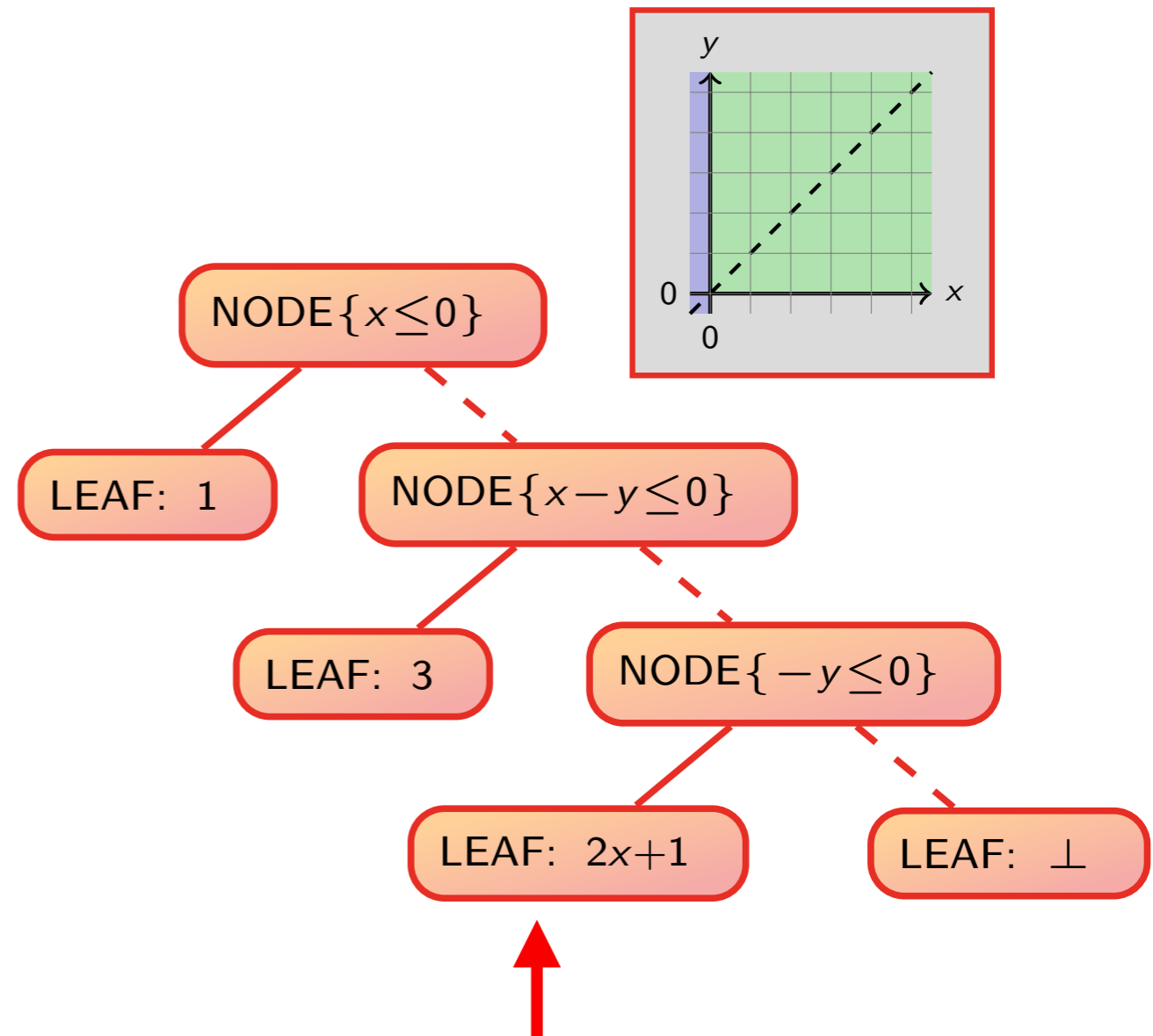
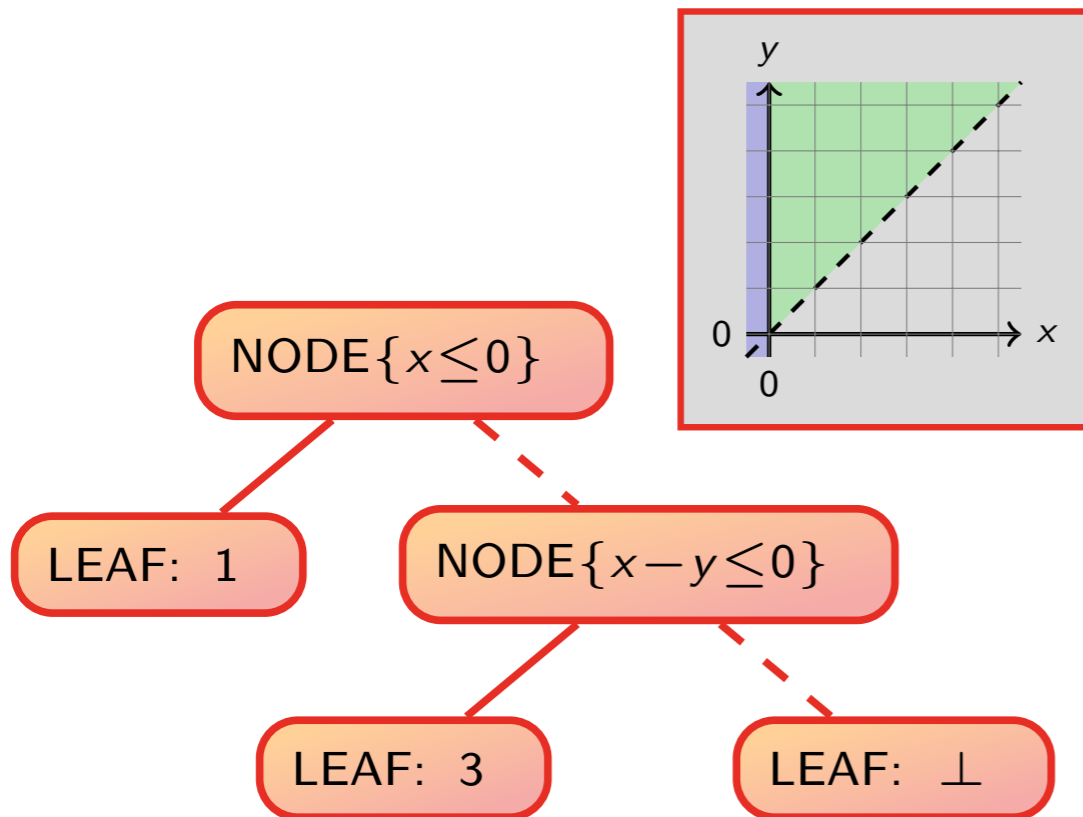


Fig. 7: Example of widening of abstract piecewise-defined ranking functions. The result of widening $v_1^\#$ (shown in (a)) with $v_2^\#$ (shown in (b)) is shown in (c).

Widening



The Abstract Domain of Segmented Ranking Functions

Caterina Urban

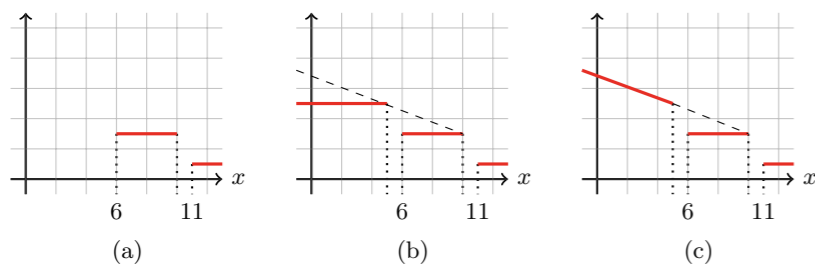
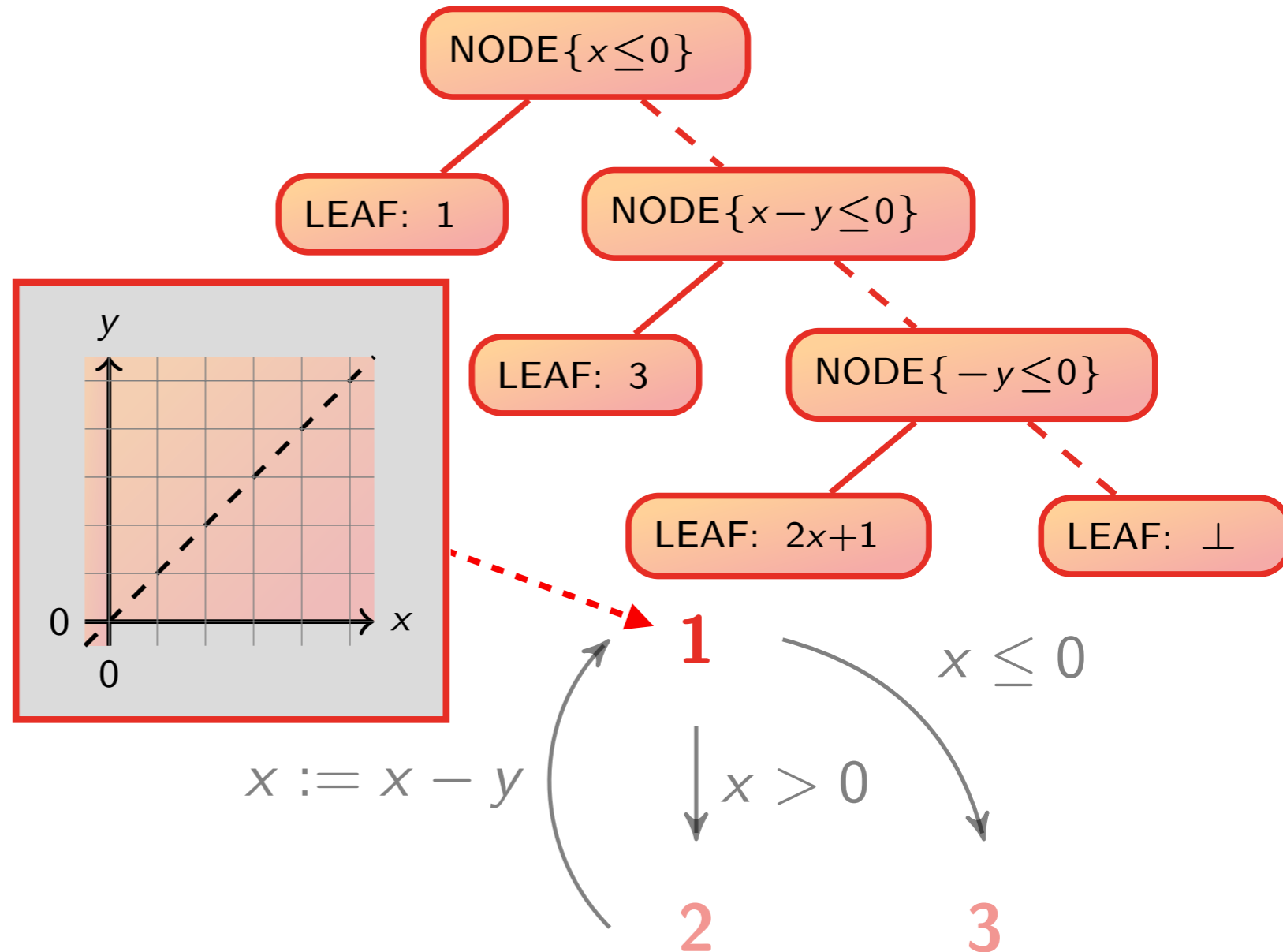


