

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

A screenshot of a web browser displaying a TechCrunch article. The title of the article is "30GB Zunes all over the world fail en masse". The article was posted on December 31, 2008, by Matt Burns (@mjburnsy). The text of the article describes a bug affecting Zune 30s, where they stop working and require a full reboot. It mentions fan boards and support forums. An update at the bottom states that letting the Zune run out of battery and charging it will fix the issue.

30GB Zunes all over the world fail en masse

Posted Dec 31, 2008 by Matt Burns (@mjburnsy)

It seems that a random bug is affecting a bunch, if not every, 30GB [Zunes](#). Real early this morning, a bunch of Zune 30s just stopped working. No official word from Redmond on this one yet but we might have a gadget Y2K going on here. [Fan boards](#) and [support forums](#) all have the same mantra saying that at 2:00 AM this morning, the Zune 30s reset on their own and doesn't fully reboot. We're sure Microsoft will get flooded with angry Zune owners as soon as the phone lines open up for the last time in 2008. More as we get it.

Update 2: The solution is ... kind of weak: let your Zune run out of battery and it'll be fixed when you wake up tomorrow and charge it.

- failure due to **non-termination**

The Zune Bug

The screenshot shows a web browser window with the URL techcrunch.com/2008/12/31/zune-bug-explained-in-detail/. The main content is an article titled "Zune bug explained in detail" posted by Devin Coldewey on Dec 31, 2008. The article discusses a bug in Zune devices where they would cry out in terror due to a self-fixing error. It includes a code snippet:

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

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

Below the code, a note says: "You can see the details here, but the important bit is that today, the day count is 366. As you".

On the left side of the browser window, there is a sidebar with other news articles and a navigation bar.

The Zune Bug

The screenshot shows a web browser window with the URL techcrunch.com/2008/12/31/zune-bug-explained-in-detail/. The main content is an article titled "Zune bug explained in detail" by Devin Coldewey, posted on Dec 31, 2008. The article discusses a bug in Zune devices where the day count would overflow from 365 to 366, causing them to reboot. Below the article is a code snippet in C:

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

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

A red circle highlights the line `days -= 366;`. At the bottom of the page, there is a note: "You can see the details here, but the important bit is that today, the day count is 366. As you".

On the left side of the browser window, there is a sidebar with other news items and a navigation bar.

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



Primer, 1949, June.

Checking a Large Routine, by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Consider the analogy of checking an addition. If it is given as:

1375
2966
4729
6337
7748

26304

one must check the whole at one sitting, because of the carries. But if the totals for the various columns are given, as below:

1375
5906
6729
6337
7748

26304

the checker's work is much easier being split up into the checking of the various assertions $3 + 9 = 7 + 3 = 29$ etc., and the small addition

375
2213

26304

This principle can be applied to the process of checking a large routine but we will illustrate the method by means of a small routine via, one to obtain $n!$ without the use of a multiplier, multiplication being carried out by repeated addition.

All a typical moment of the process we have recorded x and $x \cdot x$ for some x , i.e. we can change x to $x \cdot (x+1)$ by addition of x . Now $x = x \cdot 1$ we can change x to $x \cdot 2$ by a transfer. Unfortunately there is no coding system sufficiently generally known to justify giving the routine for this process in full, but the flow diagram given in Fig. 1 will be sufficient for illustration.

Each line of the flow diagram represents a straight sequence of instructions without changes of control. The following convention is used:

- (1) a dashed letter indicates the value at the end of the process represented by the box;
- (2) an undashed letter represents the initial value of a quantity.

One cannot equate similar letters appearing in different boxes, but it is intended that the following identifications be valid throughout

63.

Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS¹

Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination. The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated preposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_0 , the final values on completion will satisfy the relation R_F ." Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely.

These modes of proof of correctness and termination are not original; they are based on ideas of Perls and Gorn, and may have made their earliest appearance in an unpublished paper by Gorn. The establishment of formal standards for proofs about programs in languages which admit assignments, transfer of control, etc., and the proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigor for proofs about

¹This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

19



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”)
- $\sigma \in \mathcal{B} \implies f = \min \mathcal{S}$;
- $\sigma \rightarrow \sigma' \implies f(\sigma') \sqsubset f(\sigma)$.

(f counts the number of steps remaining before termination)

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

An Abstract Interpretation Framework for Termination

Patrick Cousot

CNRS, Ecole Normale Supérieure, and INRIA, France
Carnegie Mellon*, XPLAT, USA
cousot@cs.cmu.edu

Radhia Cousot

CNRS, Ecole Normale Supérieure, and INRIA, France
cousot@ens.fr

Abstract

Proof, verification and analysis methods for termination all rely on two induction principles: (1) a certain iteration or induction on time-measuring progress bounds 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 termination. The principle is then applied to the termination analysis of programs (a first expression in a context) and to the termination analysis of programs (a first expression in a context) and to the termination analysis of programs (a first expression in a context). Safety proofs and check/repair methods like those described below by our respective students. Static analysis of abstract syntax programs and their applications to termination analysis, verification and synthesis of approximation (or automatically) infer safety properties. As far as we can see, this design principle did not arise for termination so that the resulting approaches are maximal and largely incompatible with each other.

We then show that the principle of abstract interpretation actually well fits to termination analysis. The theoretical framework of abstract interpretation gives a diagnostic definition. Its abstraction respects a logical structure of the program and its refinement structure of local source terms. We also show the interest of abstract interpretation for termination analysis as well as new static analysis methods in sufficiently complete approximations of this new source variation.

In [2], we propose a generalization of the classical notion of temporal logic for linear-time temporal logic and model checking analysis based on the new semantics consisting of inductive linear time covering execution histories. It represents a new basis for terminating program analysis. In [3], we propose a new framework for termination analysis based on static and state analysis methods by abstraction on heap programs, memory constraint, and data. Examples of particular instances include Floyd's handling of loop nests as well as memory trees. Russell's annotations and static and continuous proof methods, and Podelski/Rybalchuk's transition iteration analysis.

Categories and Subject Descriptors: B.4.1 [Deductive Program Verification]; D.3.2 [Formal Methods and Theory]; D.3.3 [Specification and Synthesis and Reasoning about Programs].

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

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

1. Introduction

Floyd/Frigg's program proof methods for invariance and termination [24, 45, 39] have inspired most static analysis methods.

For static invariance analysis by abstract interpretation [28, 29], it is key step to explore the strongest iteration as a logical and mathematical object. This analysis is more interestingly related to abstract induction associated using the inductive/abstract Express approximation methods.

For static termination analysis, the discovery of variant invariants is often desirable in limited spaces [30] or also is based on the Floyd/Frigg's idea of variant iterations time until finished uses.

*Work supported in part by the CNRS/INRIA Project in Computer-verified Proofs.

**Research was made available in full copy of this work for inclusion in the INRIA report on a request submitted by the author. Requests for such copies should be addressed to the INRIA Project in Computer-verified Proofs, Institut Élie Cartan de Lorraine, Université de Lorraine, B.P. 70239, F-54500 Vandœuvre-lès-Nancy, France.

Copyright © 2012, ACM 978-1-4503-1249-3/12/06 \$15.00

achieved by observing quantities that strictly decrease within the while remaining lower-bounded, or stable. So most termination analysis methods indirectly reduce to a relational invariance analysis of the program.

The abstract interpretation design principle is instantiated with abstract characters for safety and termination analysis, proof and checking/repair methods under partial termination or the termination for nondeterministic systems.

The first nice idea for termination is that there exists a positive source function that can be expressed in Floyd's framework. The second nice idea is that there exists a negative source function that can be expressed in Floyd's framework. The third nice idea is that there exists a negative source function that can be expressed in the abstract interpretation design principle of relational static invariants. This yields static analysis methods automatically inferring abstractions of source functions by the continuous Floyd approximation and of source interpretation.

The second basic idea introduced in this paper both for safety and termination analysis is that one can prove termination properties over time invariant covers and their abstract. These requires are more powerful than linear relations between states which have been used traditionally in program verification: provide the example, the iteration intervals used in [20] are iterative abstraction of the set of user-writes sequences. Programs can be analyzed by abstract interpretation, the proof methods can be applied to the Floyd [20] and Podelski/Rybalchuk [30] methods. The proof of the termination relation closure by well-founded relations, Podelski/Rybalchuk [30], are combinations and generalizations.

2. Theopins, theopint induction, abstraction, and approximation

We express estimates on degrees of maps: $f: A \rightarrow B$ if A, B are sets and $x \in A$ such that $x = f(y)$. We let B^A be the class of maps $f: A \rightarrow B$ on the power $|A| \cdot |B|$ product that is equal to \emptyset or $\{x\}$.

The dual notion is that of prepoint (maps) $f: B \rightarrow A$, where $y \in B$ is the image of x , and $f^{-1}(y)$ is the pointed under class from the source. By Tarski/Paluszak's Logical Extension [26], $f: A \rightarrow B$ is a prepoint if and only if $\forall x \in A \exists y \in B \forall z \in f^{-1}(y) \forall w \in f^{-1}(z) \forall u \in f^{-1}(w) \forall v \in f^{-1}(u) \forall t \in f^{-1}(v) \forall s \in f^{-1}(t) \forall r \in f^{-1}(s) \forall p \in f^{-1}(r) \forall q \in f^{-1}(p) \forall n \in f^{-1}(q) \forall m \in f^{-1}(n) \forall l \in f^{-1}(m) \forall k \in f^{-1}(l) \forall j \in f^{-1}(k) \forall i \in f^{-1}(j) \forall h \in f^{-1}(i) \forall g \in f^{-1}(h) \forall d \in f^{-1}(g) \forall c \in f^{-1}(d) \forall b \in f^{-1}(c) \forall a \in f^{-1}(b) \forall r \in f^{-1}(a) \forall s \in f^{-1}(r) \forall t \in f^{-1}(s) \forall u \in f^{-1}(t) \forall v \in f^{-1}(u) \forall w \in f^{-1}(v) \forall x \in f^{-1}(w) \forall y \in f^{-1}(x) \forall z \in f^{-1}(y) \forall w' \in f^{-1}(z) \forall v' \in f^{-1}(w') \forall u' \in f^{-1}(v') \forall r' \in f^{-1}(u') \forall s' \in f^{-1}(r') \forall t' \in f^{-1}(s') \forall u'' \in f^{-1}(t') \forall v'' \in f^{-1}(u'') \forall w'' \in f^{-1}(v'') \forall x' \in f^{-1}(w'') \forall y' \in f^{-1}(x') \forall z' \in f^{-1}(y') \forall w''' \in f^{-1}(z') \forall v''' \in f^{-1}(w''') \forall u''' \in f^{-1}(v''') \forall r''' \in f^{-1}(u''') \forall s''' \in f^{-1}(r''') \forall t''' \in f^{-1}(s''') \forall u'''' \in f^{-1}(t''') \forall v'''' \in f^{-1}(u''') \forall w'''' \in f^{-1}(v''') \forall x'''' \in f^{-1}(w''') \forall y'''' \in f^{-1}(x''') \forall z'''' \in f^{-1}(y''') \forall w''''' \in f^{-1}(z''') \forall v''''' \in f^{-1}(w''') \forall u''''' \in f^{-1}(v''') \forall r''''' \in f^{-1}(u''') \forall s''''' \in f^{-1}(r''') \forall t''''' \in f^{-1}(s''') \forall u'''''' \in f^{-1}(t''') \forall v'''''' \in f^{-1}(u''') \forall w'''''' \in f^{-1}(v''') \forall x'''''' \in f^{-1}(w''') \forall y'''''' \in f^{-1}(x''') \forall z'''''' \in f^{-1}(y''') \forall w''''''' \in f^{-1}(z''') \forall v''''''' \in f^{-1}(w''') \forall u''''''' \in f^{-1}(v''') \forall r''''''' \in f^{-1}(u''') \forall s''''''' \in f^{-1}(r''') \forall t''''''' \in f^{-1}(s''') \forall u''''''' \in f^{-1}(t''') \forall v''''''' \in f^{-1}(u''') \forall w''''''' \in f^{-1}(v''') \forall x''''''' \in f^{-1}(w''') \forall y''''''' \in f^{-1}(x''') \forall z''''''' \in f^{-1}(y''') \forall w'''''''' \in f^{-1}(z''') \forall v'''''''' \in f^{-1}(w''') \forall u'''''''' \in f^{-1}(v''') \forall r'''''''' \in f^{-1}(u''') \forall s'''''''' \in f^{-1}(r''') \forall t'''''''' \in f^{-1}(s''') \forall u''''''''' \in f^{-1}(t''') \forall v''''''''' \in f^{-1}(u''') \forall w''''''''' \in f^{-1}(v''') \forall x''''''''' \in f^{-1}(w''') \forall y''''''''' \in f^{-1}(x''') \forall z''''''''' \in f^{-1}(y''') \forall w''''''''' \in f^{-1}(z''') \forall v''''''''' \in f^{-1}(w''') \forall u''''''''' \in f^{-1}(v''') \forall r''''''''' \in f^{-1}(u''') \forall s''''''''' \in f^{-1}(r''') \forall t''''''''' \in f^{-1}(s''') \forall u'''''''''' \in f^{-1}(t''') \forall v'''''''''' \in f^{-1}(u''') \forall w'''''''''' \in f^{-1}(v''') \forall x'''''''''' \in f^{-1}(w''') \forall y'''''''''' \in f^{-1}(x''') \forall z'''''''''' \in f^{-1}(y''') \forall w'''''''''''' \in f^{-1}(z''') \forall v'''''''''''' \in f^{-1}(w''') \forall u'''''''''''' \in f^{-1}(v''') \forall r'''''''''''' \in f^{-1}(u''') \forall s'''''''''''' \in f^{-1}(r''') \forall t'''''''''''' \in f^{-1}(s''') \forall u''''''''''''' \in f^{-1}(t''') \forall v''''''''''''' \in f^{-1}(u''') \forall w''''''''''''' \in f^{-1}(v''') \forall x''''''''''''' \in f^{-1}(w''') \forall y''''''''''''' \in f^{-1}(x''') \forall z''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u''''''''''''''''''''''''' \in f^{-1}(v''') \forall r''''''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''''''''' \in f^{-1}(s''') \forall u''''''''''''''''''''''''' \in f^{-1}(t''') \forall v''''''''''''''''''''''''' \in f^{-1}(u''') \forall w''''''''''''''''''''''''' \in f^{-1}(v''') \forall x''''''''''''''''''''''''' \in f^{-1}(w''') \forall y''''''''''''''''''''''''' \in f^{-1}(x''') \forall z'''''''''''''''''''''' \in f^{-1}(y''') \forall w''''''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''' \in f^{-1}(z''') \forall v''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s'''''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''' \in f^{-1}(r''') \forall t''''''''''''''''''' \in f^{-1}(s''') \forall u'''''''''''''''''''''' \in f^{-1}(t''') \forall v'''''''''''''''''''''' \in f^{-1}(u''') \forall w'''''''''''''''''''''' \in f^{-1}(v''') \forall x'''''''''''''''''''''' \in f^{-1}(w''') \forall y'''''''''''''''''''''' \in f^{-1}(x''') \forall z''''''''''''''''''' \in f^{-1}(y''') \forall w'''''''''''''''''''''' \in f^{-1}(z''') \forall v'''''''''''''''''''''' \in f^{-1}(w''') \forall u'''''''''''''''''''''' \in f^{-1}(v''') \forall r'''''''''''''''''''''' \in f^{-1}(u''') \forall s''''''''''''''''''' \in f^{-1}(r''') \forall t'''''''''''''''' \in f^{-1}(s''') \$



- **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

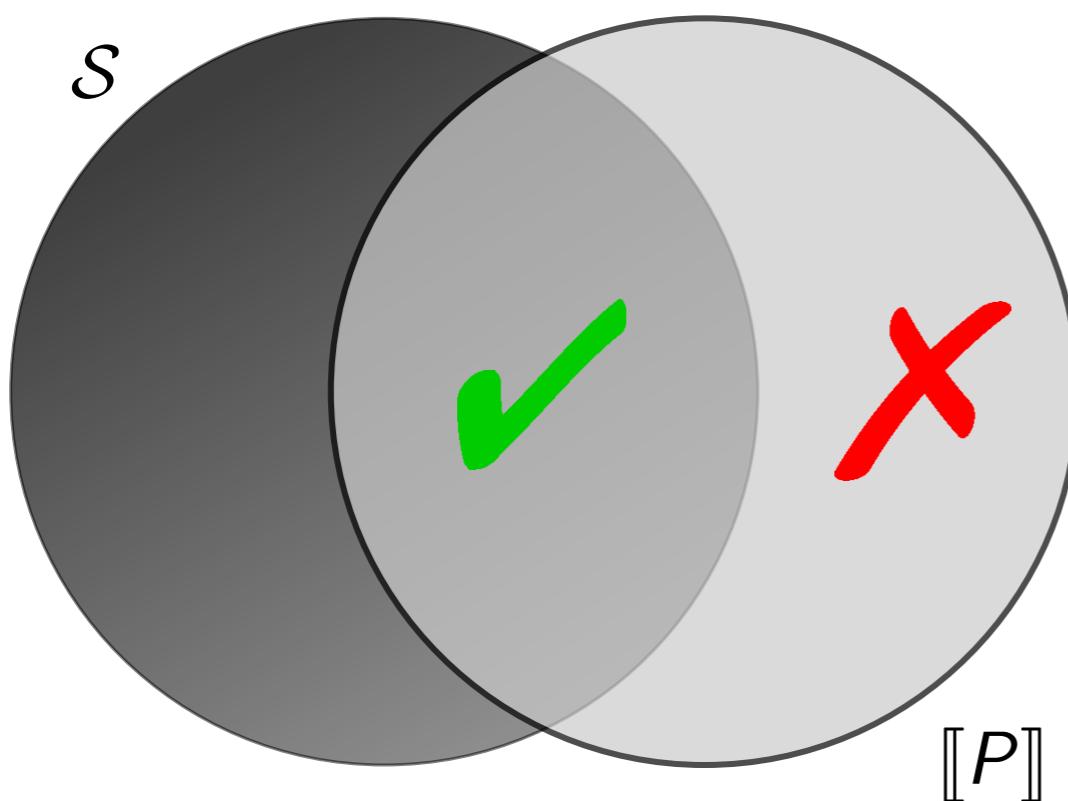


Urban - *The Abstract Domain of Segmented Ranking Functions* (SAS 2013)