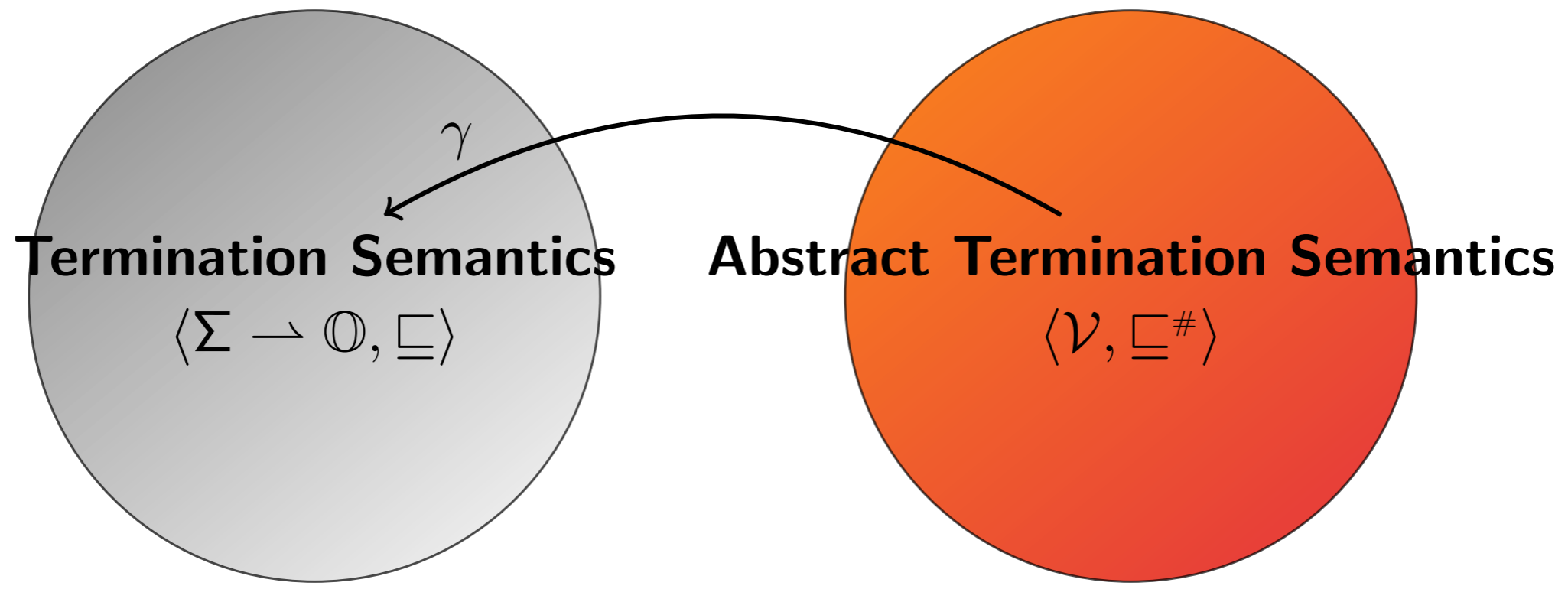
Fig. 7: Example of widening of abstract piecewise-defined ranking functions. The result of widening $v_1^\#$ (shown in (a)) with $v_2^\#$ (shown in (b)) is shown in (c).

Example

```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

the analysis gives the **weakest precondition** $x \leq 0 \vee y > 0$





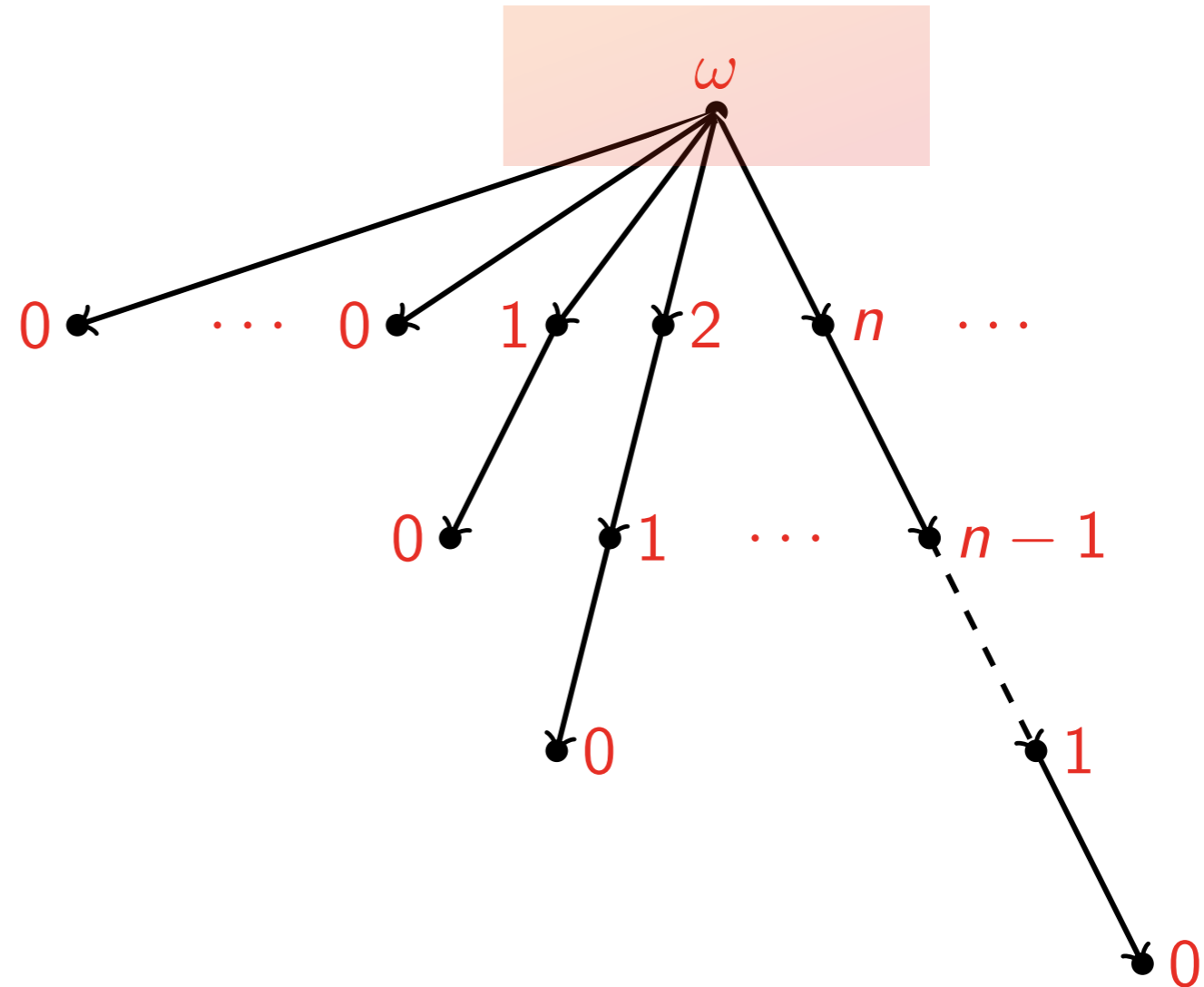
Theorem (**Soundness**)

*the abstract termination semantics is **sound**
to prove the termination of programs*

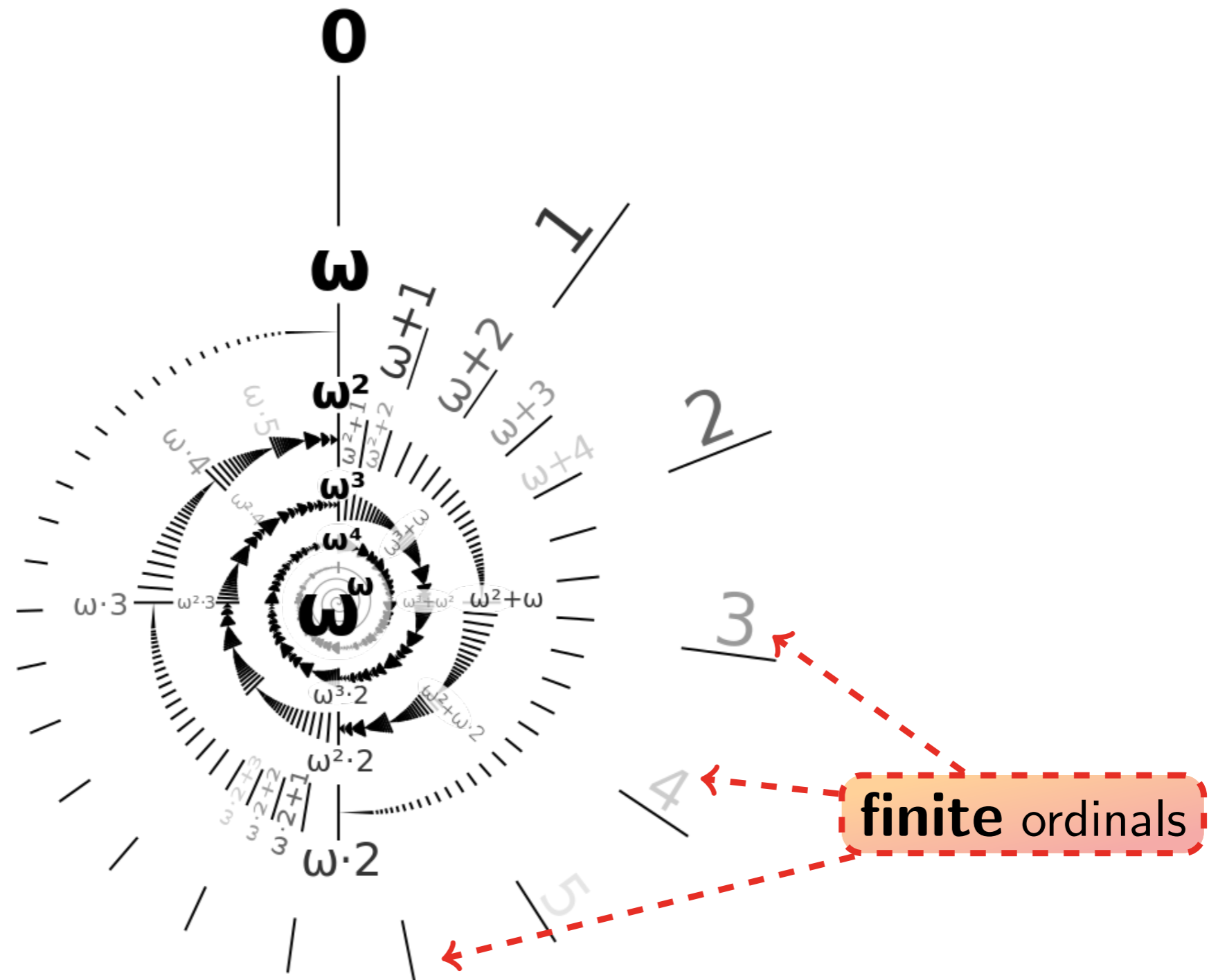
- **remark:** natural-valued ranking functions are **not sufficient!**

Example

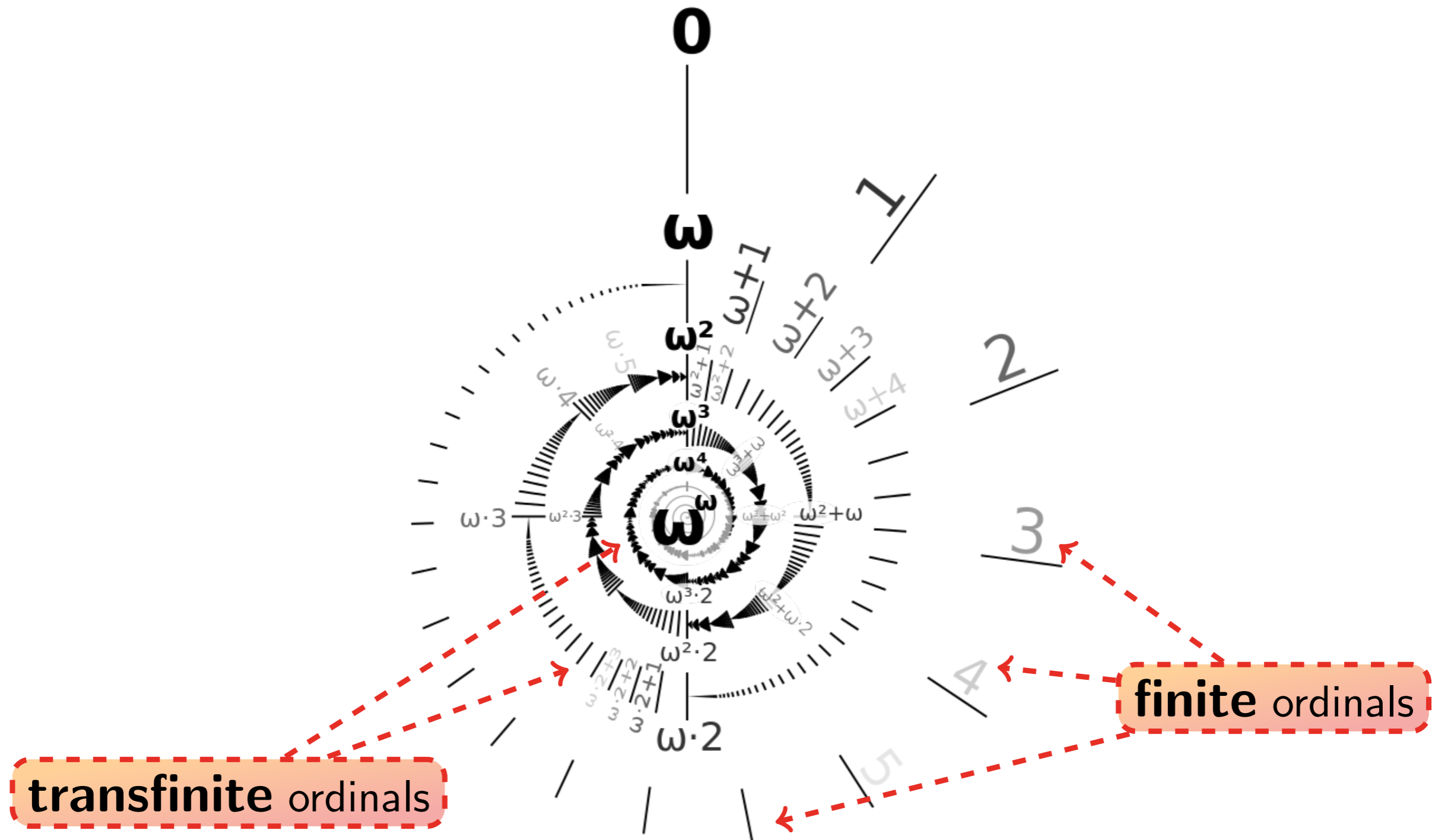
```
int : x  
x := ?  
while (x > 0) do  
  x := x - 1  
od
```



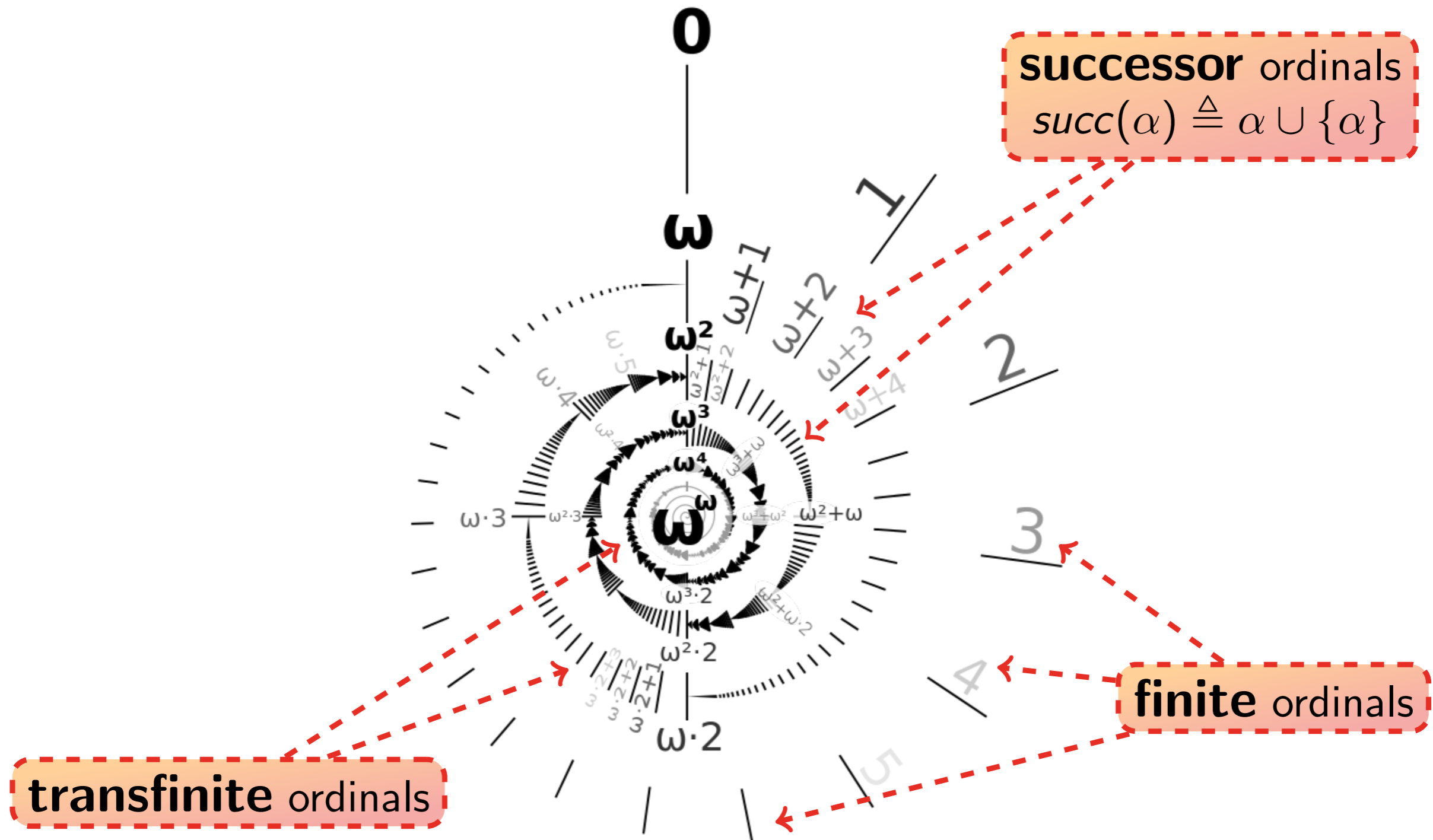
Ordinals



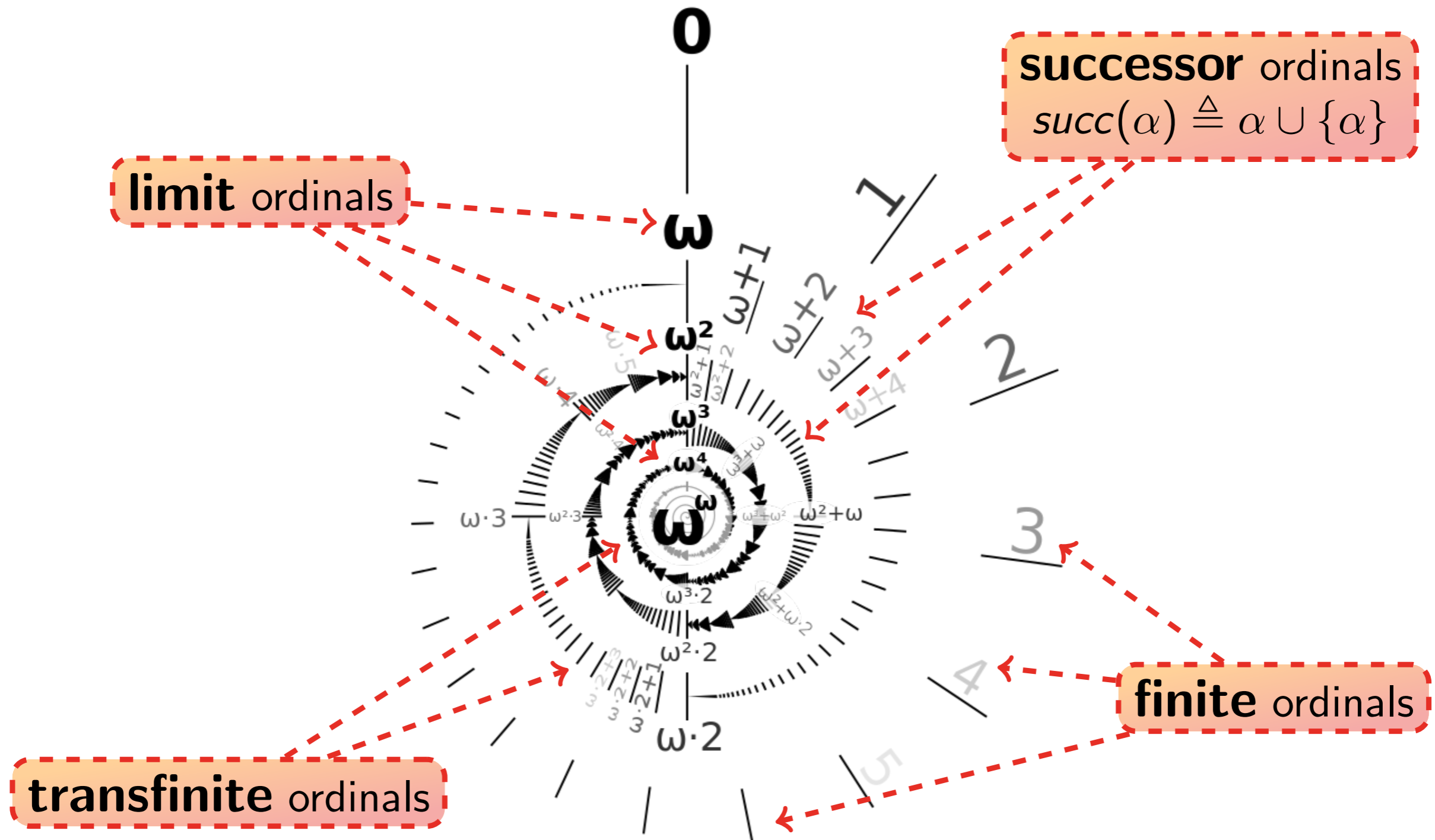
Ordinals



Ordinals



Ordinals



Ordinal Arithmetic

- **addition**

$$\alpha + 0 = \alpha \quad \text{(zero case)}$$

$$\alpha + \text{succ}(\beta) = \text{succ}(\alpha + \beta) \quad \text{(successor case)}$$

$$\alpha + \beta = \bigcup_{\gamma < \beta} (\alpha + \gamma) \quad \text{(limit case)}$$

- associative: $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$
- not commutative: $1 + \omega = \omega \neq \omega + 1$

- **multiplication**

Ordinal Arithmetic

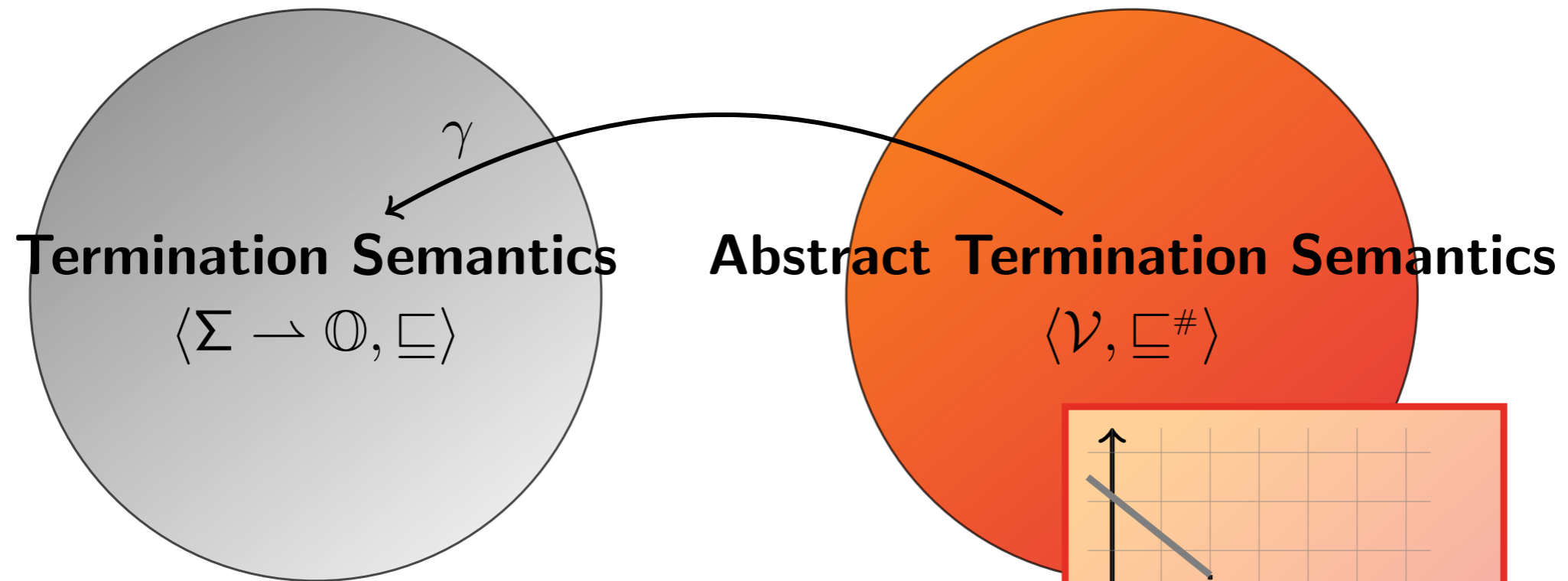
- addition
- multiplication

$$\alpha \cdot 0 = 0 \quad \text{(zero case)}$$

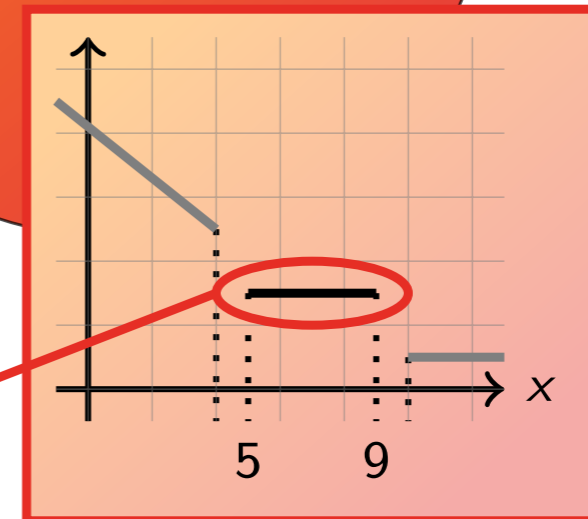
$$\alpha \cdot \text{succ}(\beta) = (\alpha \cdot \beta) + \alpha \quad \text{(successor case)}$$

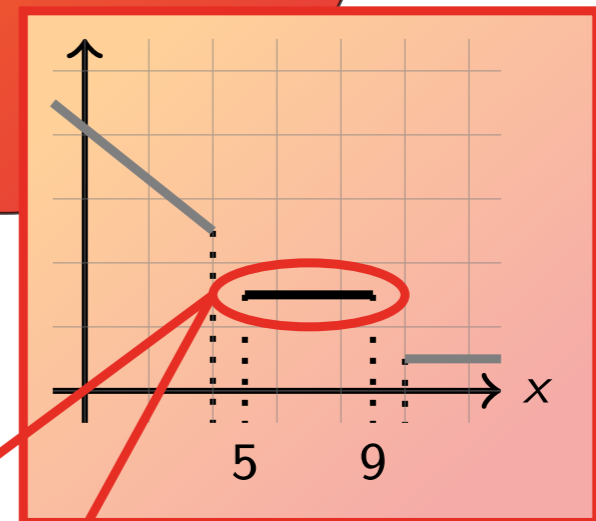
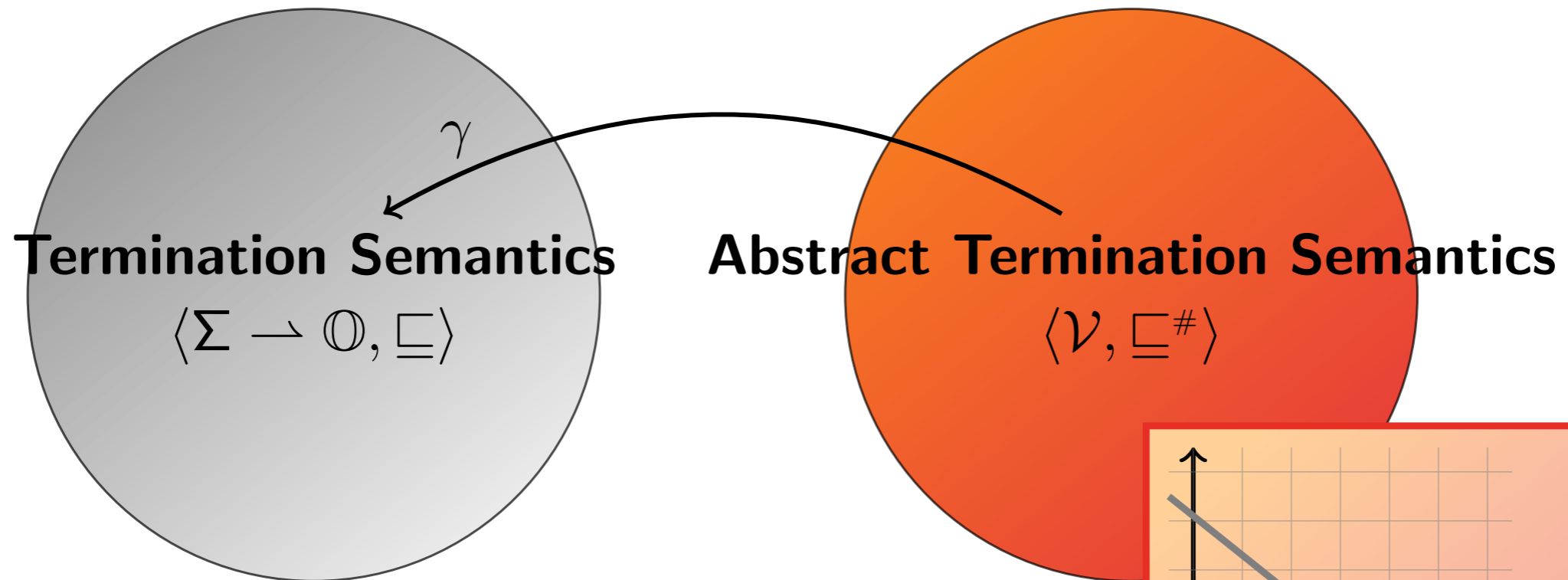
$$\alpha \cdot \beta = \bigcup_{\gamma < \beta} (\alpha \cdot \gamma) \quad \text{(limit case)}$$

- associative: $(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$
- left distributive: $\alpha \cdot (\beta + \gamma) = (\alpha \cdot \beta) + (\alpha \cdot \gamma)$
- not commutative: $2 \cdot \omega = \omega \neq \omega \cdot 2$