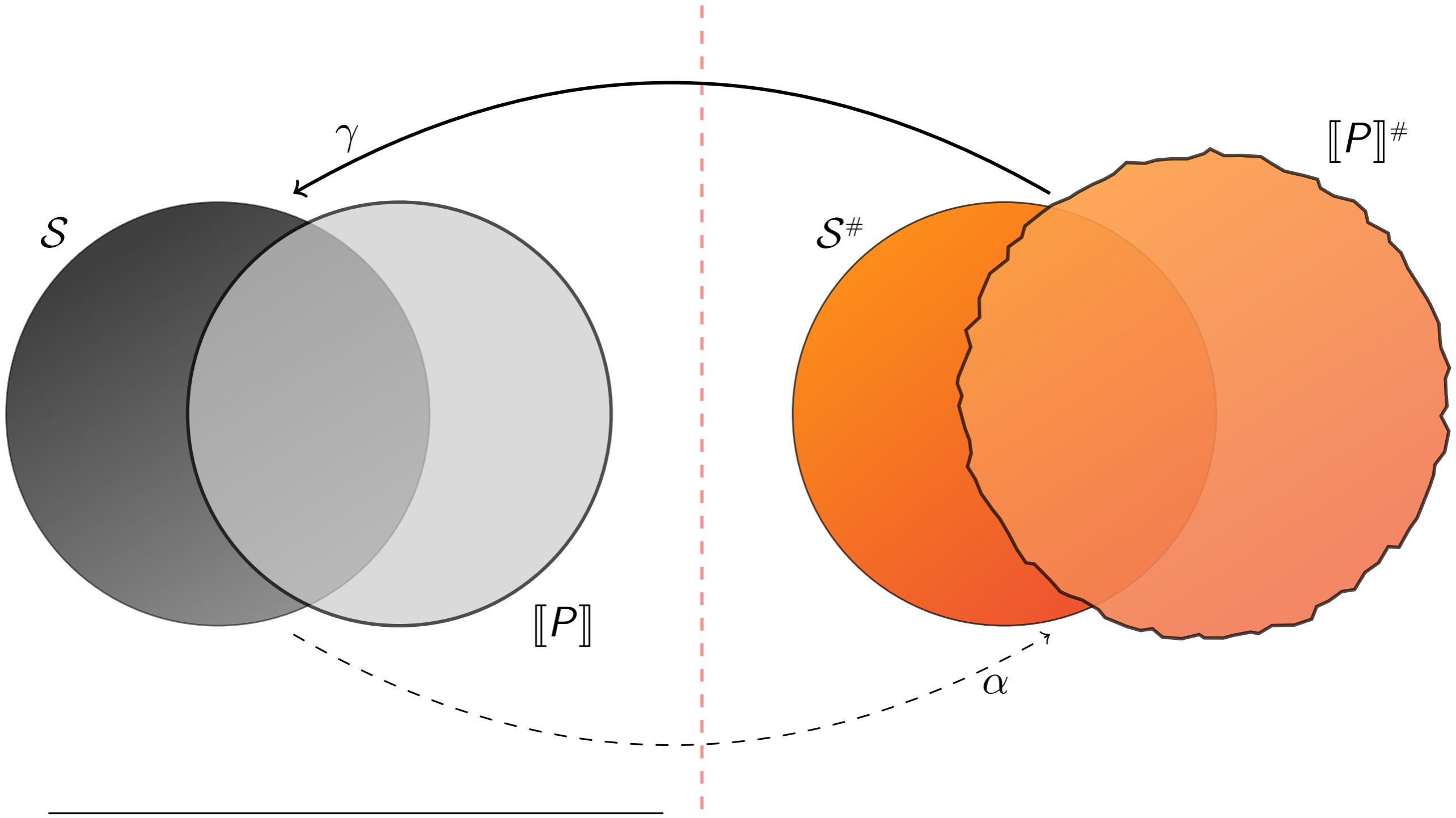
Urban&Miné - An Abstract Domain to Infer Ordinal-Valued Ranking Functions (ESOP 2014)

Urban&Miné - A Decision Tree Abstract Domain for Proving Conditional Termination (SAS'2014)

Abstract Interpretation

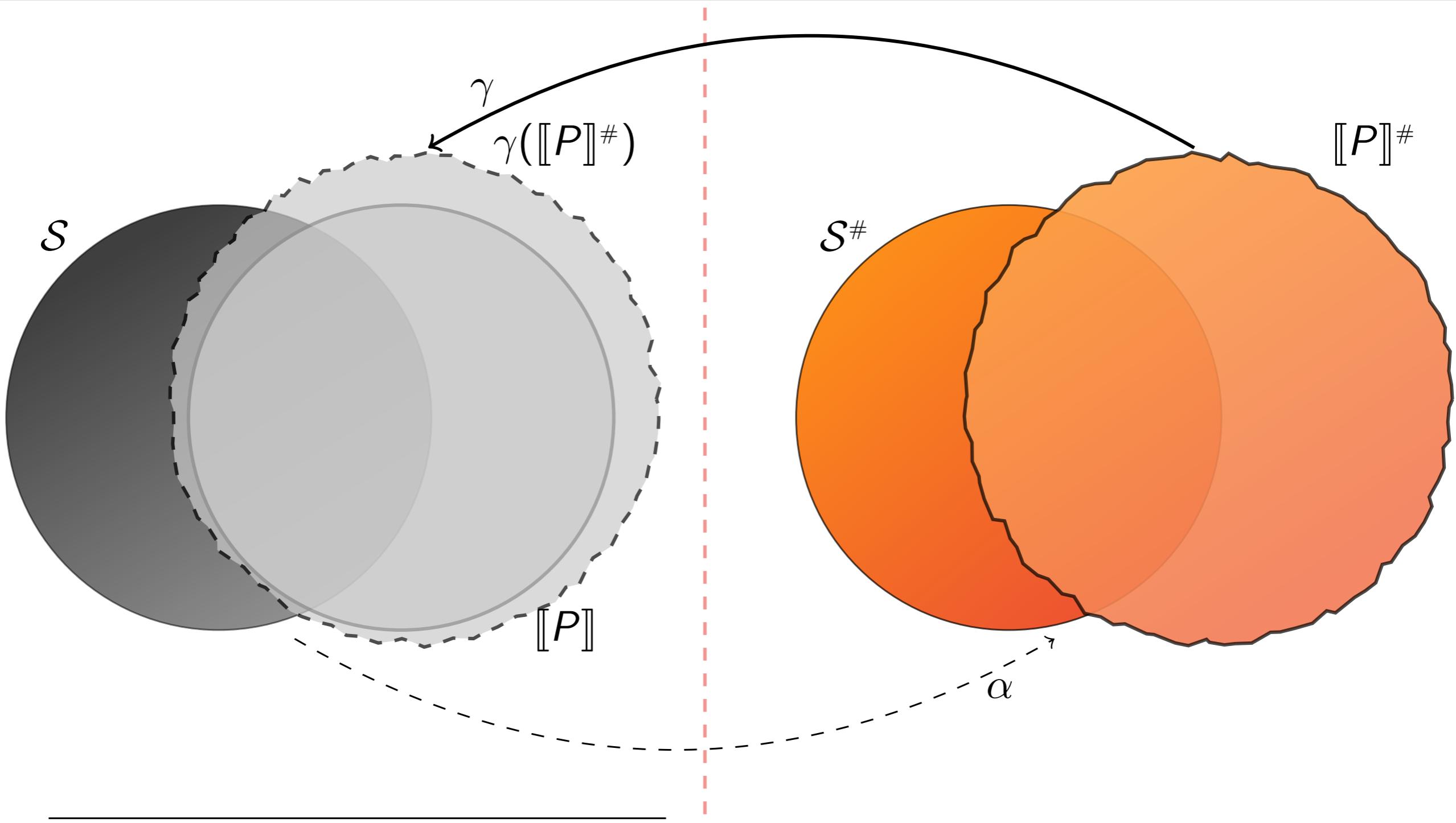


Abstract Interpretation

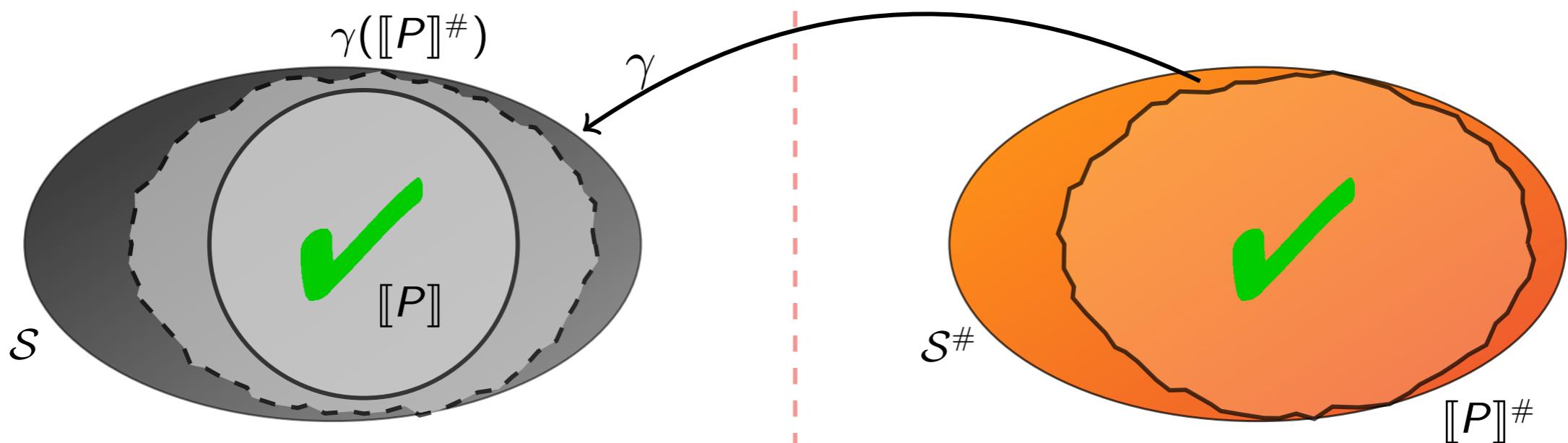


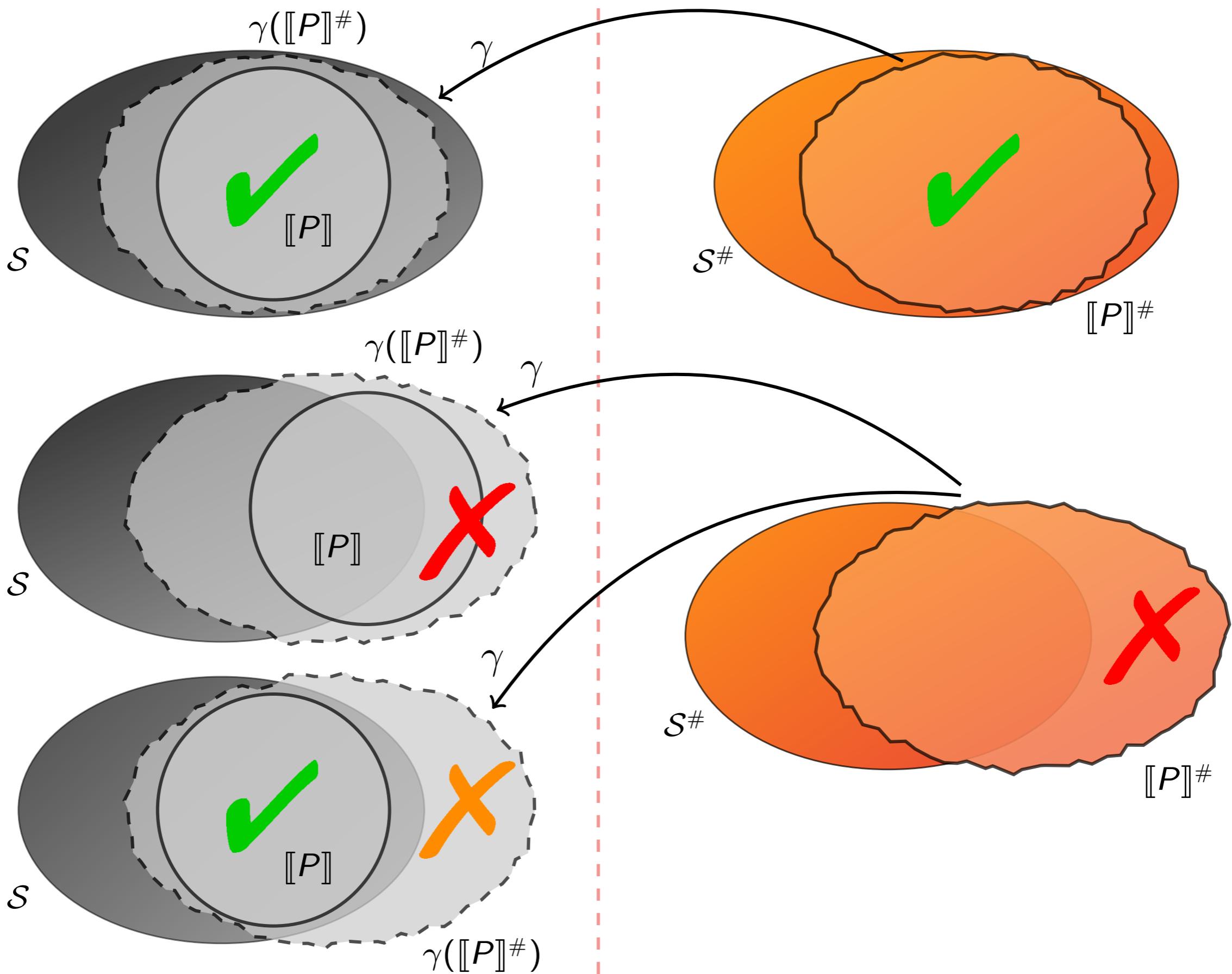
Cousot&Cousot - *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints* (POPL 1977)

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, École Normale Supérieure, and INRIA, Paris
Computer Science, 75231 Paris Cedex 05, France
cousot@cs.enst.fr, cousin@inria.fr

Réaline Cousot

CNRS, École Normale Supérieure, and INRIA, Paris
cousot@cs.enst.fr

Abstract:

exact, universal and analysis methods for termination of rely on two main kinds of well-founded and well-ordering methods: (1) methods for safety; (2) methods for termination.

The abstract interpretation design principle is first illustrated. As the theory of well-founded and well-ordering methods, verification and analysis methods for termination are based on the same basic principles: (1) safety properties of programs are first expressed in a constructive fragment of logic, and proved by abstract interpretation methods that automatically prove them; (2) termination properties of programs are first expressed in a non-constructive fragment of logic, and proved by abstract interpretation methods that automatically prove them.

For (1), we show that the safety principle applies readily well to pointer and object annotations. The tree-based termination analysis methods for safety are based on the notion of trace abstraction, a generalization of the least common ancestor. By further abstraction of a tree least common ancestor, we derive the Floyd-like termination proof method as well as new static analysis methods to efficiently compute approximations of termination proofs.

For (2), we introduce a generalization of the weakest notion of strict induction (as found in Moore logic) into a semantic structural induction based on the few semantic concepts of inductive trace (over-control conditions) and trace abstraction. The tree-based termination analysis methods for termination are based on the notion of trace abstraction, which generalizes its interaction with the generation invariant proof, verification and static analysis methods by iteration on both program structure, control and data flow. A particular instance of Moore's Floyd's theorem of least common ancestor, namely Termination, Termination's approximation and termination proof method, and Peleg's Krylakowski's termination iteration.

Categories and Subject Descriptors: D.3.4 [Software/Program Verification]; D.3.3 [Formal Definitions and Theory]; D.3.2 [Design and Depiling and Recovery from Programs]; G.3 [Mathematics of Computing]: Languages, Security, Theory, Verification.

Keywords: Abstract Interpretation, Induction, Proof, Safety, Static Analysis, Virtue Assertion, Verification, Termination.

1. Introduction

Most existing program proof methods for verification and termination [24, 25] are based on termination analysis.

The most common analysis by abstract interpretation [24, 25], a key step is to express the strongest invariant as a logical form and how to approximate this strongest invariant to automatically infer an abstract inductive overapproximation using the constructive fragment approximation methods.

This work is concerned with the discovery of various functions that are definable in bounded arithmetics [26] or other theories based on the Floyd/Turing idea of various functions from well-founded sets.

*Work supported in part by the CNRS/INP/INRIA/ENS/Computer science joint laboratory.

Revisions to this digital or hard copy of all or part of this work will be accepted or classroom use is granted without prior permission or fee. Other rights are reserved or transferred according to the copyright notices contained on the pages within this document.

© 2012 ACM 0001-0782/12/0100-0001-0001 \$15.00

obtained by observing quantities that satisfy discrete while loops while they are implemented, or similarly for exact termination analysis methods indirectly reduce to a relational invariant analysis hence can reuse classical safety analysis methods.

The abstract interpretation design principle is instantiated with suitable abstractions, safety and termination analysis, proof, and checking/validation rather potential termination or definite termination analysis.

The first main idea is that there exists a most precise invariant function that can be expressed in discrete form by abstract interpretation of a termination collecting semantics itself abstracting the program operational trace semantics. This yields safety analysis methods automatically inferring abstractions of that invariant function by the constructive fragment approximation method of abstract interpretation.

The second main idea introduced in this paper both for analysis and termination is that of semantic structural induction, including termination proofs, over trace segment covers and their abstractions. Trace segments are more powerful than binary relations between states which have been used traditionally in program termination analysis, for example, in the analysis of loops and termination analysis of the set of trace segments. Examples include effective induction on the program entries (catching loop invariants) & in Floyd [20] induction on data, & in Russell [3], the occurring of the transition relation closest to well-founded relations, & in Peleg's Rydelekafka [30], their combinations and generalizations.

2. Floyd-like, Reproductive Induction, abstraction, and approximation

We express induction in discrete form of maps $f: A \rightarrow B$ i.e. $\text{GCD}(f, A) = 1$ such that $x = f(y)$. We let f^k be the k-th iterate of f & $A \sqsubset B$ the power (A, B) greater than or equal to $n \times A$, if $(A, B) \sqsubset (C, D)$ means that $(A, C) \sqsubset (B, D)$ i.e. $x \in A \sqsubset B$ iff $x \in C \sqsubset D$ if C is a subsegment of A , and by f the f -th iterate of A is in B from the context. By David Peleg's Floyd-Essays [20, 29], $f = T(P \times A) \times_{\leq} P \times P$ values for f increasing on a complete lattice $(A, \sqsubset, \sqsubseteq, \sqcap, \sqcup, \sqsupset)$ in its image $(B, \sqsubset, \sqsubseteq, \sqcap, \sqcup, \sqsupset)$. The discrete invariants are $f^{\text{min}}(A) = \text{min}_{x \in A} f(x) = f(\text{min}(A))$, $f^{\text{max}}(A) = \text{max}_{x \in A} f(x) = f(\text{max}(A))$, where f is a pre-discrete and f a continuous [27]. If f is increasing but not continuous, transfinite iterations may have to be used [28].

$f: A \rightarrow B$ is increasing (also monotone increasing...) if on a power (A, \sqsubset) if and only if $(x, y) \in A \times A$ with $x \sqsubset y$ then $f(x) \sqsubset f(y)$ [29].

A is complete lattice if $(A, \sqsubset, \sqsubseteq, \sqcap, \sqcup, \sqsupset)$ is a power (A, \sqsubset) such that every nonempty subset has a least upper bound (lub) [30]. Hence a greatest lower bound (glb) is a $x \in A$ with $y \sqsupset x$ for all $y \in A$.

A is a complete partial order (cpo) if $(A, \sqsubset, \sqsubseteq, \sqcap, \sqcup, \sqsupset)$ is a power (A, \sqsubset) such that if all increasing chains $C \in \wp(A)$ such that in $\text{inf}(C)$ does exist if the lub $\text{lub}(C)$ exists and in each the $\text{lub}(C) \sqsubseteq C$.

$f: X \rightarrow Y$ is the preimage of Y as the set of all values of a set Y .

The preimage (x) image of $X \in \wp(X)$ by a map $f: X \rightarrow Y$ is $f^{-1}(X) = \{y \in Y \mid f^{-1}(y) \in X\}$.



Transition systems

Transition systems: definition

Language-neutral formalism to discuss about program semantics.

Transition system: (Σ, τ)

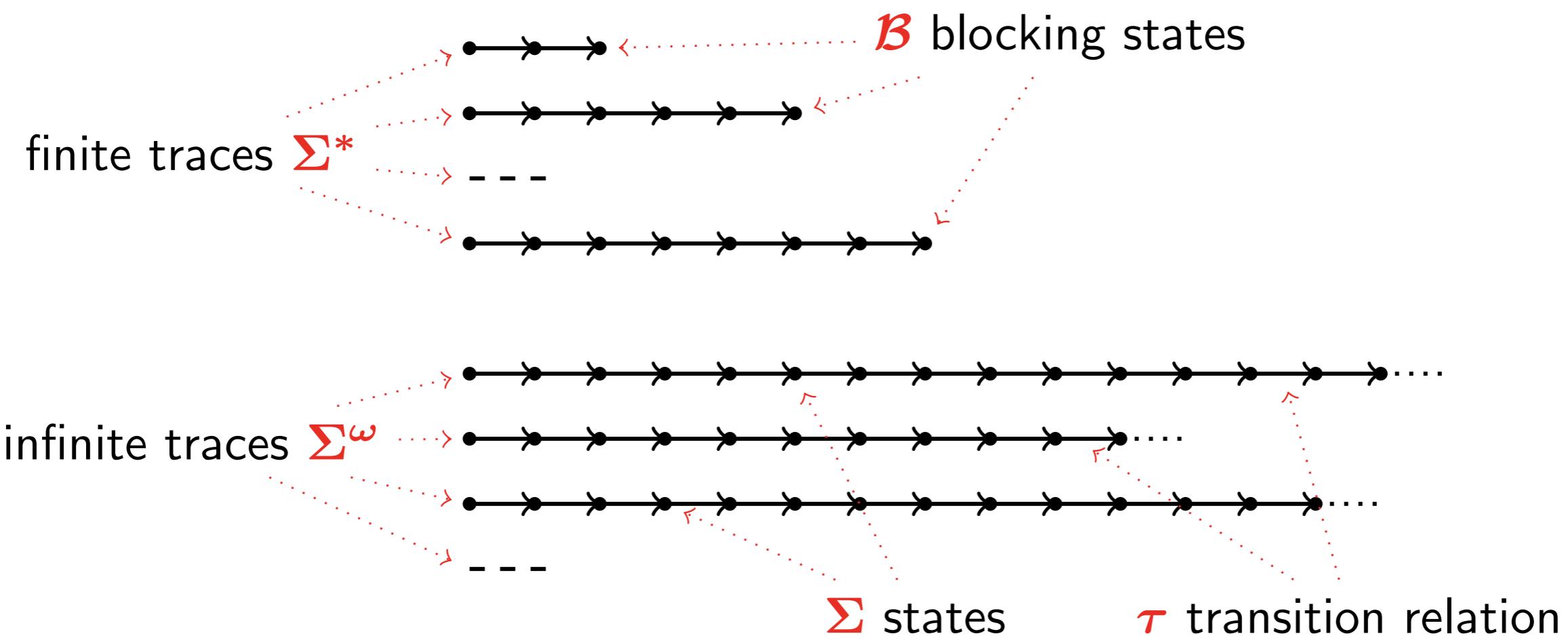
- set of states Σ ,
(memory states, λ -terms, configurations, etc., generally infinite)
- transition relation $\tau \subseteq \Sigma \times \Sigma$.

(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 $(\Sigma^\omega, \supseteq, \cap, \cup, \Sigma^\omega, \emptyset)$.

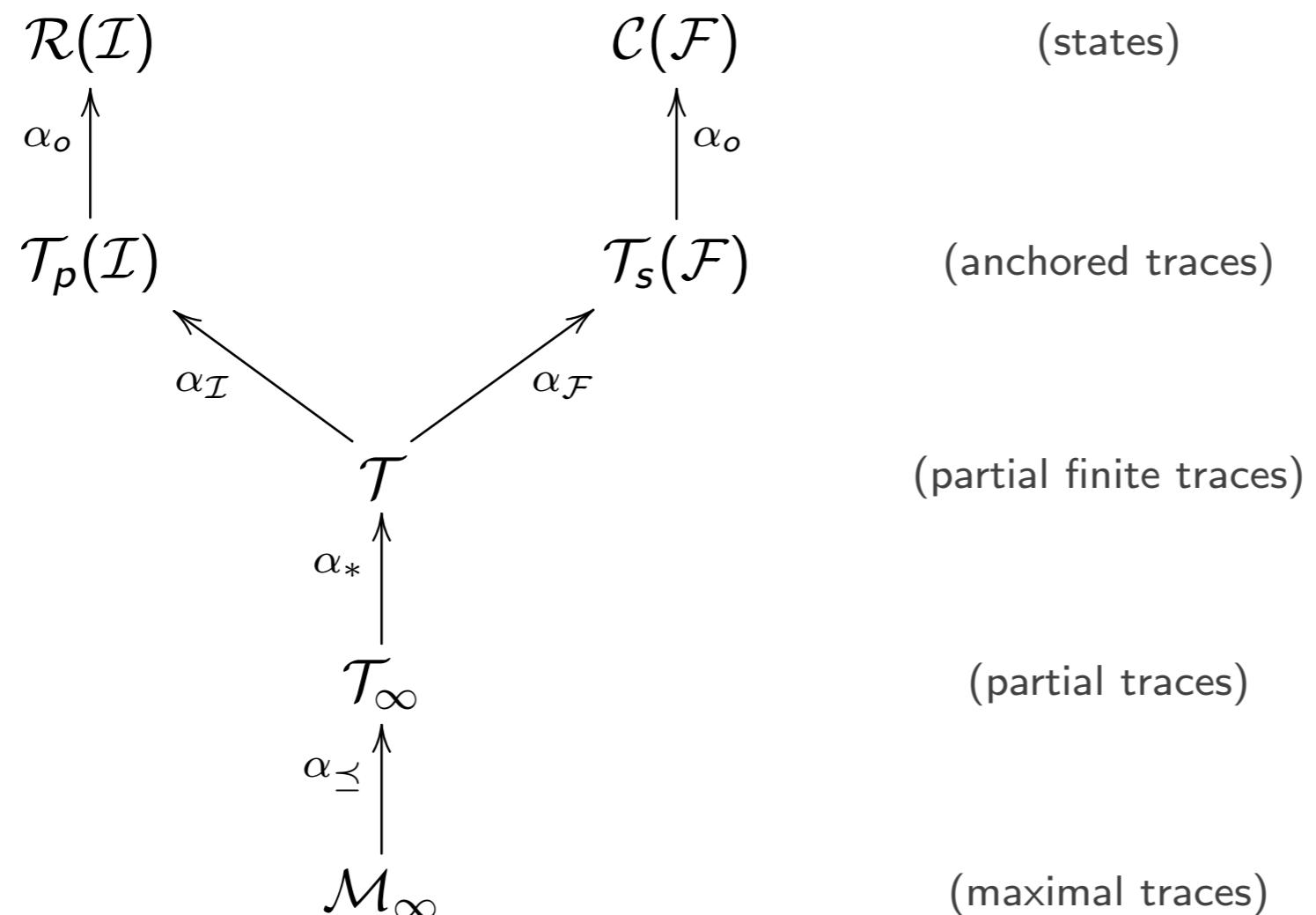
We mix them into a **new** complete lattice $(\Sigma^\infty, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$:

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

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

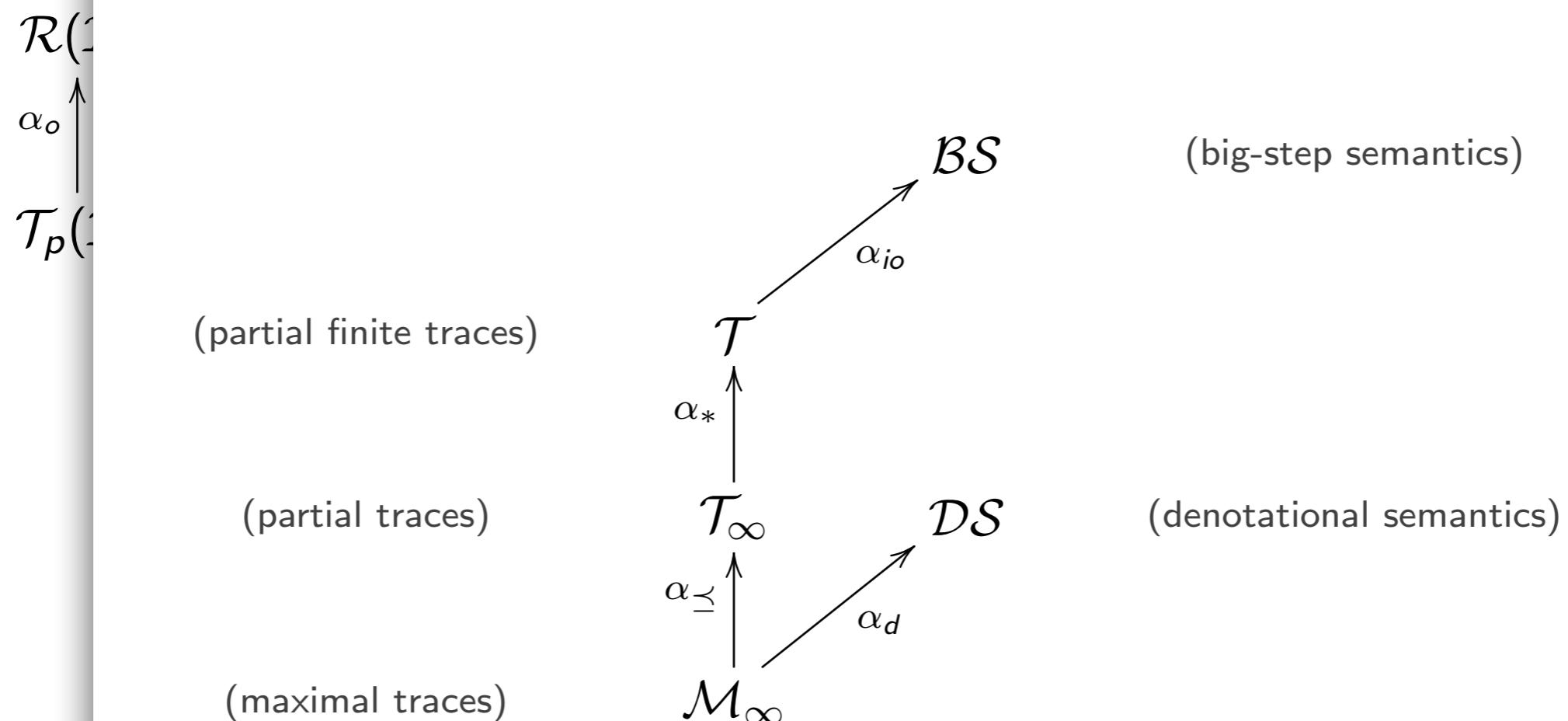


(Partial) hierarchy of semantics

Relational semantics

Denotational semantics

Another part of the hierarchy of semantics



See [Cou82] for more semantics in this diagram.

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

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

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

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

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

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

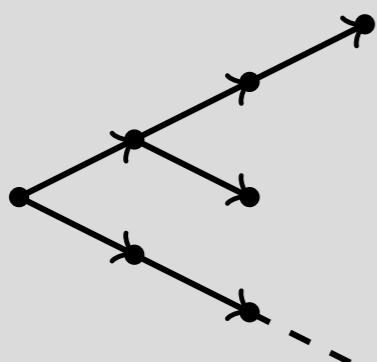
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

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

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**

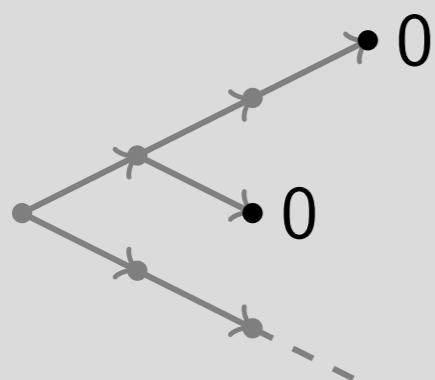
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

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

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**

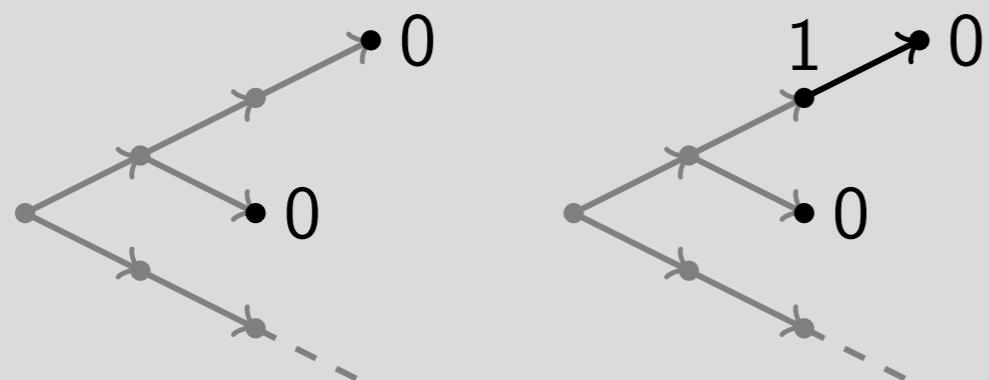
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

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

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**

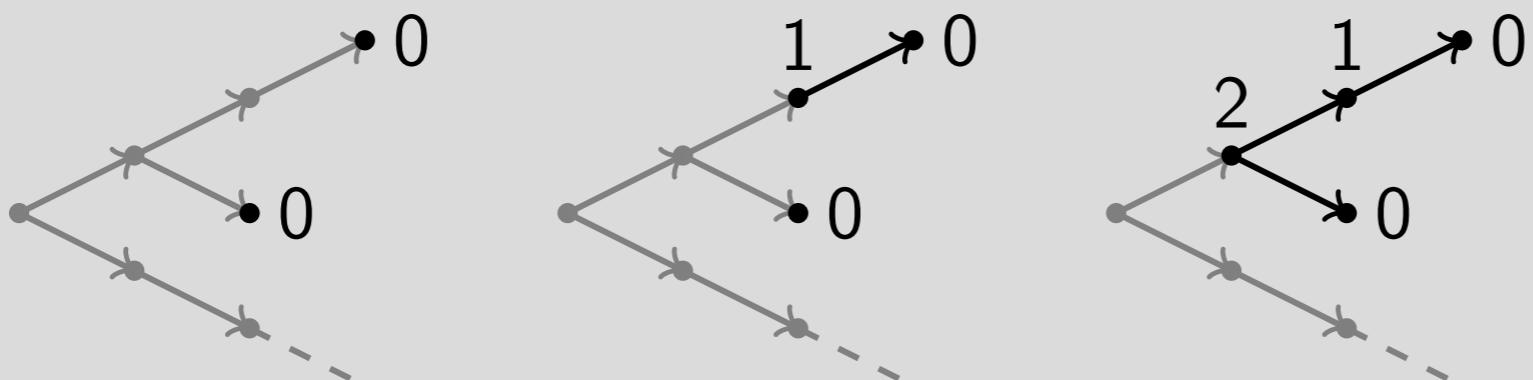
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

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

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**

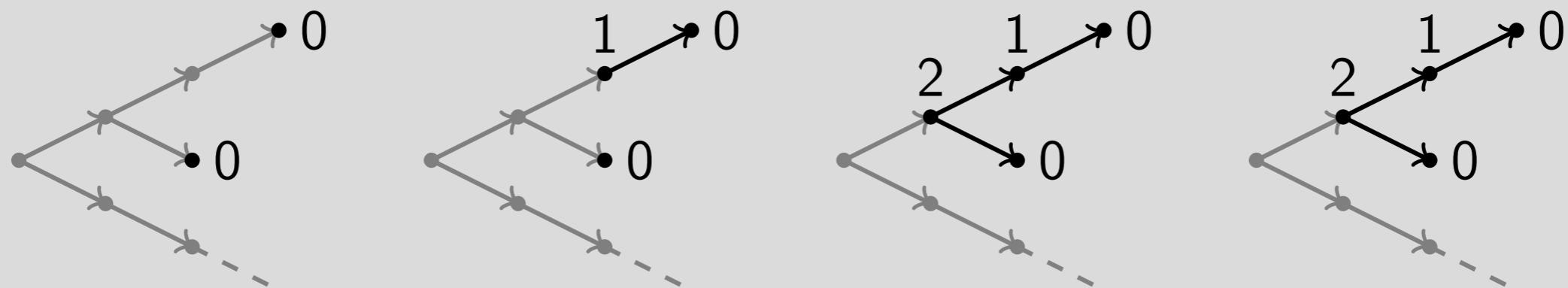
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

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

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** :

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

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

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

Definition (Computational Order)