- not right distributive: $(\omega + 1) \cdot \omega = \omega \cdot \omega \neq \omega \cdot \omega + \omega$

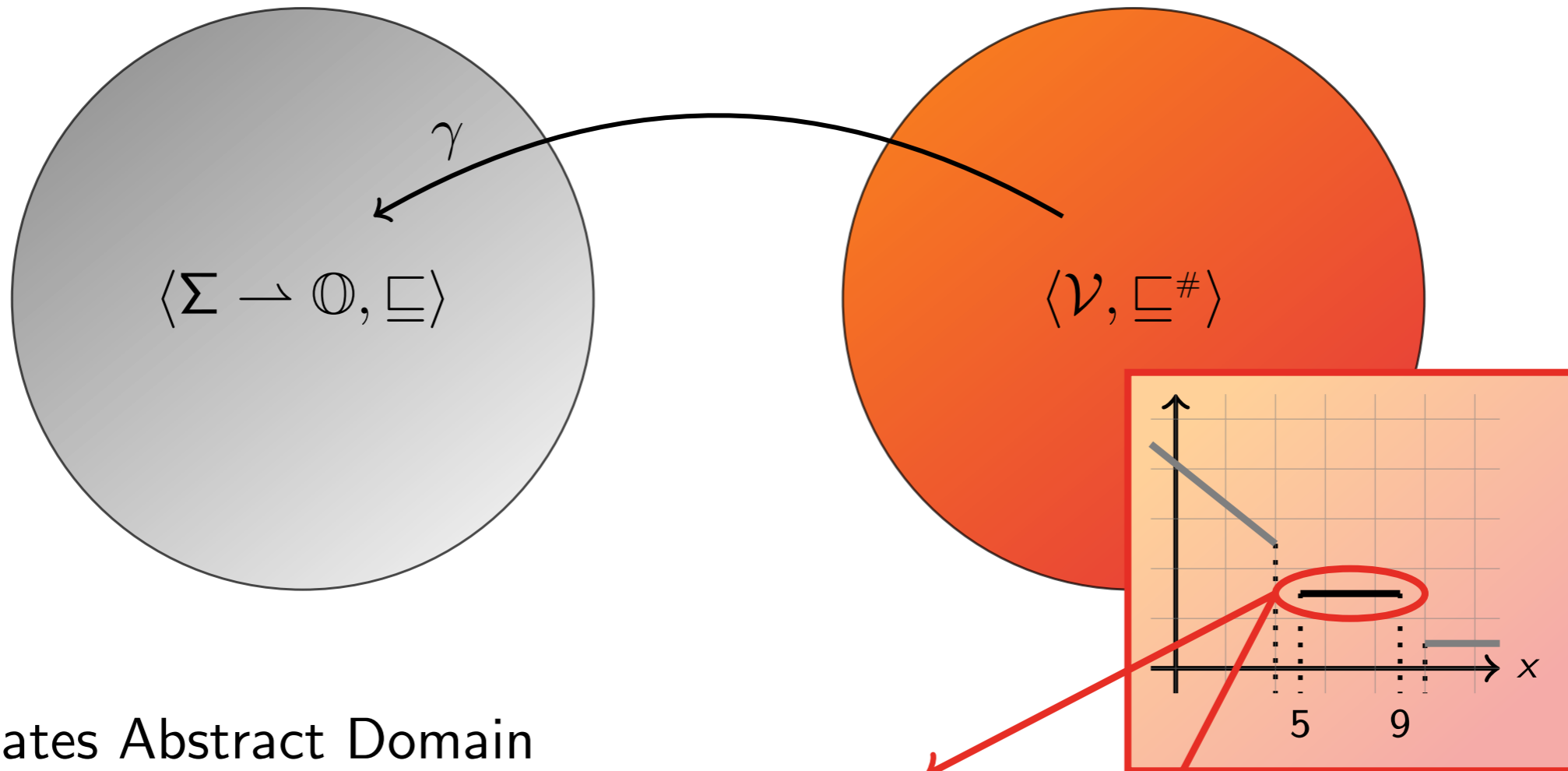


- States Abstract Domain
 - **Functions Abstract Domain**
 - Piecewise-Defined Ranking Functions Abstract Domain
- S
F
V(S, F)





- States Abstract Domain
- **Natural-Valued** Functions Abstract Domain S
- **Ordinal-Valued** Functions Abstract Domain F
- Piecewise-Defined Ranking Functions Abstract Domain $\mathbb{O}(F)$
- Piecewise-Defined Ranking Functions Abstract Domain $V(S, \mathbb{O}(F))$



- States Abstract Domain
- **Natural-Valued** Functions Abstract Domain
 - $\mathcal{F} \stackrel{\text{def}}{=} \{\perp\} \cup \{f \mid f \in \mathbb{Z}^n \rightarrow \mathbb{N}\} \cup \{\top\}$
 where $f \equiv f(x_1, \dots, x_n) = m_1x_1 + \dots + m_nx_n + q$
- **Ordinal-Valued** Functions Abstract Domain
 - $\mathcal{O} \stackrel{\text{def}}{=} \{\perp\} \cup \{\sum_i \omega^i \cdot f_i \mid f_i \in \mathcal{F} \setminus \{\perp, \top\}\} \cup \{\top\}$
- Piecewise-Defined Ranking Functions Abstract Domain

Backward Non-Deterministic Assignments

- non-deterministic assignments are carried out in ascending powers of ω

Example

$$[-\infty, +\infty] \mapsto \omega \cdot x_1 + x_2$$

↓ $x_1 := ?$

$$[-\infty, +\infty] \mapsto ?$$

Backward Non-Deterministic Assignments

- non-deterministic assignments are carried out in ascending powers of ω

Example

$$[-\infty, +\infty] \mapsto \omega \cdot x_1 + x_2$$

↓ $x_1 := ?$

$$[-\infty, +\infty] \mapsto + 1$$

Backward Non-Deterministic Assignments

- non-deterministic assignments are carried out in ascending powers of ω

Example

$$\begin{array}{rcl}
 [-\infty, +\infty] \mapsto & \omega \cdot x_1 & + x_2 \\
 & \Downarrow x_1 := ? & \\
 [-\infty, +\infty] \mapsto & & + x_2 + 1
 \end{array}$$

Backward Non-Deterministic Assignments