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

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

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

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

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

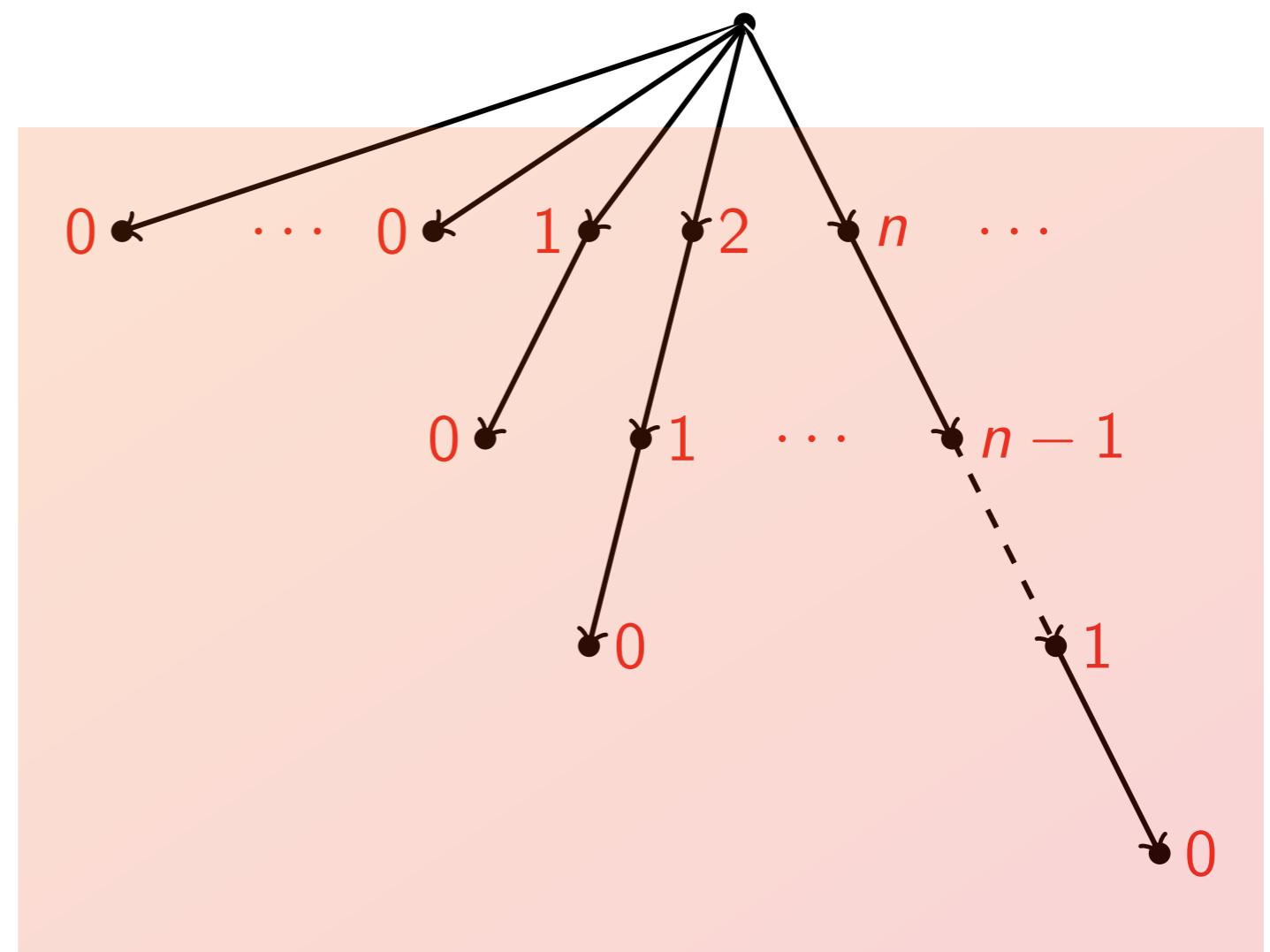
Theorem (Soundness and Completeness)

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

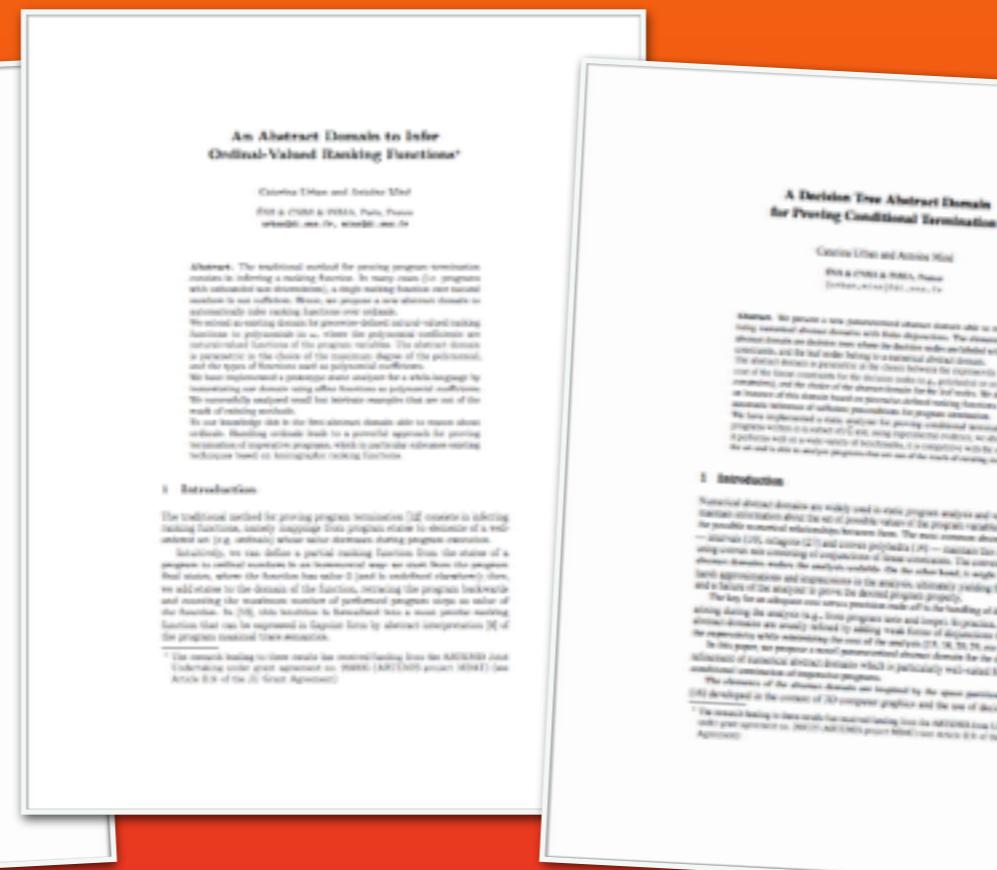
- **remark:** the termination semantics is **not computable!**

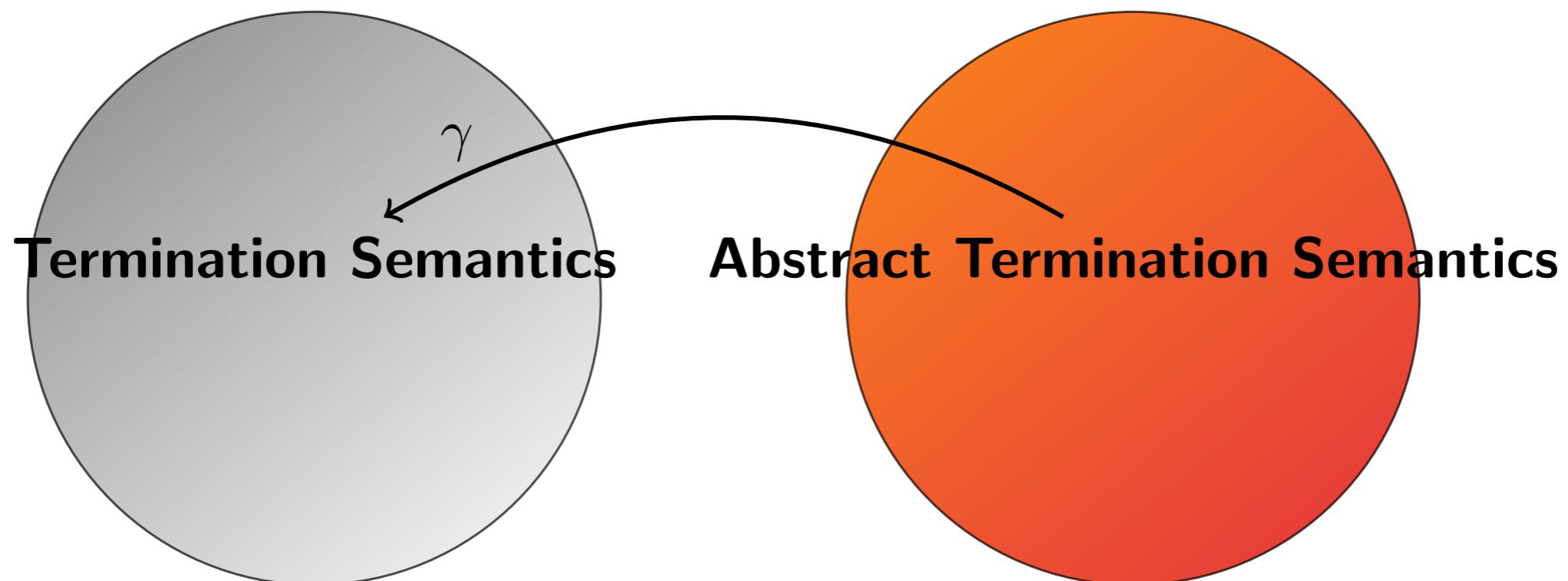
Example

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



Piecewise-Defined Ranking Functions





Fixpoint approximations

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 $c \leq \gamma(a)$.

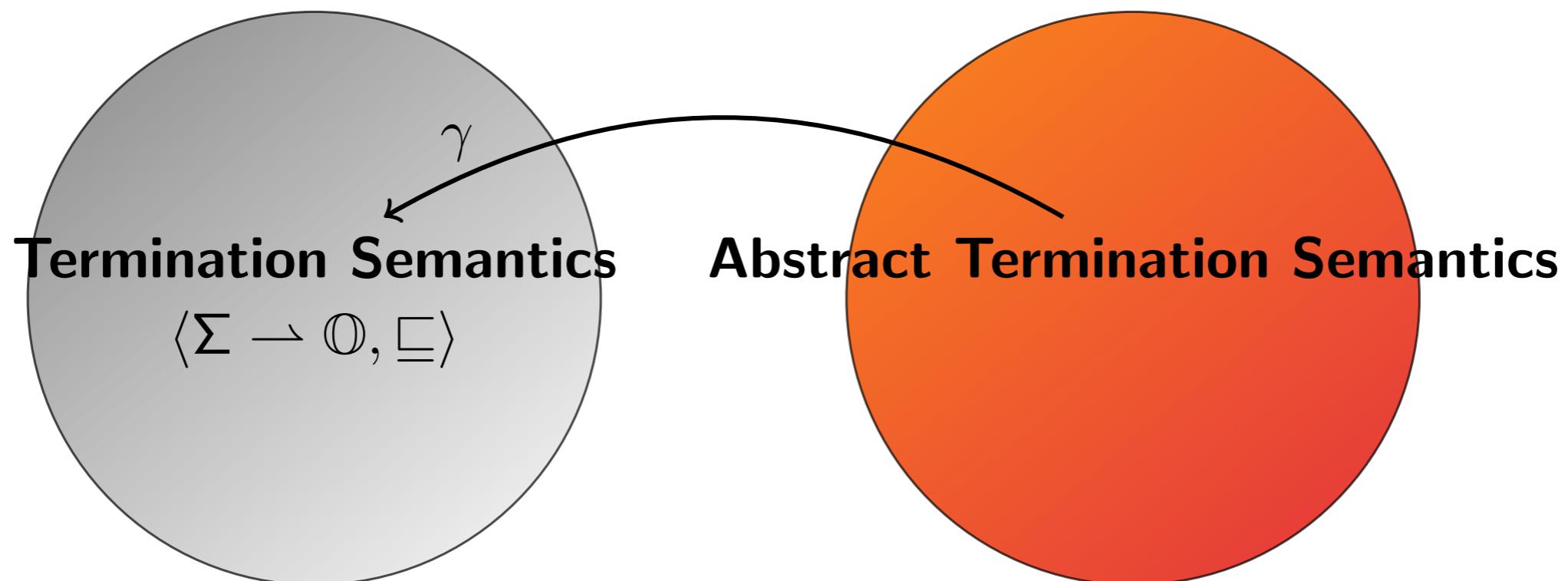
(e.g.: $[0, 10]$ is a sound abstraction of $\{0, 1, 2, 5\}$ in the integer interval domain)

- $g : A \rightarrow A$ 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)

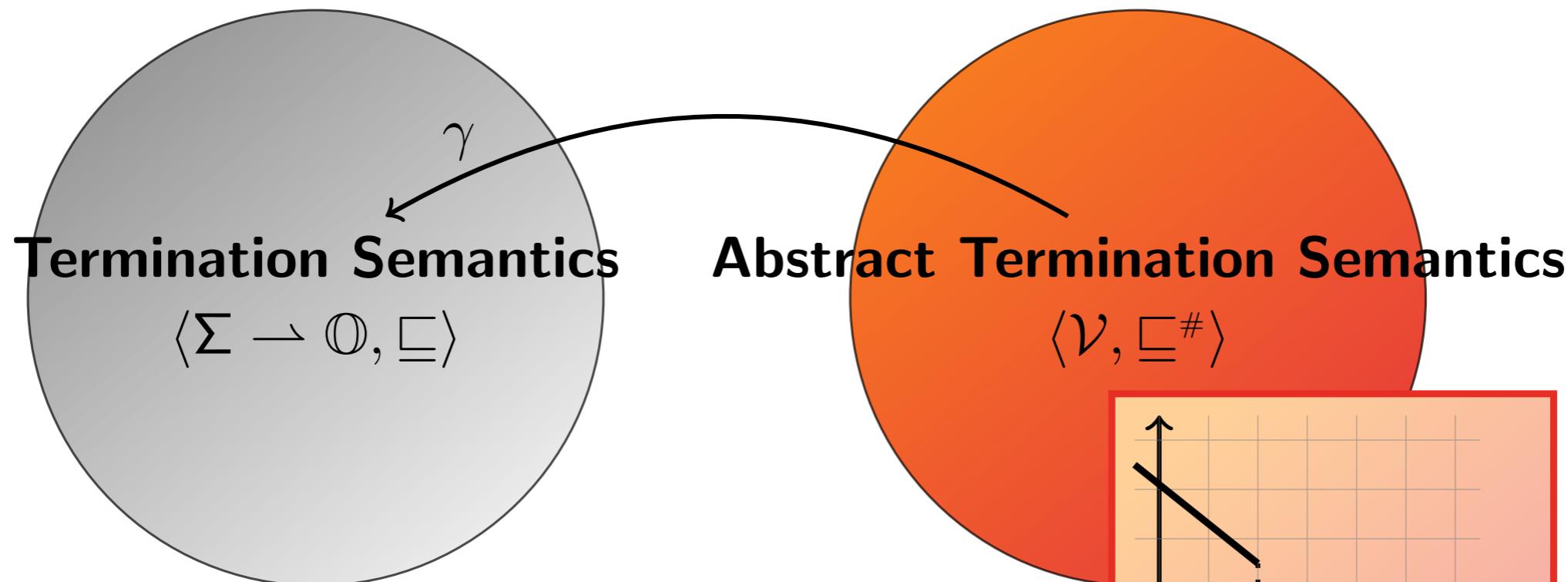
$$v_1 \sqsubseteq v_2 \stackrel{\text{def}}{=} \text{dom}(v_1) \supseteq \text{dom}(v_2) \wedge \forall x \in \text{dom}(v_1) : v_1(x) \leq v_2(x)$$

Partial orders

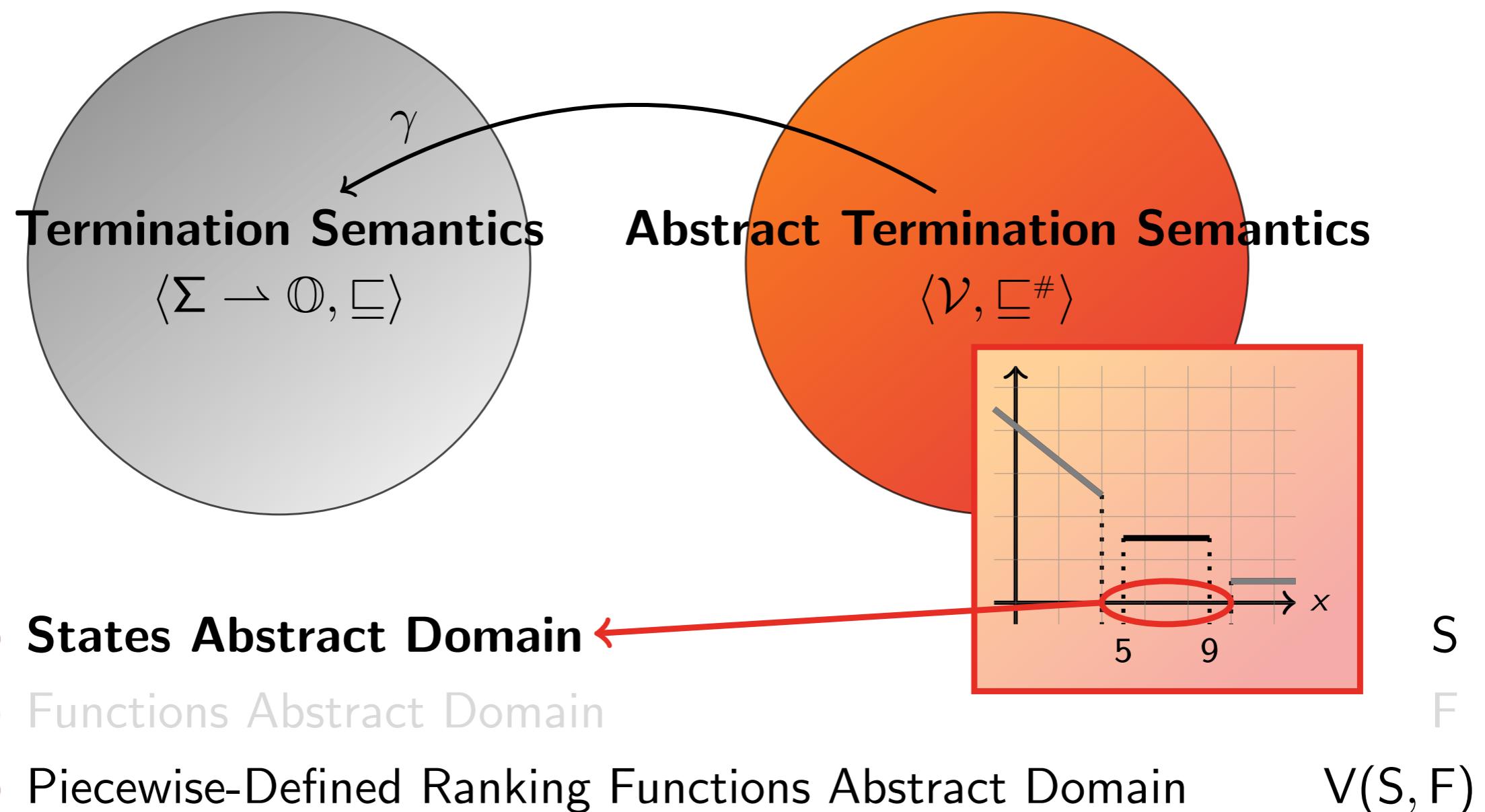
Use of posets (informally)