- non-deterministic assignments are carried out in ascending powers of ω

Example

$$[-\infty, +\infty] \mapsto \omega \cdot \mathbf{x}_1 + x_2$$

$\Downarrow \mathbf{x}_1 := ?$

$$[-\infty, +\infty] \mapsto \omega^2 \cdot \mathbf{1} + \omega \cdot \mathbf{0} + x_2 + 1$$

$$\omega \cdot \omega = \omega^2 \cdot \mathbf{1} + \omega \cdot \mathbf{0}$$

Backward Non-Deterministic Assignments

- non-deterministic assignments are carried out in ascending powers of ω

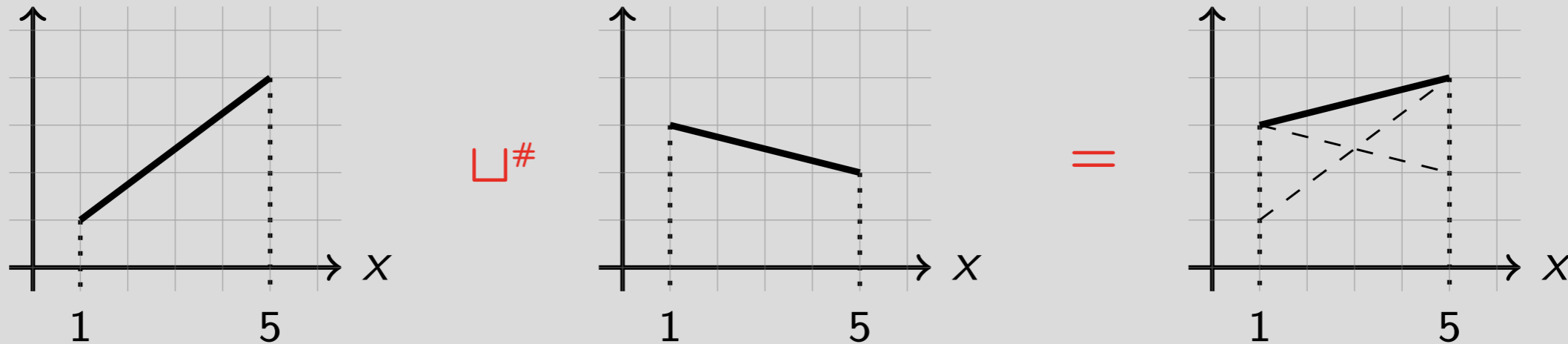
Example

$$\begin{array}{rcl}
 [-\infty, +\infty] \mapsto & \omega \cdot x_1 & + x_2 \\
 & \Downarrow x_1 := ? & \\
 [-\infty, +\infty] \mapsto & \omega^2 & + x_2 + 1
 \end{array}$$

Join

- join of natural-valued functions:

Example



- join of ordinal-valued functions:

Join

- join of natural-valued functions:
- join of ordinal-valued functions:
 - join of natural-valued functions in ascending powers of ω

Example

$$[-\infty, +\infty] \mapsto o_1 \equiv \omega^2 \cdot x_1 + \omega \cdot x_2 + 3$$

$$[-\infty, +\infty] \mapsto o_2 \equiv \omega^2 \cdot x_1 + \omega \cdot (-x_2) + 4$$

$$[-\infty, +\infty] \mapsto o_1 \sqcup^\# o_2 \equiv ?$$

Join

- join of natural-valued functions:
- join of ordinal-valued functions:
 - join of natural-valued functions in ascending powers of ω

Example

$$\begin{array}{l}
 [-\infty, +\infty] \mapsto o_1 \equiv \omega^2 \cdot x_1 + \omega \cdot x_2 + \mathbf{3} \\
 [-\infty, +\infty] \mapsto o_2 \equiv \omega^2 \cdot x_1 + \omega \cdot (-x_2) + \mathbf{4} \\
 \hline
 [-\infty, +\infty] \mapsto o_1 \sqcup^\# o_2 \equiv + \mathbf{4}
 \end{array}$$

Join

- join of natural-valued functions:
- join of ordinal-valued functions:
 - join of natural-valued functions in ascending powers of ω

Example

$[-\infty, +\infty]$	\mapsto	o_1	\equiv	$\omega^2 \cdot x_1$	$+$	$\omega \cdot x_2$	$+$	3
$[-\infty, +\infty]$	\mapsto	o_2	\equiv	$\omega^2 \cdot x_1$	$+$	$\omega \cdot (-x_2)$	$+$	4
$[-\infty, +\infty]$	\mapsto	$o_1 \sqcup^\# o_2$	\equiv	$\omega^2 \cdot 1$	$+$	$\omega \cdot 0$	$+$	4

$$\omega \cdot \omega = \omega^2 \cdot 1 + \omega \cdot 0$$

Join

- join of natural-valued functions:
- join of ordinal-valued functions:
 - join of natural-valued functions in ascending powers of ω

Example

$[-\infty, +\infty] \mapsto$	o_1	\equiv	$\omega^2 \cdot \mathbf{x_1}$	$+$	$\omega \cdot x_2$	$+$	3
$[-\infty, +\infty] \mapsto$	o_2	\equiv	$\omega^2 \cdot \mathbf{x_1}$	$+$	$\omega \cdot (-x_2)$	$+$	4
$[-\infty, +\infty] \mapsto$	$o_1 \sqcup^\# o_2$	\equiv	$\omega^2 \cdot \mathbf{x_1}$	$\omega^{2.1}$	$+$		4

Join

- join of natural-valued functions:
- join of ordinal-valued functions:
 - join of natural-valued functions in ascending powers of ω

Example

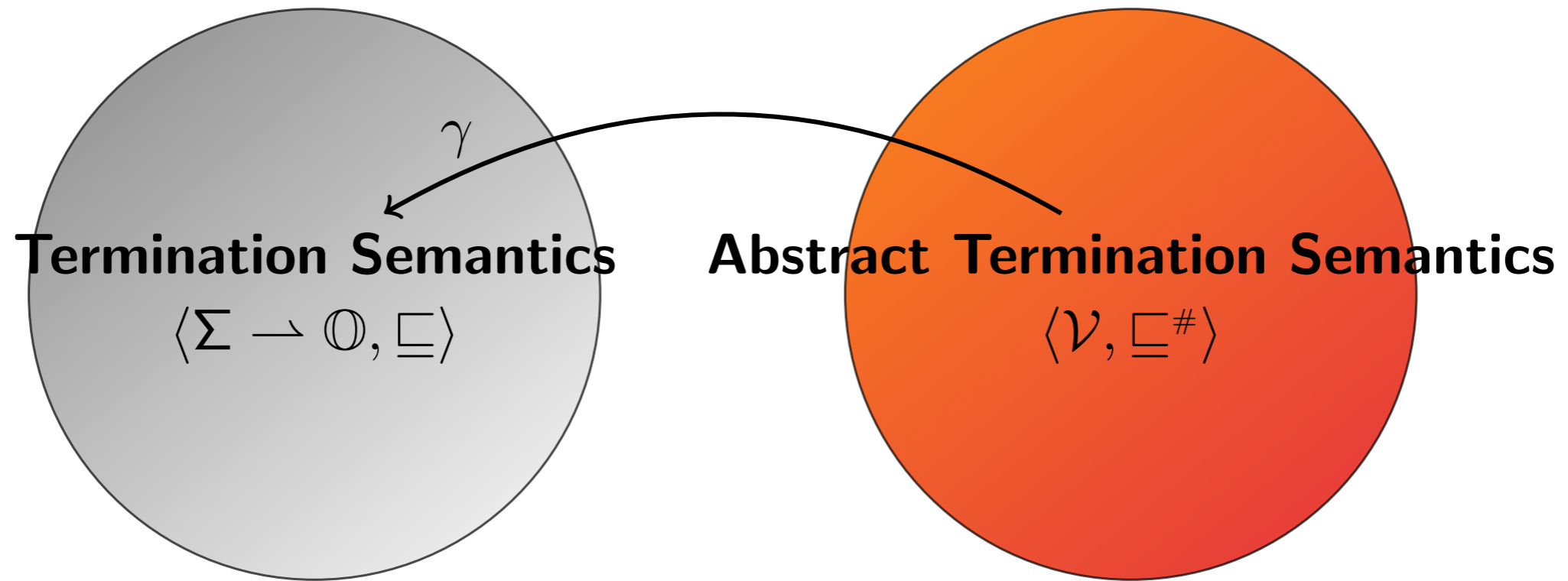
$[-\infty, +\infty] \mapsto$	o_1	\equiv	$\omega^2 \cdot \mathbf{x_1}$	$+$	$\omega \cdot x_2$	$+$	3
$[-\infty, +\infty] \mapsto$	o_2	\equiv	$\omega^2 \cdot \mathbf{x_1}$	$+$	$\omega \cdot (-x_2)$	$+$	4
$[-\infty, +\infty] \mapsto$	$o_1 \sqcup^\# o_2$	\equiv	$\omega^2 \cdot (\mathbf{x_1 + 1})$	$+$		$+$	4

Join

- join of natural-valued functions:
- join of ordinal-valued functions:
 - join of natural-valued functions in ascending powers of ω

Example

$$\begin{array}{l}
 [-\infty, +\infty] \mapsto o_1 \equiv \omega^2 \cdot x_1 + \omega \cdot x_2 + 3 \\
 [-\infty, +\infty] \mapsto o_2 \equiv \omega^2 \cdot x_1 + \omega \cdot (-x_2) + 4 \\
 \hline
 [-\infty, +\infty] \mapsto o_1 \sqcup^\# o_2 \equiv \omega^2 \cdot (x_1 + 1) + 4
 \end{array}$$



Theorem (**Soundness**)

*the abstract termination semantics is **sound**
to prove the termination of programs*

Example