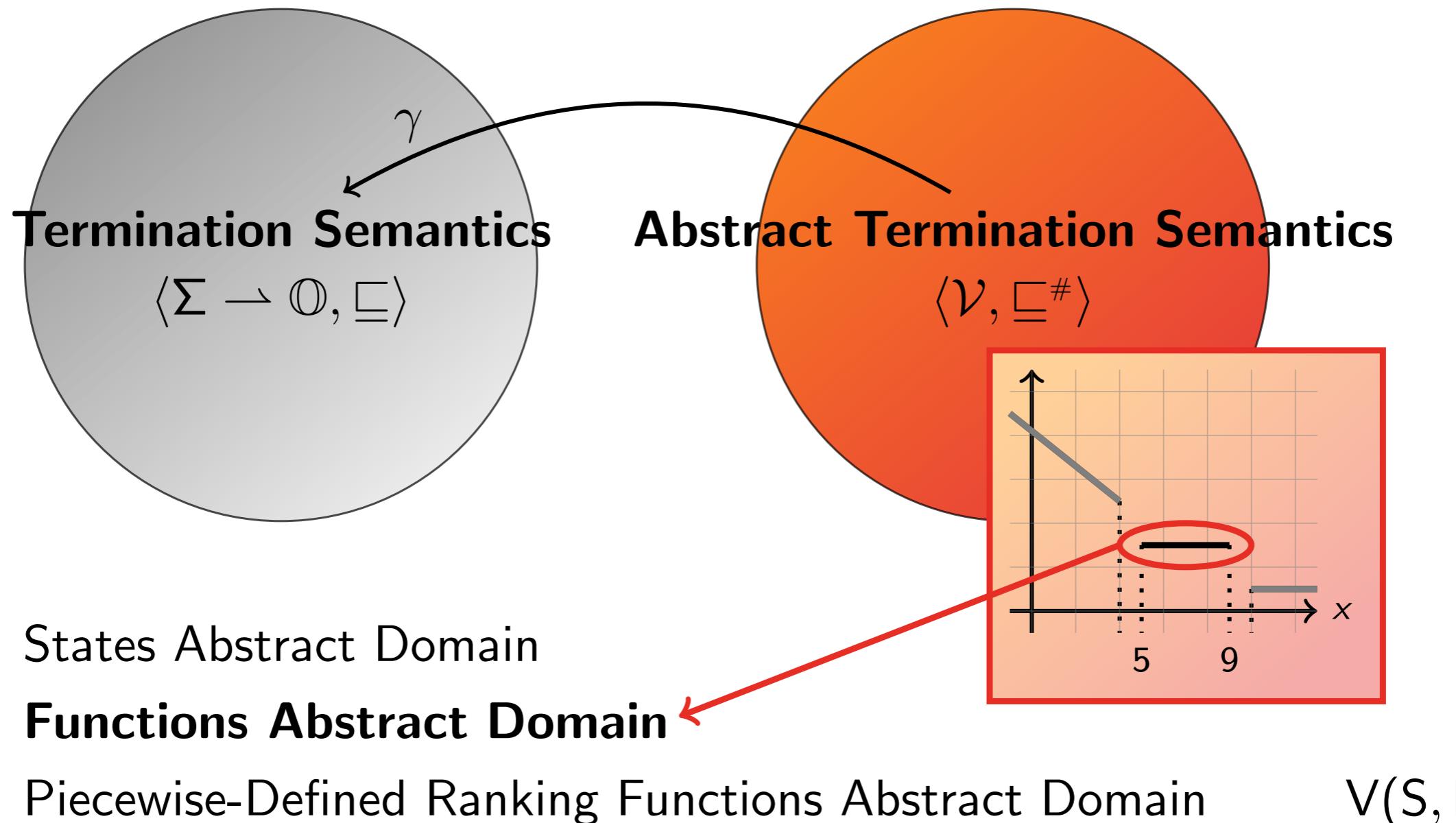
Posets are a very useful notion to discuss about:

- 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 \sqsubseteq accepted behaviors)
- iteration: fixpoint computation
(e.g., a computation is directed, with a limit: $X_1 \sqsubseteq X_2 \sqsubseteq \dots \sqsubseteq X_n$)



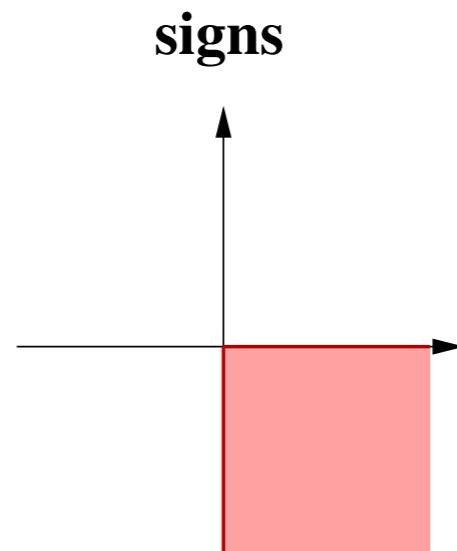
- States Abstract Domain
- Functions Abstract Domain
- **Piecewise-Defined Ranking Functions Abstract Domain $V(S, F)$**



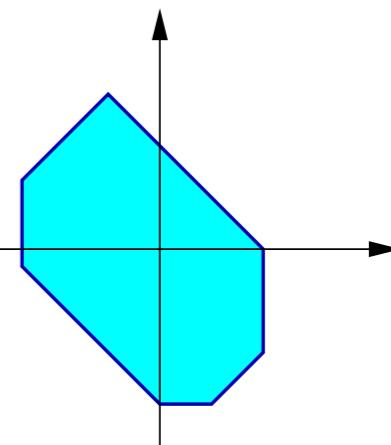
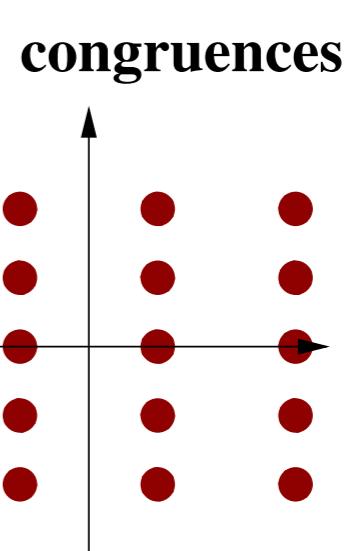
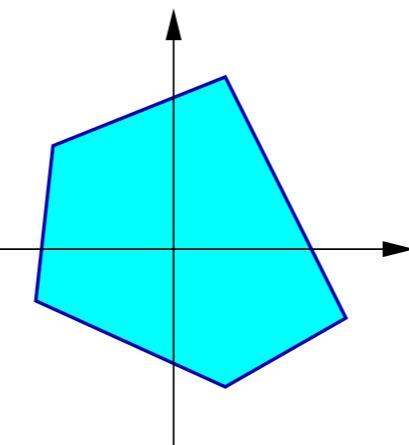
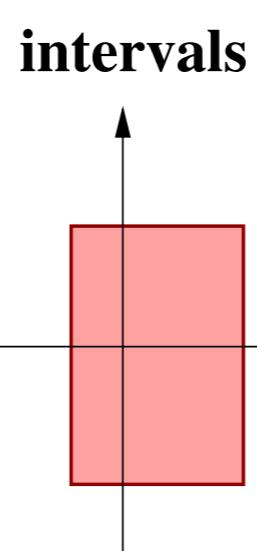
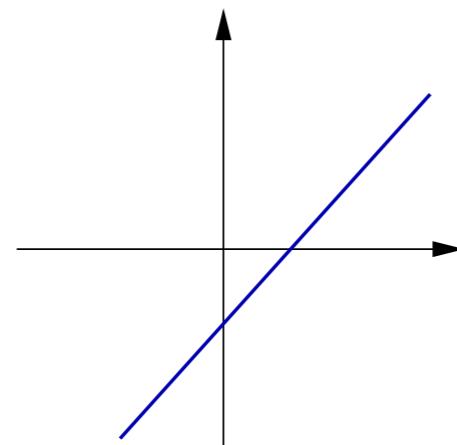


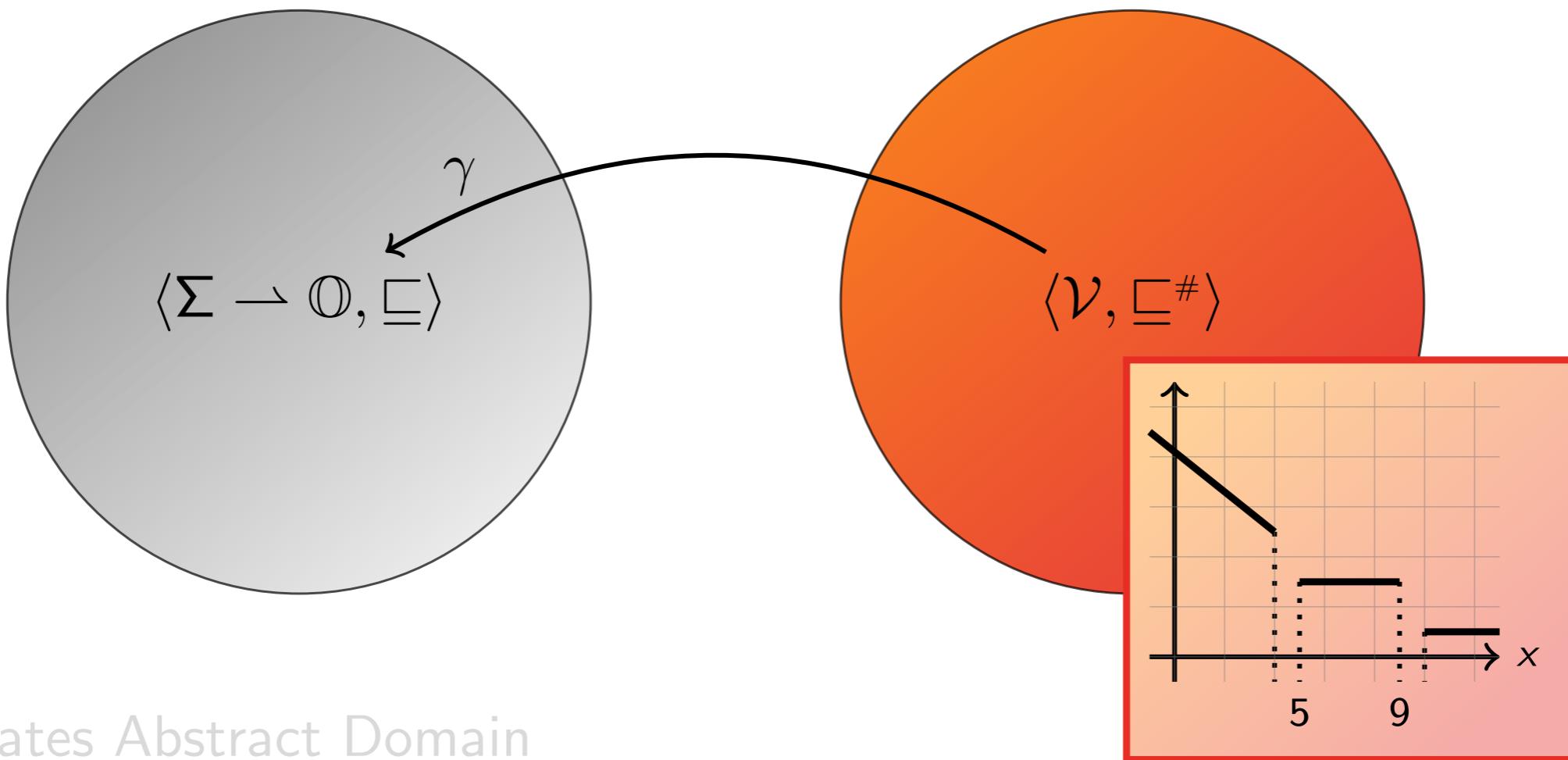
Numerical abstract domain examples

**non
relational**

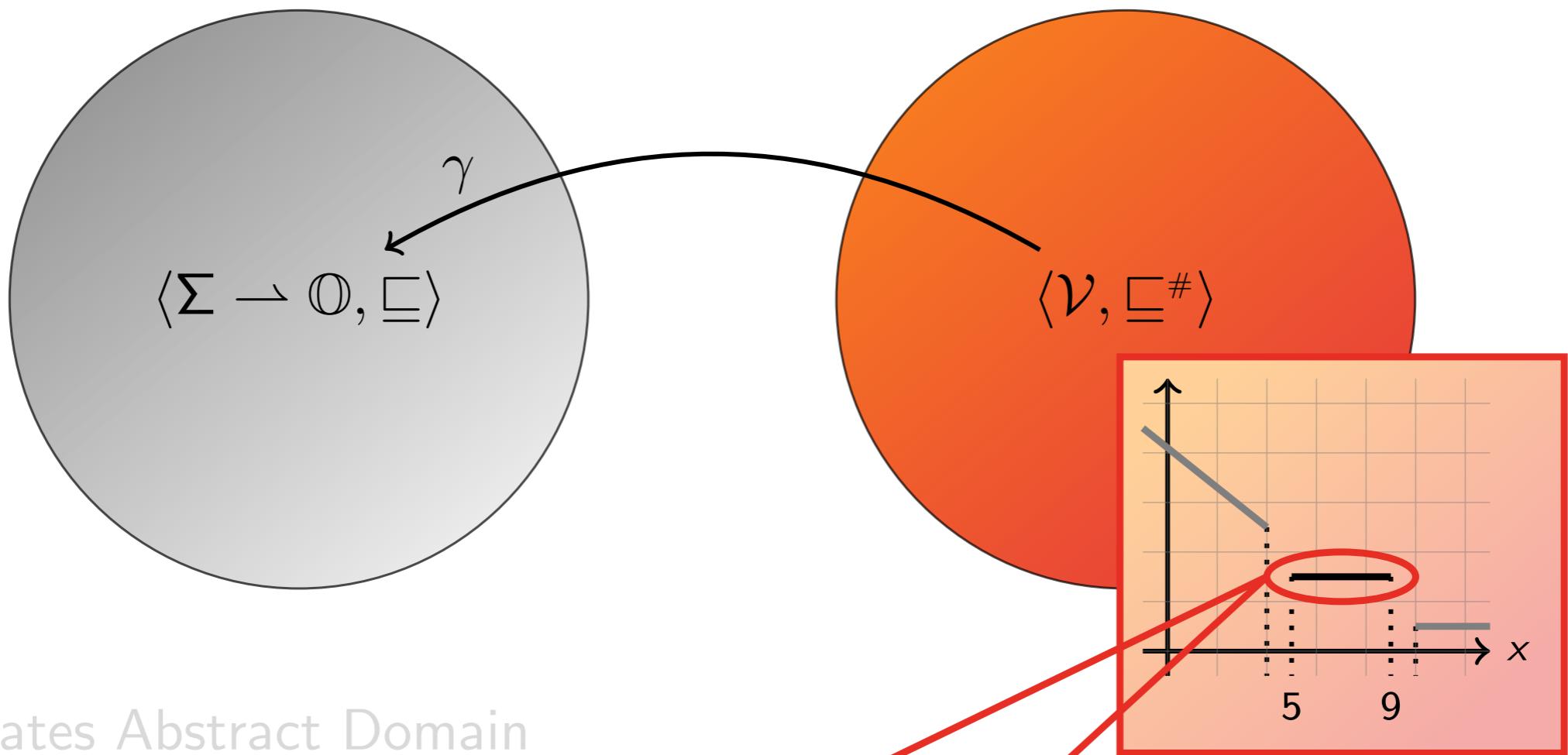


relational

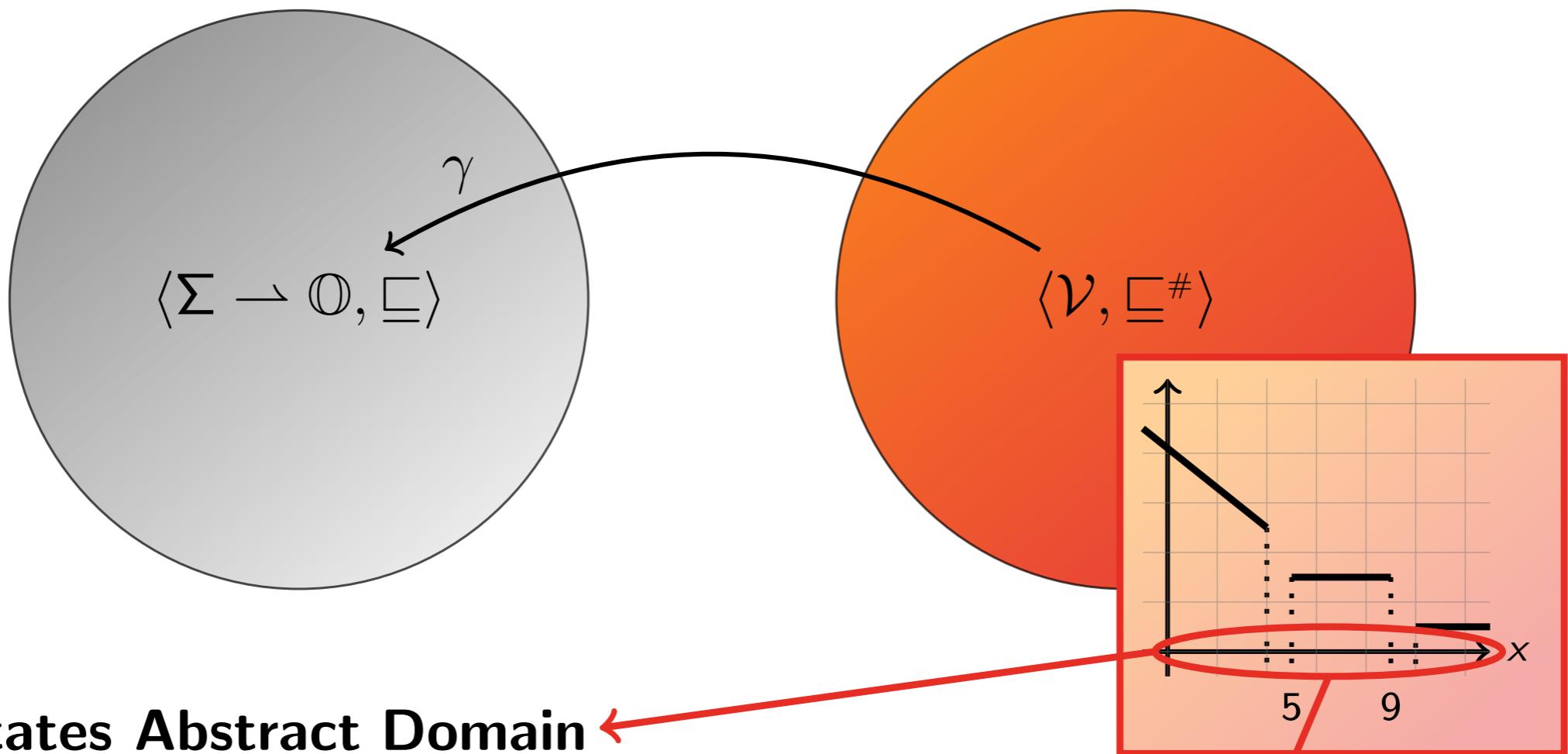




- States Abstract Domain
 - $\mathcal{L} \stackrel{\text{def}}{=} \text{Interval/Octagonal/Polyhedral Linear Constraints}$
- Functions Abstract Domain
 - $\mathcal{F} \stackrel{\text{def}}{=} \{\perp\} \cup \{f \mid f \in \mathbb{Z}^n \rightarrow \mathbb{N}\} \cup \{\top\}$
where $f \equiv f(x_1, \dots,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



- **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**
 - $\mathcal{V} \stackrel{\text{def}}{=} \{\text{LEAF} : f \mid f \in \mathcal{F}\} \cup \{\text{NODE}\{c\} : t_1, t_2 \mid c \in \mathcal{L} \wedge t_1, t_2 \in \mathcal{V}\}$

Example

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

Example

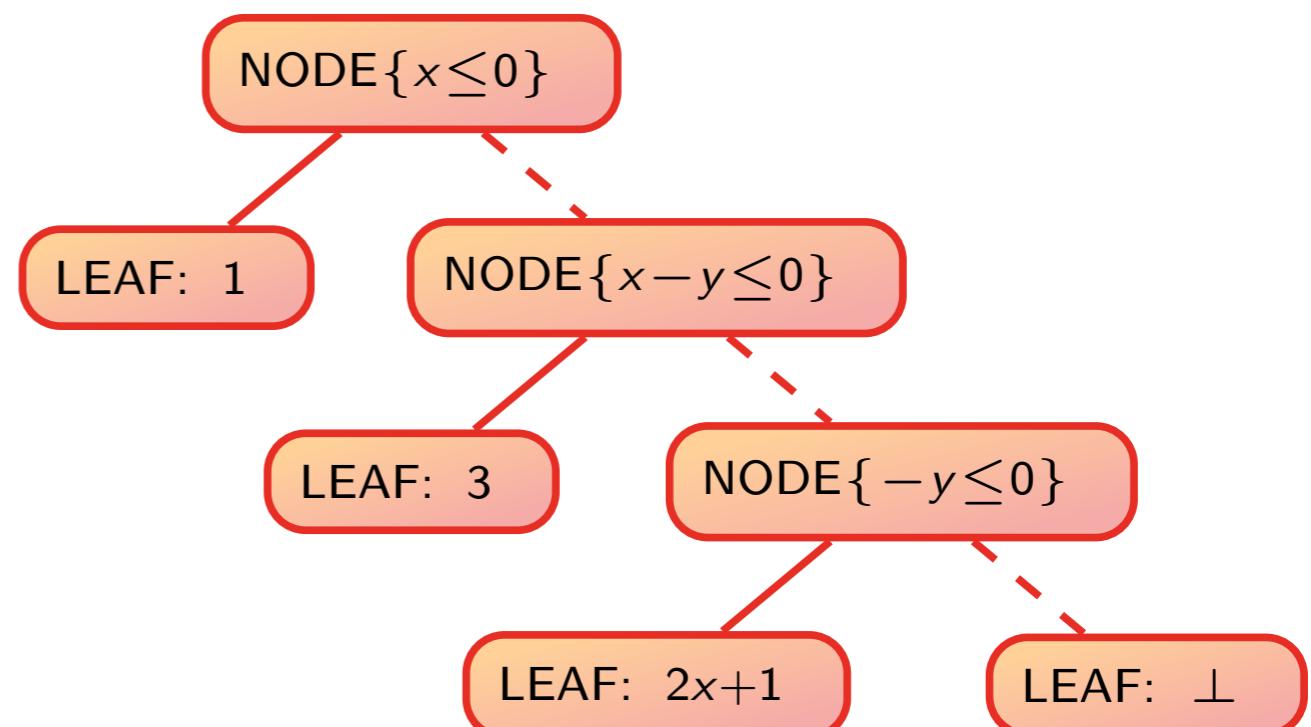
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

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
```

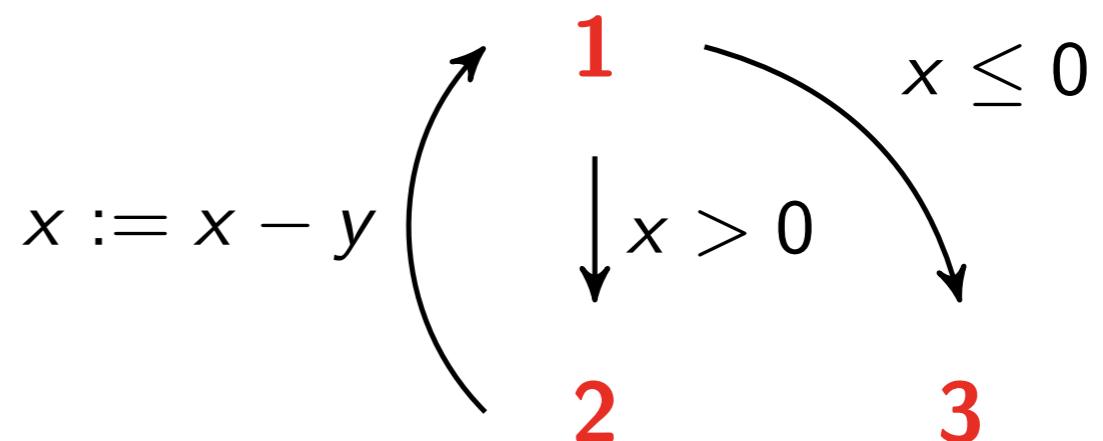
the program terminates if
and only if $x \leq 0 \vee y > 0$



Example

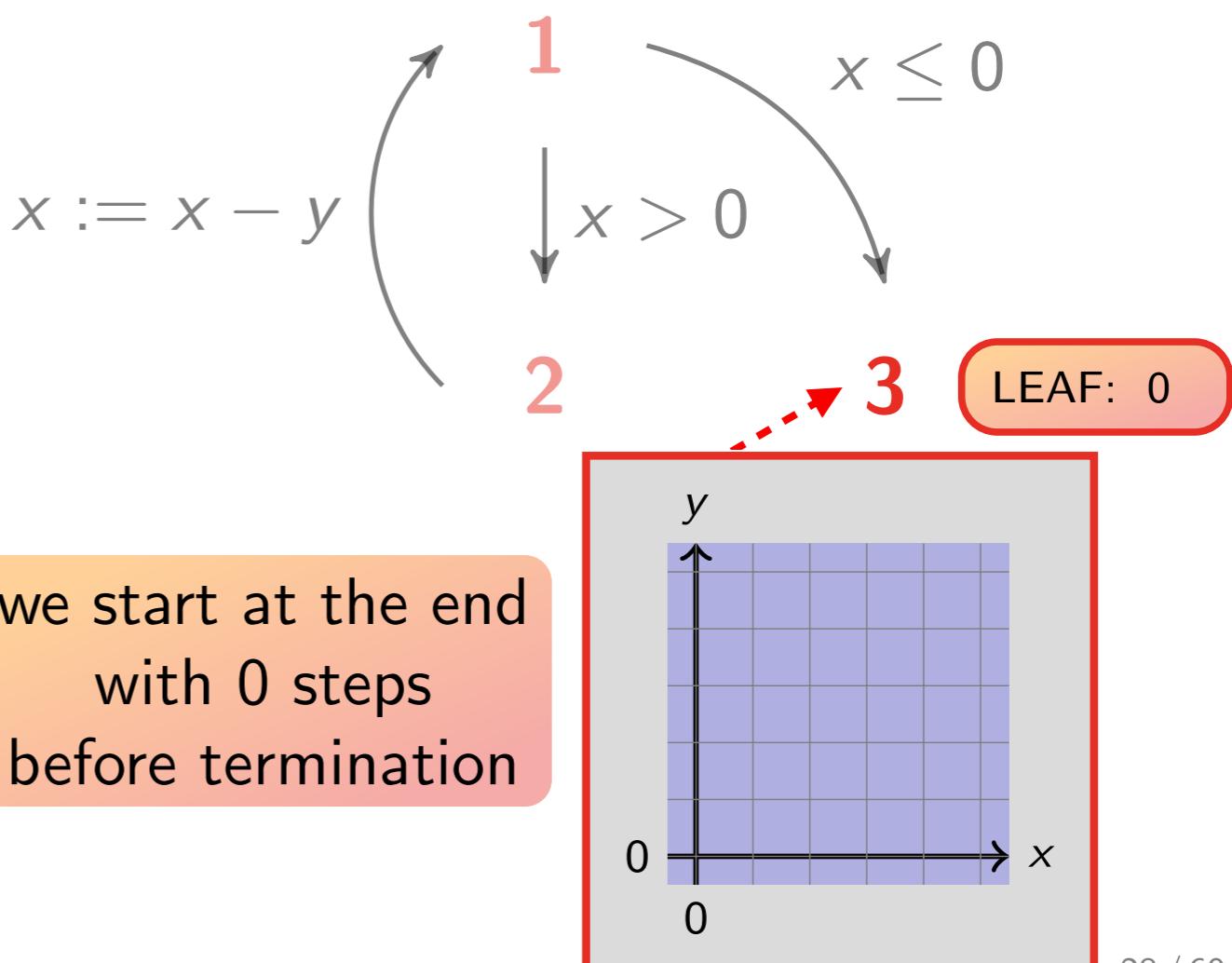
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

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



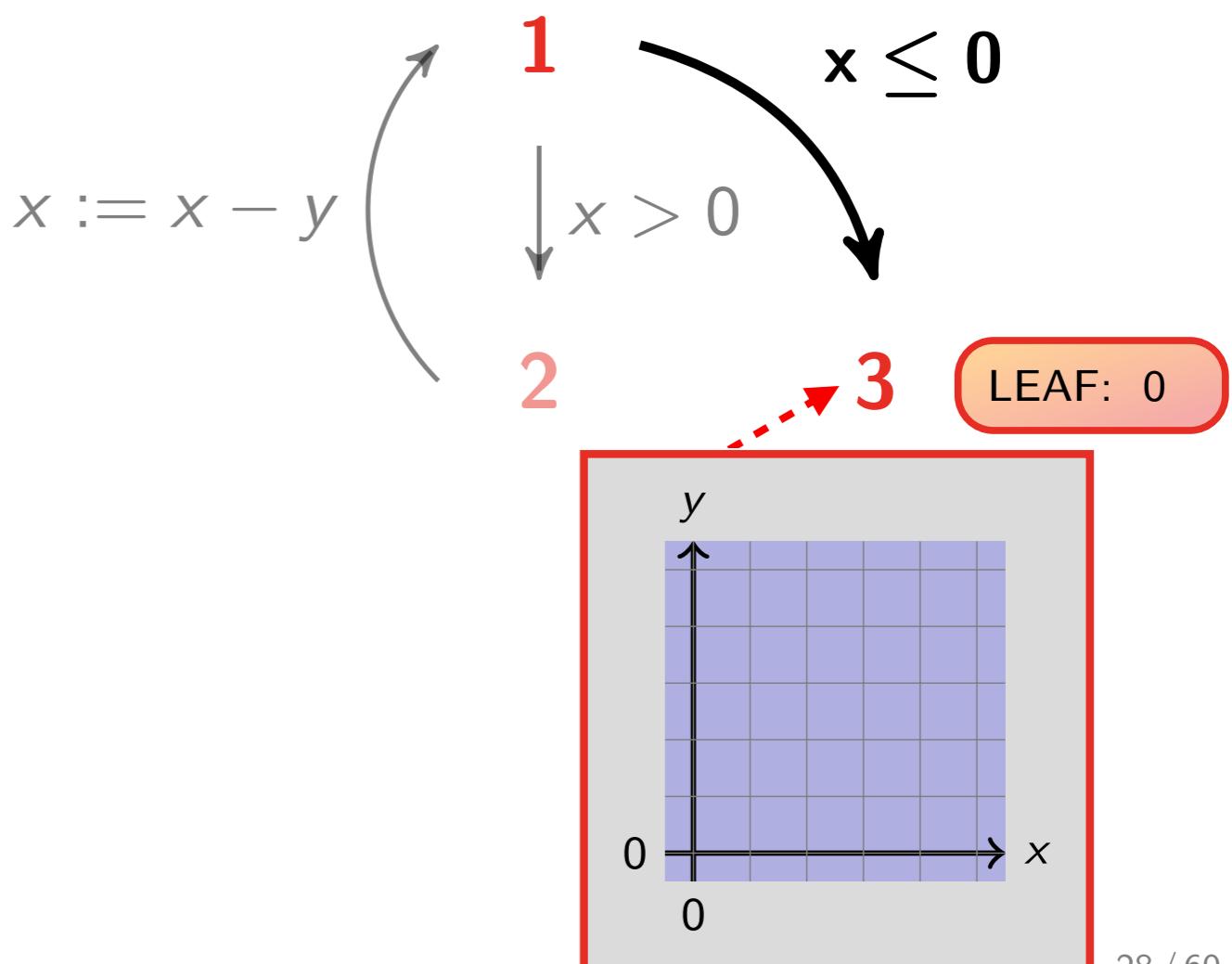
Example

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



Example

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



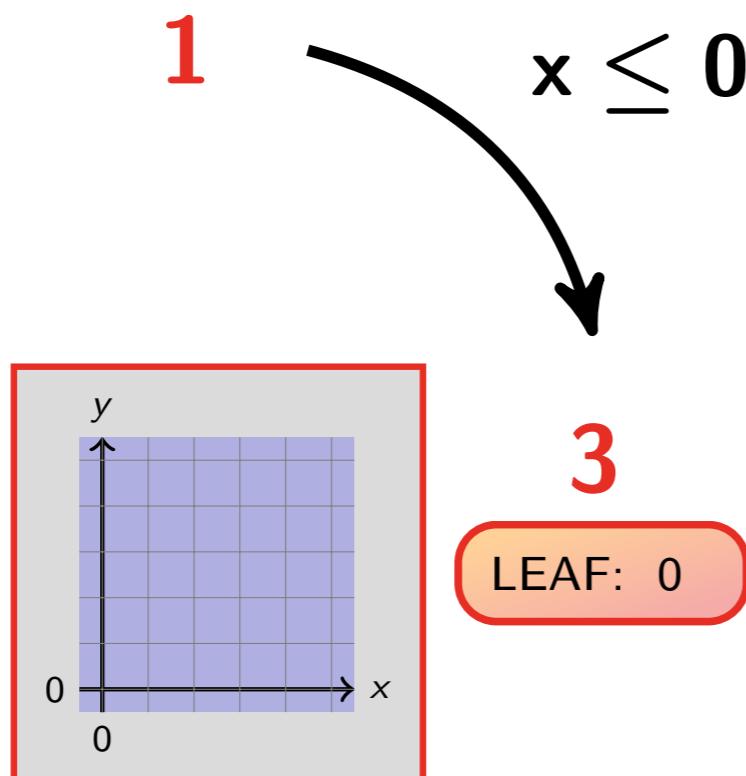
Tests

Algorithm 4 : Tree Filter

```
1: function FILTER-AUX( $t, c$ )
2:   if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq \text{LEAF} : f$  */
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( $t, c$ ),  $C$ )
```

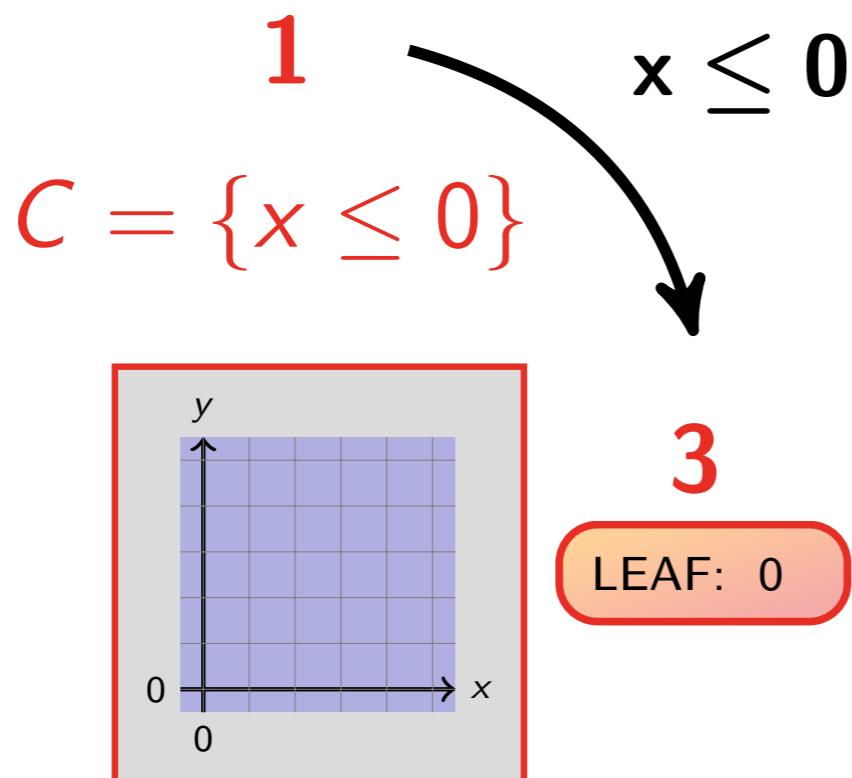


Tests

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( $t, c$ ),  $C$ )
  
```

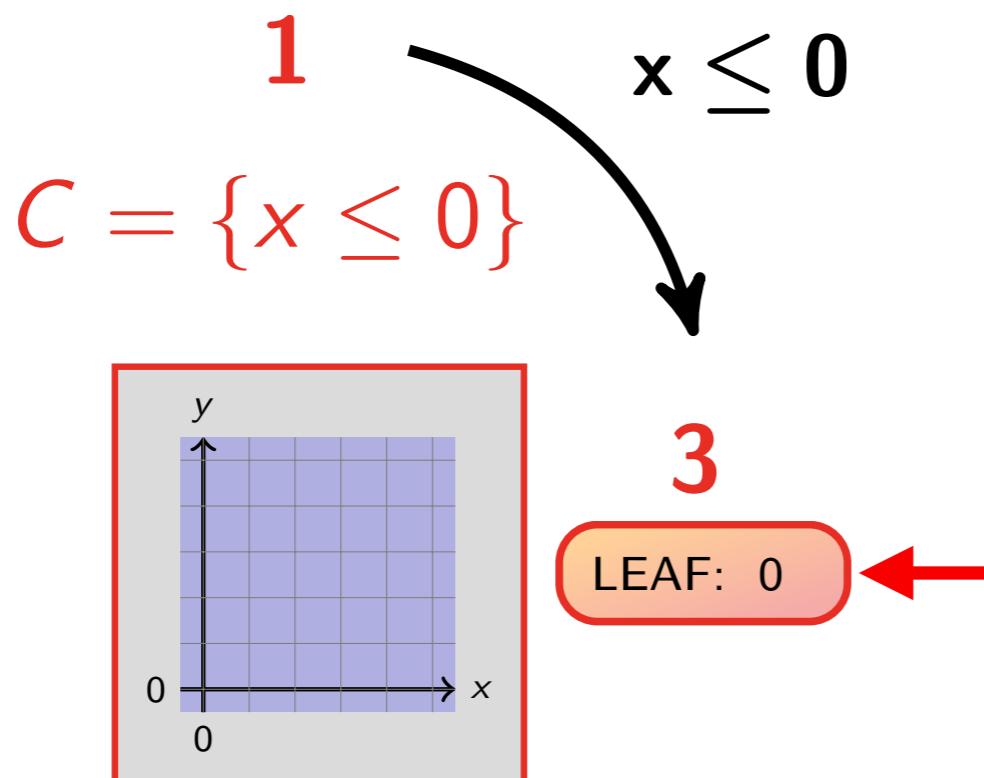


Tests

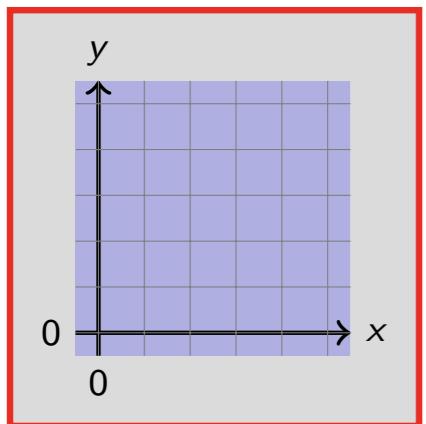
Algorithm 4 : Tree Filter

```

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( $t, c$ ),  $C$ )
  