```
int :  $x_1, x_2$   
while 1( $x_1 > 0 \wedge x_2 > 0$ ) do  
  if 2( ? ) then  
    3 $x_1 := x_1 - 1$   
    4 $x_2 := ?$   
  else  
    5 $x_2 := x_2 - 1$   
od6
```

$$f_1(x_1, x_2) = \begin{cases} 1 & x_1 \leq 0 \vee x_2 \leq 0 \\ \omega \cdot (x_1 - 1) + 7x_1 + 3x_2 - 5 & x_1 > 0 \wedge x_2 > 0 \end{cases}$$

Example

```

int : x1, x2
while 1(x1 ≠ 0 ∧ x2 > 0) do
  if 2(x1 > 0) then
    if 3( ? ) then
      4x1 := x1 - 1
      5x2 := ?
    else
      6x2 := x2 - 1
  else /* x1 < 0 */
    if 7( ? ) then
      8x1 := x1 + 1
    else
      9x2 := x2 - 1
      10x1 := ?
    od11

```

$$f_1(x_1, x_2) = \begin{cases} \omega^2 + \omega \cdot (x_2 - 1) - 4x_1 + 9x_2 - 2 & x_1 < 0 \wedge x_2 > 0 \\ 1 & x_1 = 0 \vee x_2 \leq 0 \\ \omega \cdot (x_1 - 1) + 9x_1 + 4x_2 - 7 & x_1 > 0 \wedge x_2 > 0 \end{cases}$$

Example

```

int : x1, x2
while 1(x1 ≠ 0 ∧ x2 > 0) do
  if 2(x1 > 0) then
    if 3( ? ) then
      4x1 := x1 - 1
      5x2 := ?
    else
      6x2 := x2 - 1
  else /* x1 < 0 */
    if 7( ? ) then
      8x1 := x1 + 1
    else
      9x2 := x2 - 1
      10x1 := ?
    
```

the coefficients and their **order** are inferred by the analysis

$$f_1(x_1, x_2) = \begin{cases} \omega^2 + \omega \cdot (x_2 - 1) - 4x_1 + 9x_2 - 2 & x_1 < 0 \wedge x_2 > 0 \\ 1 & x_1 = 0 \vee x_2 \leq 0 \\ \omega \cdot (x_1 - 1) + 9x_1 + 4x_2 - 7 & x_1 > 0 \wedge x_2 > 0 \end{cases}$$

Non-Linear Ranking Functions

Example

int : N , x_1 , x_2

¹ $x_1 := N$

while ² ($x_1 \geq 0$) do

³ $x_2 := N$

 while ⁴ ($x_2 \geq 0$) do

⁵ $x_2 := x_2 - 1$

 od⁶

⁷ $x_1 := x_1 - 1$

od⁸

$$f_1(x_1, x_2, N) = \begin{cases} 1 & x_1 < 0 \\ \omega \cdot (x_1 + 1) + 6x_1 + 7 & x_1 \geq 0 \end{cases}$$

Non-Linear Ranking Functions

Example

```

int : N, x1, x2
1x1 := N
while 2(x1 ≥ 0) do
  3x2 := N
  while 4(x2 ≥ 0) do
    5x2 := x2 - 1
  od6
  7x1 := x1 - 1
od8
    