```



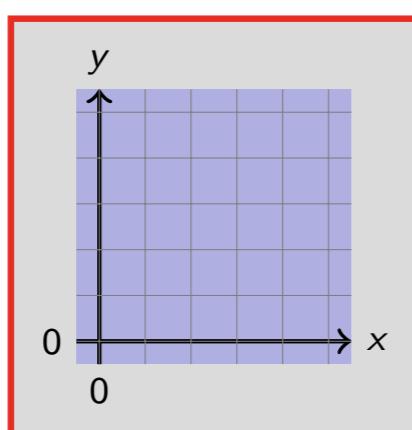
Tests



LEAF: 1

1
 $C = \{x \leq 0\}$

$x \leq 0$



3

LEAF: 0

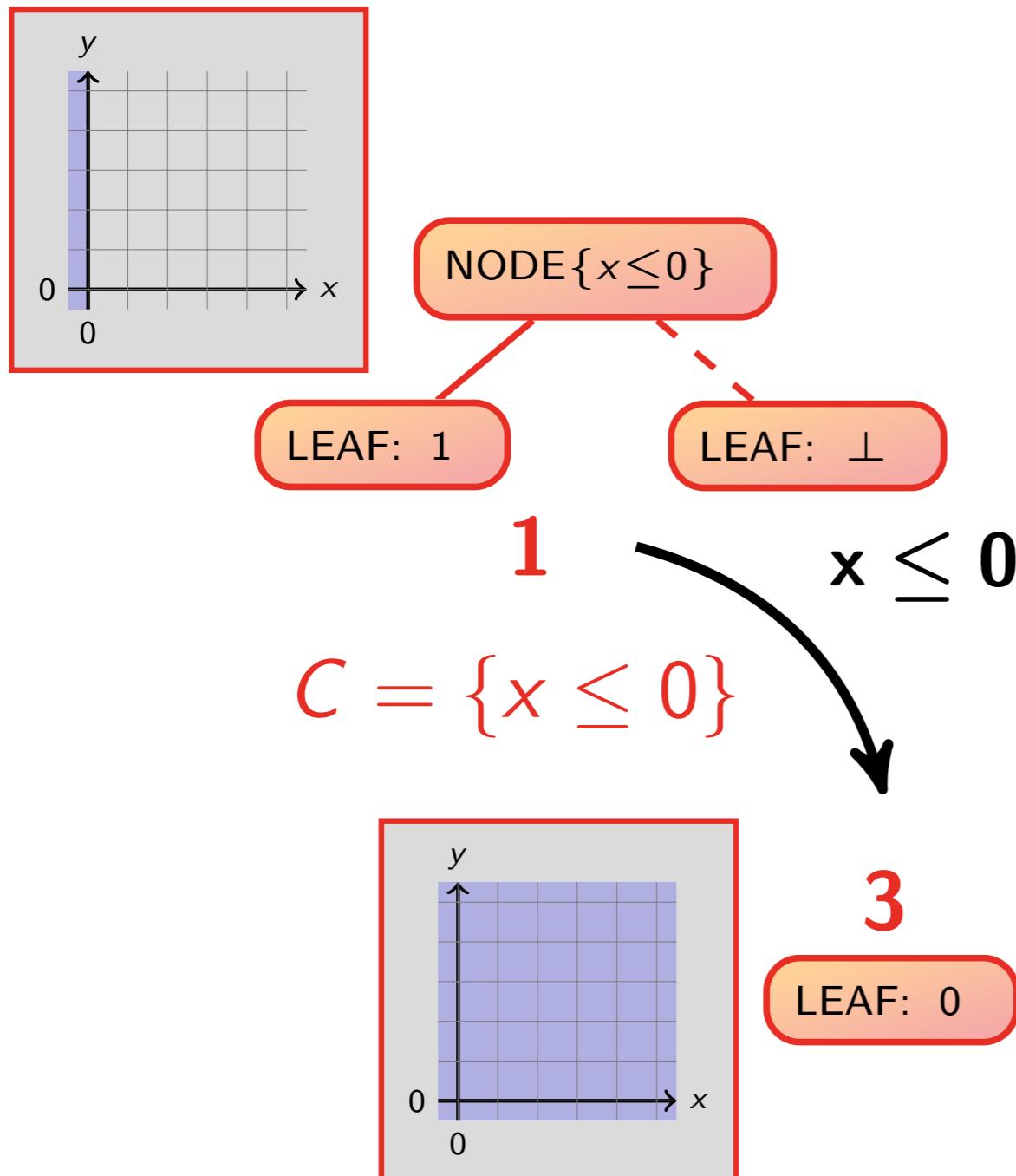
Algorithm 4 : Tree Filter

```

1: function FILTER-AUX( $t, c$ )
2:   if ISLEAF( $t$ ) then return LEAF : FILTERF( $f, c$ )           /*  $t \triangleq \text{LEAF} : f$  */
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( $t, c$ ),  $C$ )
  
```

Tests



Algorithm 4 : Tree Filter

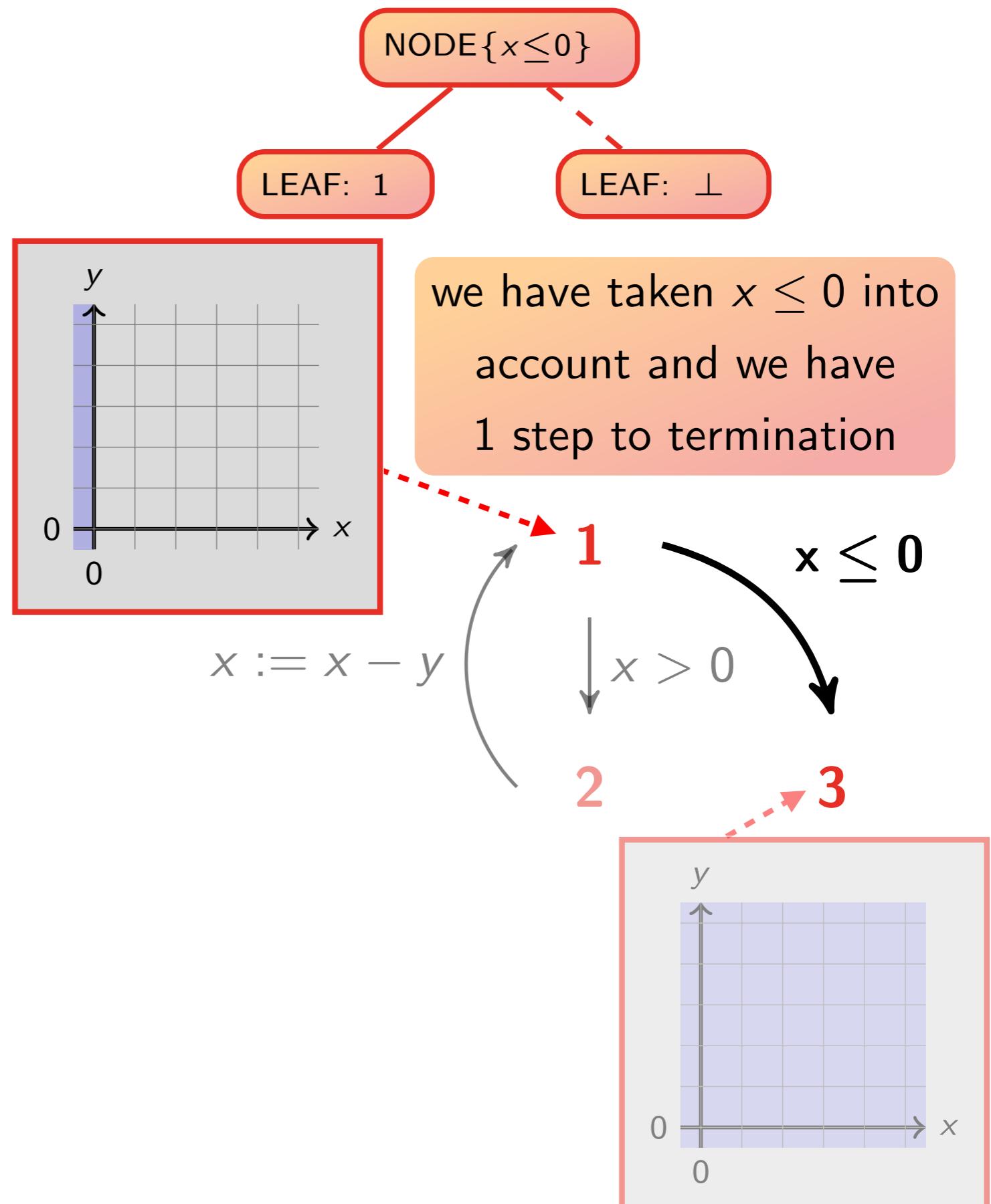
```

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( $t, c$ ),  $C$ )