```

$$f_1(x_1, x_2, N) = \begin{cases} 1 & x_1 < 0 \\ \omega \cdot (x_1 + 1) + 6x_1 + 7 & x_1 \geq 0 \end{cases}$$

the loop terminates in a
finite number of iterations

FuncTion

www.di.ens.fr/~urban/FuncTion.html

Home Page Papers Talks/Posters **FuncTion**

An Abstract Domain Functor for Termination

Welcome to FuncTion's web interface!

Type your program:

or choose a predefined example: Choose File

and choose an entry point:

Forward option(s):

- Widening delay:

Backward option(s):

- Partition Abstract Domain: Intervals
- Function Abstract Domain: Affine Functions
- Ordinal-Valued Functions
 - Maximum Degree:
- Widening delay:

Guarantee Semantics

Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation*

Cecilia Urbas and Antoine Mine
INRIA & CNRS & UMR5, France
{urbas,mine}@inria.fr

Abstract. We present new static analysis methods for proving liveness properties of programs. In particular, with reference to the hierarchy of temporal properties proposed by Maass and Paoletti, we focus on guarantee (i.e., “something good occurs at least once”) and recurrence (i.e., “something good occurs infinitely often”) temporal properties. We generalize the abstract interpretation framework for liveness proposed by Cousot and Cousot. Specifically, static analysis of guarantee and recurrence temporal properties are automatically derived by abstraction of the program operational trace semantics. These methods automatically infer sufficient preconditions for the temporal properties by solving existing numerical abstract domains based on glushko-defined matching functions. We suggest these abstract domains with new abstract operators, including a dual ordering. To illustrate the potential of the proposed methods, we have implemented a research prototype static analyzer, for programs written in a C-like syntax, that yielded interesting preliminary results.

1 Introduction

Temporal properties play a major role in the specification and verification of programs. The hierarchy of temporal properties proposed by Maass and Paoletti [2] distinguishes four basic classes:

- safety properties: “something good always happens”, i.e., the program never reaches an unacceptable state (e.g., partial correctness, mutual exclusion);
- guarantee properties: “something good happens at least once”, i.e., the program eventually reaches a desirable state (e.g., termination);
- recurrence properties: “something good happens infinitely often”, i.e., the program reaches a desirable state infinitely often (e.g., starvation freedom);
- persistence properties: “something good eventually happens continuously”.

This paper concerns the verification of programs by static analysis. We set our work in the framework of Abstract Interpretation [3], a general theory of semantic

* The research leading to these results has received funding from the ANR/INRIA Joint Undertaking under grant agreement no. 194516 (ARTEMIS project MDAAT) (see Article 17.9 of the JU Grant Agreement).



program \mapsto maximal trace semantics \rightarrow φ -guarantee semantics

$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_g^\varphi \stackrel{\text{def}}{=} \text{lfp } F_g^\varphi$$

$$F_g^\varphi(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 & s \models \varphi \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \wedge s \not\models \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

idea = define a ranking function **counting the number of program steps** from the property φ

program \mapsto maximal trace semantics \rightarrow φ -guarantee semantics

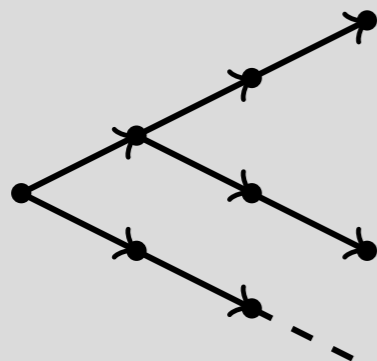
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_g^\varphi \stackrel{\text{def}}{=} \text{lfp } F_g^\varphi$$

$$F_g^\varphi(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 & s \models \varphi \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \wedge s \not\models \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

idea = define a ranking function **counting the number of program steps** from the property φ

Example



program \mapsto maximal trace semantics \rightarrow φ -guarantee semantics

$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_g^\varphi \stackrel{\text{def}}{=} \text{lfp } F_g^\varphi$$

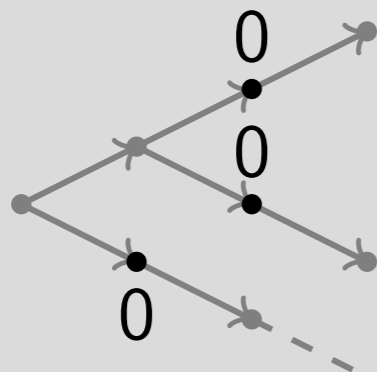
$$F_g^\varphi(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} \\ \text{undefined} \end{cases}$$

$$s \models \varphi \leftarrow$$

$$s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \wedge s \not\models \varphi \\ \text{otherwise}$$

idea = define a ranking function **counting the number of program steps** from the property φ

Example



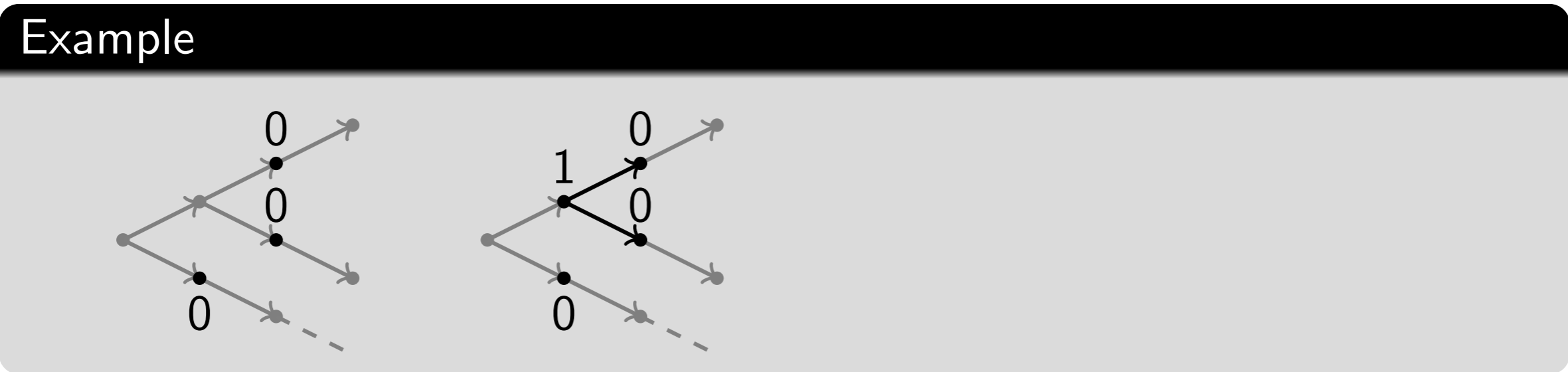
program \mapsto maximal trace semantics \rightarrow φ -guarantee semantics

$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_g^\varphi \stackrel{\text{def}}{=} \text{lfp } F_g^\varphi$$

$$F_g^\varphi(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 & s \models \varphi \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \wedge s \not\models \varphi \leftarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

idea = define a ranking function **counting the number of program steps** from the property φ



program \mapsto maximal trace semantics \rightarrow φ -guarantee semantics

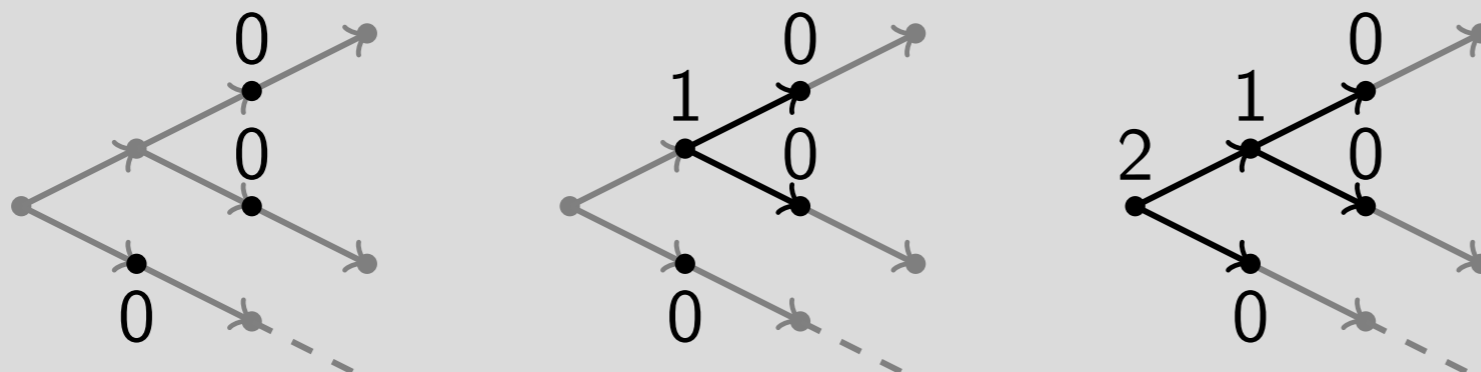
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_g^\varphi \stackrel{\text{def}}{=} \text{lfp } F_g^\varphi$$

idea = define a ranking function **counting the number of program steps** from the property φ

$$F_g^\varphi(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 & s \models \varphi \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \wedge s \not\models \varphi \leftarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



program \mapsto maximal trace semantics \rightarrow φ -guarantee semantics

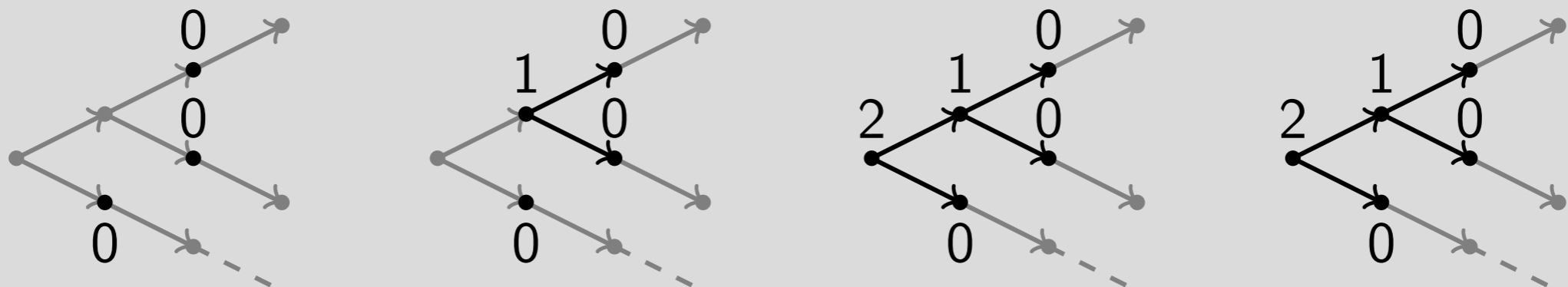
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_g^\varphi \stackrel{\text{def}}{=} \text{lfp } F_g^\varphi$$

$$F_g^\varphi(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 & s \models \varphi \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \wedge s \not\models \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

idea = define a ranking function **counting the number of program steps** from the property φ

Example



program \mapsto maximal trace semantics \rightarrow φ -guarantee semantics

$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

$$\mathcal{T}_g^\varphi \stackrel{\text{def}}{=} \text{lfp } F_g^\varphi$$

$$F_g^\varphi(\mathbf{v})s \stackrel{\text{def}}{=} \begin{cases} 0 & s \models \varphi \\ \sup\{ \mathbf{v}(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(\mathbf{v})) \wedge s \not\models \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

Theorem (Soundness and Completeness)

the φ -guarantee semantics is **sound** and **complete**
 to prove the guarantee property $\diamond\varphi$

Bibliography

An Abstract Interpretation Framework for
Patrick Cousot
CNRS, Ecole Nationale Supérieure de l'Aviation, France
cousot@cict.fr

Abstract. This paper presents a general framework for abstract interpretation of programs. It is based on the notion of a concrete domain, which is a lattice with a least element and a greatest element. The abstract domain is a lattice with a least element and a greatest element, and a mapping from the concrete domain to the abstract domain. The abstract domain is used to approximate the concrete domain. The abstract domain is used to approximate the concrete domain. The abstract domain is used to approximate the concrete domain.

The Abstract Domain of Segmented Ranking Functions
Cécilia Uthais
ENS & CNRS & INRIA, Paris, France
uthais@cict.fr

Abstract. We present a parameterized abstract domain for proving program termination by abstract interpretation. The domain abstractly represents concrete-defined ranking functions and values sufficient for proving program termination. The analysis uses over-approximations that are precise for non-terminating, meaning that all program execution segments that do not terminate are not-terminating.

An Abstract Domain to Infer Ordinal-Valued Ranking Functions*
Cécilia Uthais and Antoine Miné
ENS & CNRS & INRIA, Paris, France
uthais@cict.fr, mine@cict.fr

Abstract. The traditional method for proving program termination consists in inferring a ranking function. In many cases (in programs with unbounded non-termination), a single ranking function may not be sufficient. Hence, we propose a new abstract domain to automatically infer ranking functions over ordinals.

A Decision Tree Abstract Domain for Proving Conditional Termination*
Cécilia Uthais and Antoine Miné
ENS & CNRS & INRIA, Paris, France
uthais@cict.fr, mine@cict.fr

Abstract. We present a new parameterized abstract domain able to infer on being executed abstract domains with finite disjunction. The domain of the abstract domain is a decision tree where the decision nodes are labeled with finite ordinals, and the leaf nodes being a bounded abstract domain.

Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation*
Cécilia Uthais and Antoine Miné
ENS & CNRS & INRIA, Paris, France
uthais@cict.fr, mine@cict.fr

Abstract. We present new static analysis methods for proving liveness properties of programs. In particular, with reference to the theory of abstract interpretation, we propose a new abstract domain for proving liveness properties. The abstract domain is used to approximate the concrete domain. The abstract domain is used to approximate the concrete domain.

- **Cousot & Cousot** - *An Abstract Interpretation Framework for Termination* (POPL 2012)
- **Urban** - *The Abstract Domain of Segmented Ranking Functions* (SAS 2013)
- **Urban & Miné** - *An Abstract Domain to Infer Ordinal-Valued Ranking Functions* (ESOP 2014)
- **Urban & Miné** - *A Decision Tree Abstract Domain for Proving Conditional Termination* (SAS 2014)
- **Urban & Miné** - *Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation* (VMCAI 2015)