```

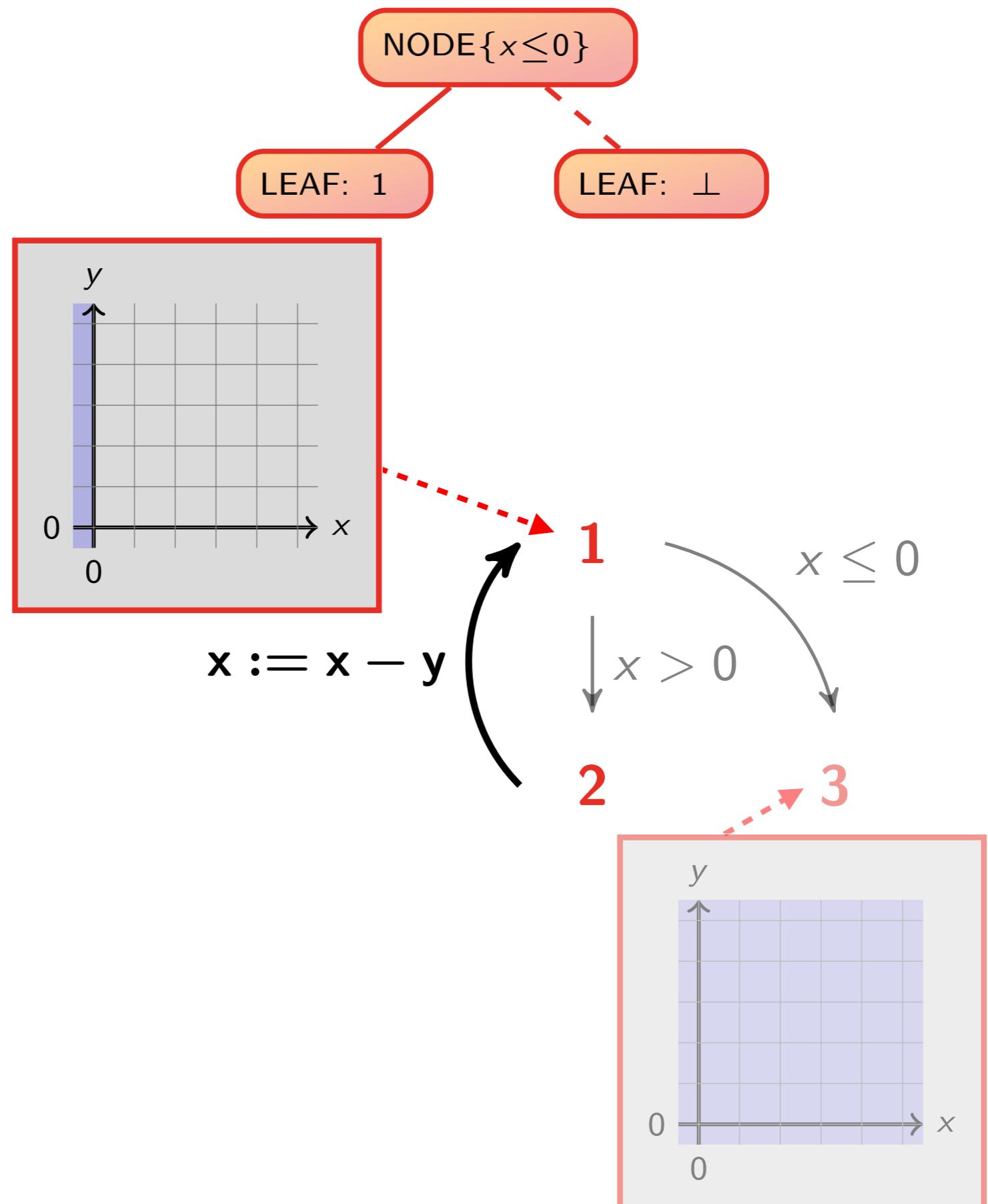
Example

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

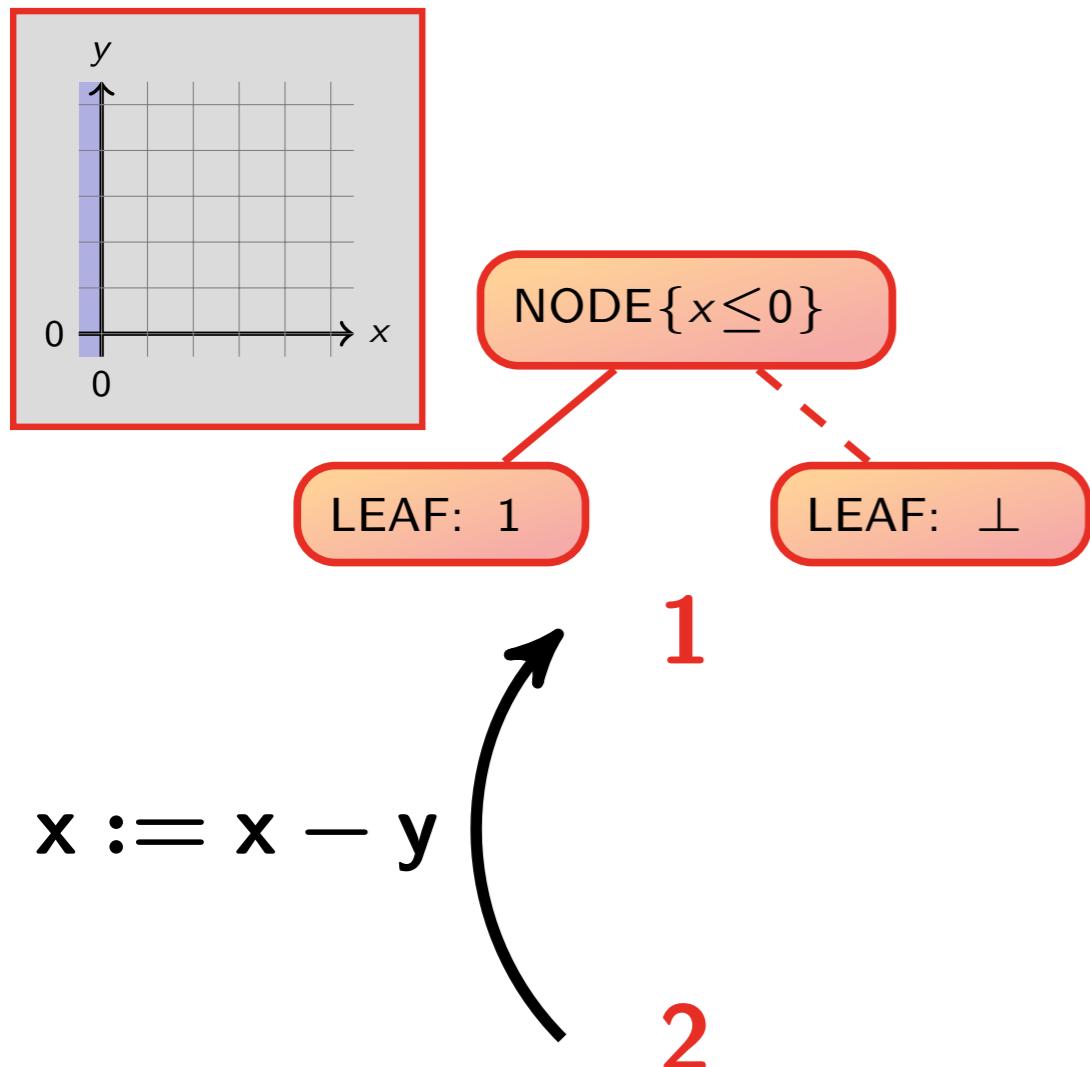


Example

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



Assignments



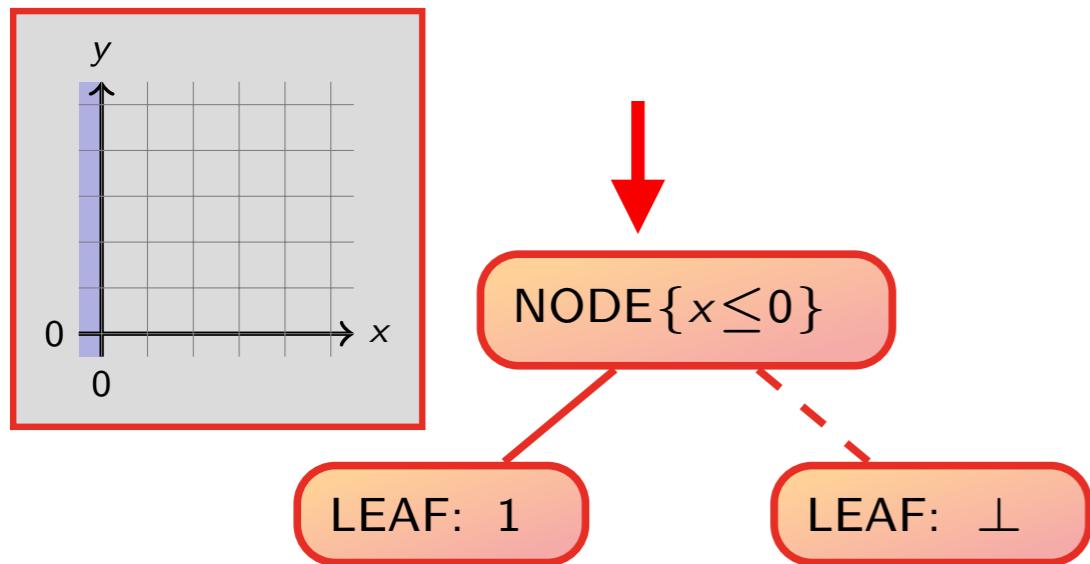
Algorithm 3 : Tree Assign

```

1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )           /*  $t \triangleq \text{LEAF} : f$  */
3:   else
4:      $C \leftarrow \text{ASSIGN}_L(t.c, x := a)$ 
5:     if ISEMPTRY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_T$  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



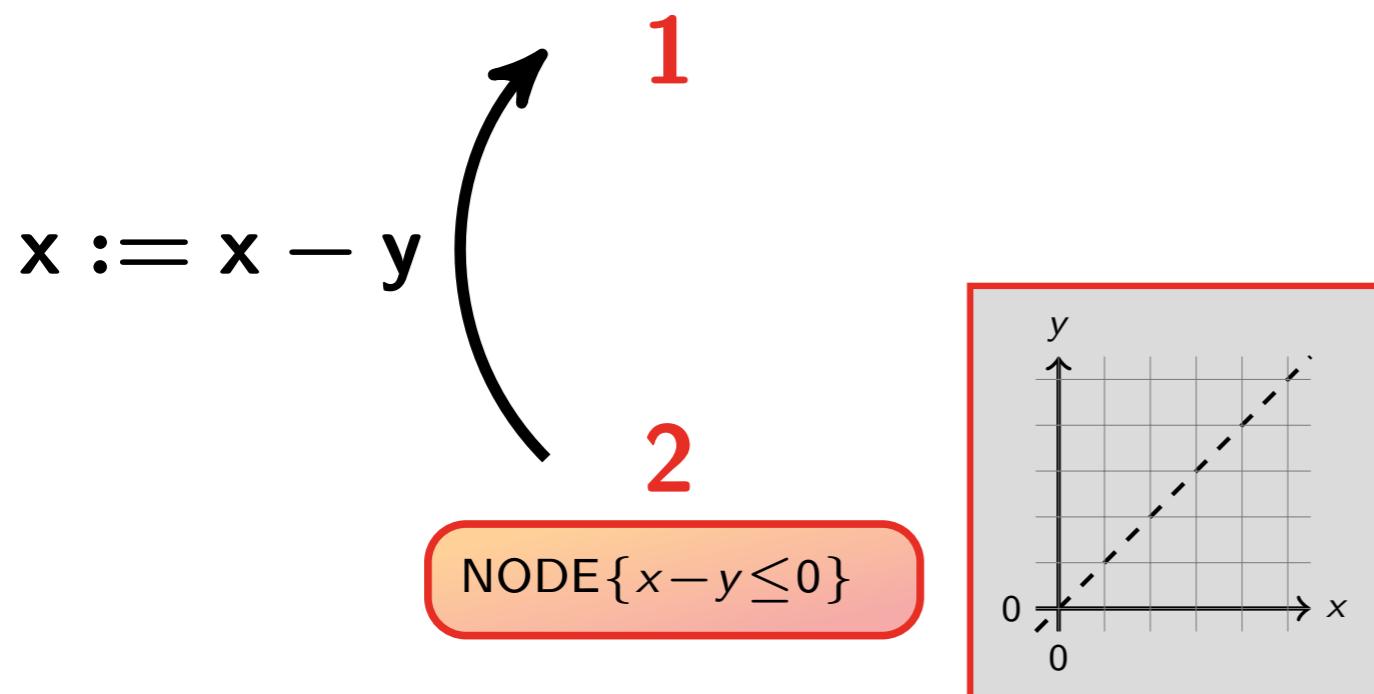
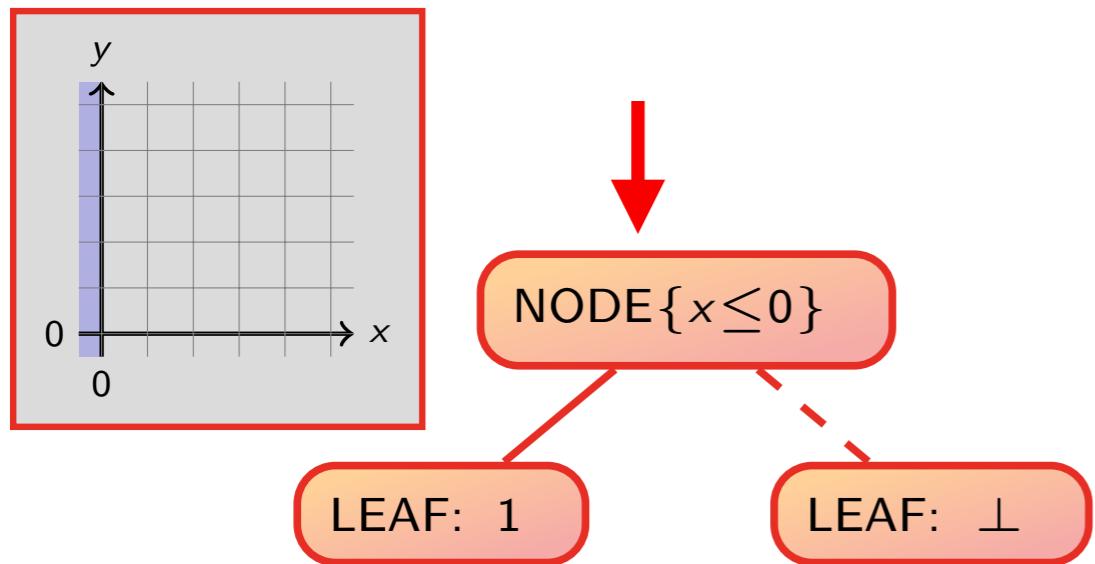
Algorithm 3 : Tree Assign

```

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 ISEMPTRY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_T$  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{ $l.c$ } :  $l; r$ 
  
```

$x := x - y$ 1
 2

Assignments

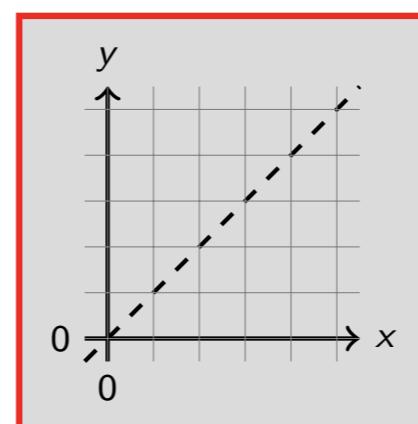
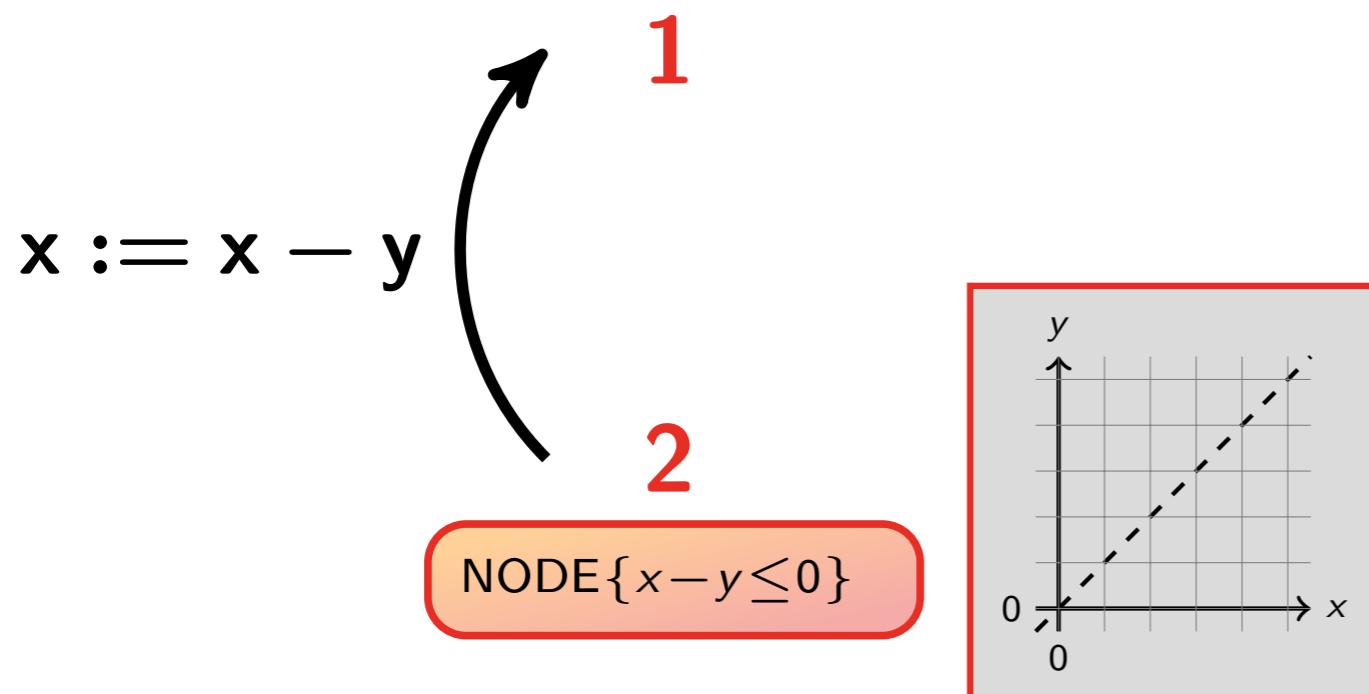
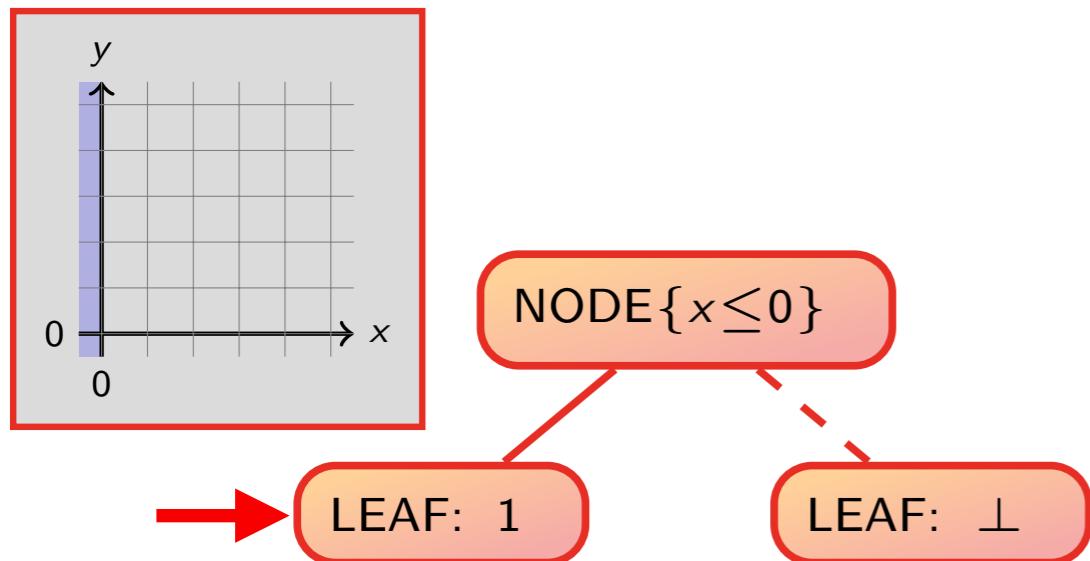


Algorithm 3 : Tree Assign

```

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( $t.l, x := a$ )  $\sqcup_T$  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

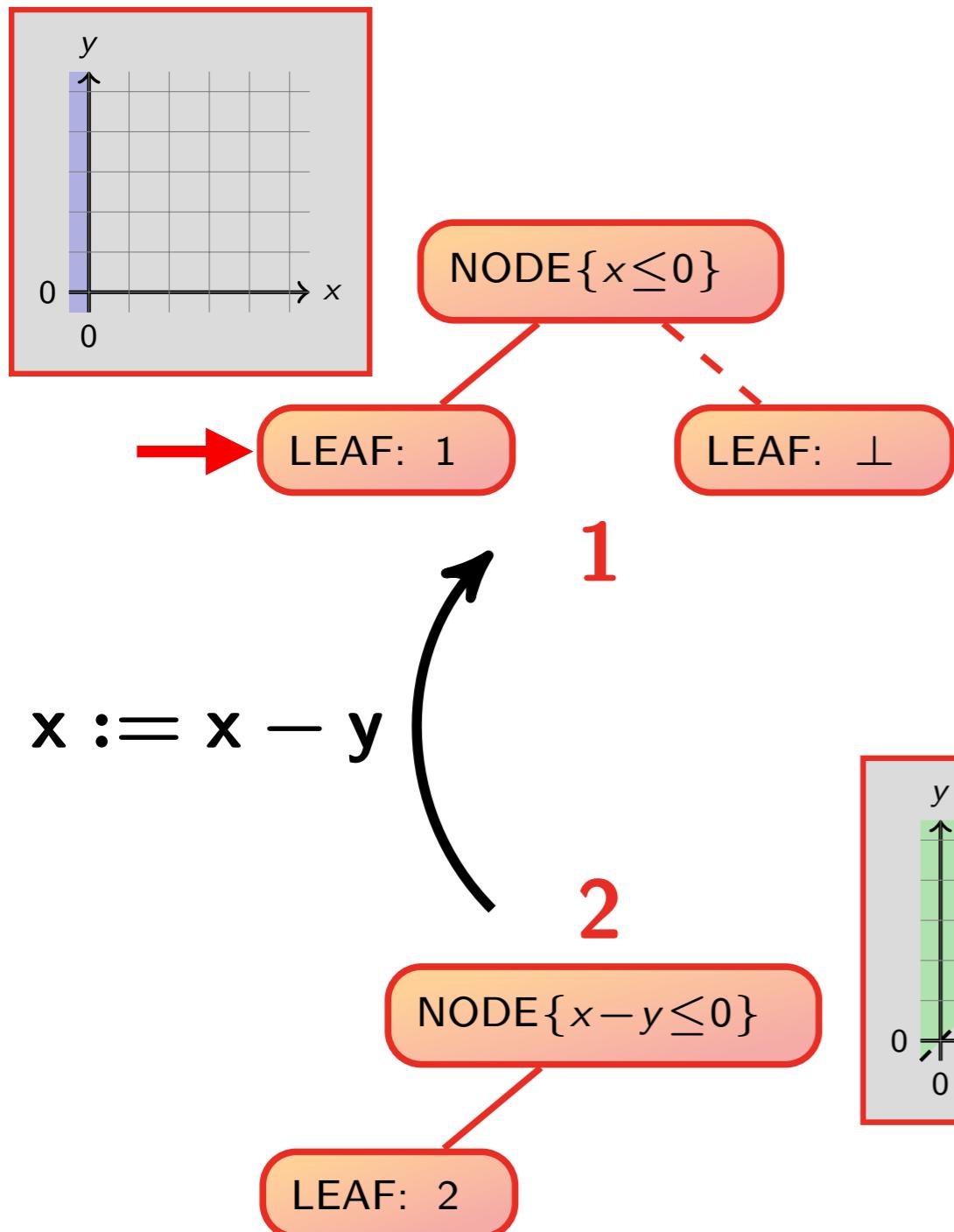


Algorithm 3 : Tree Assign

```

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 ISEMPHY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_T$  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

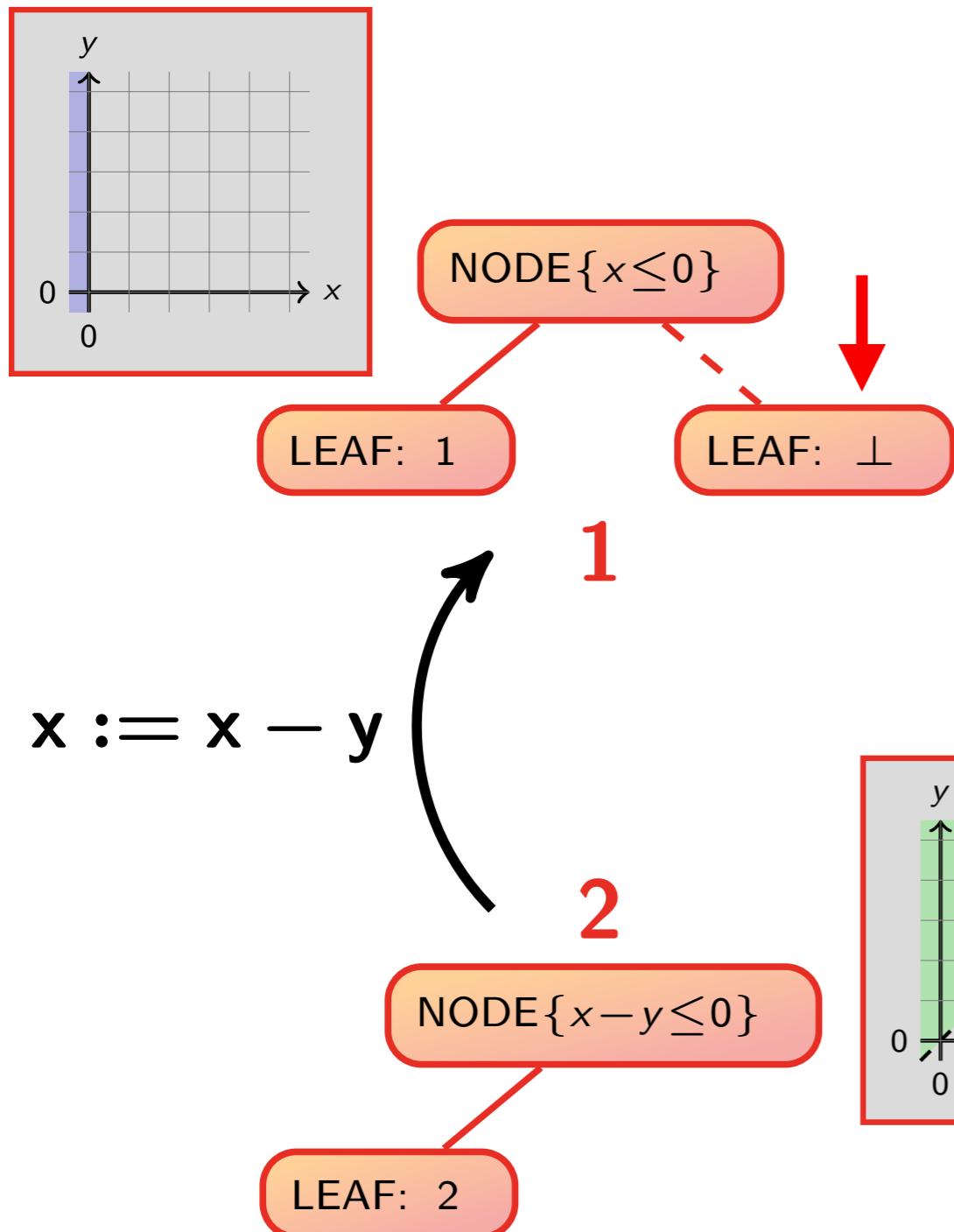


Algorithm 3 : Tree Assign

```

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 ISEMPty( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_T$  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

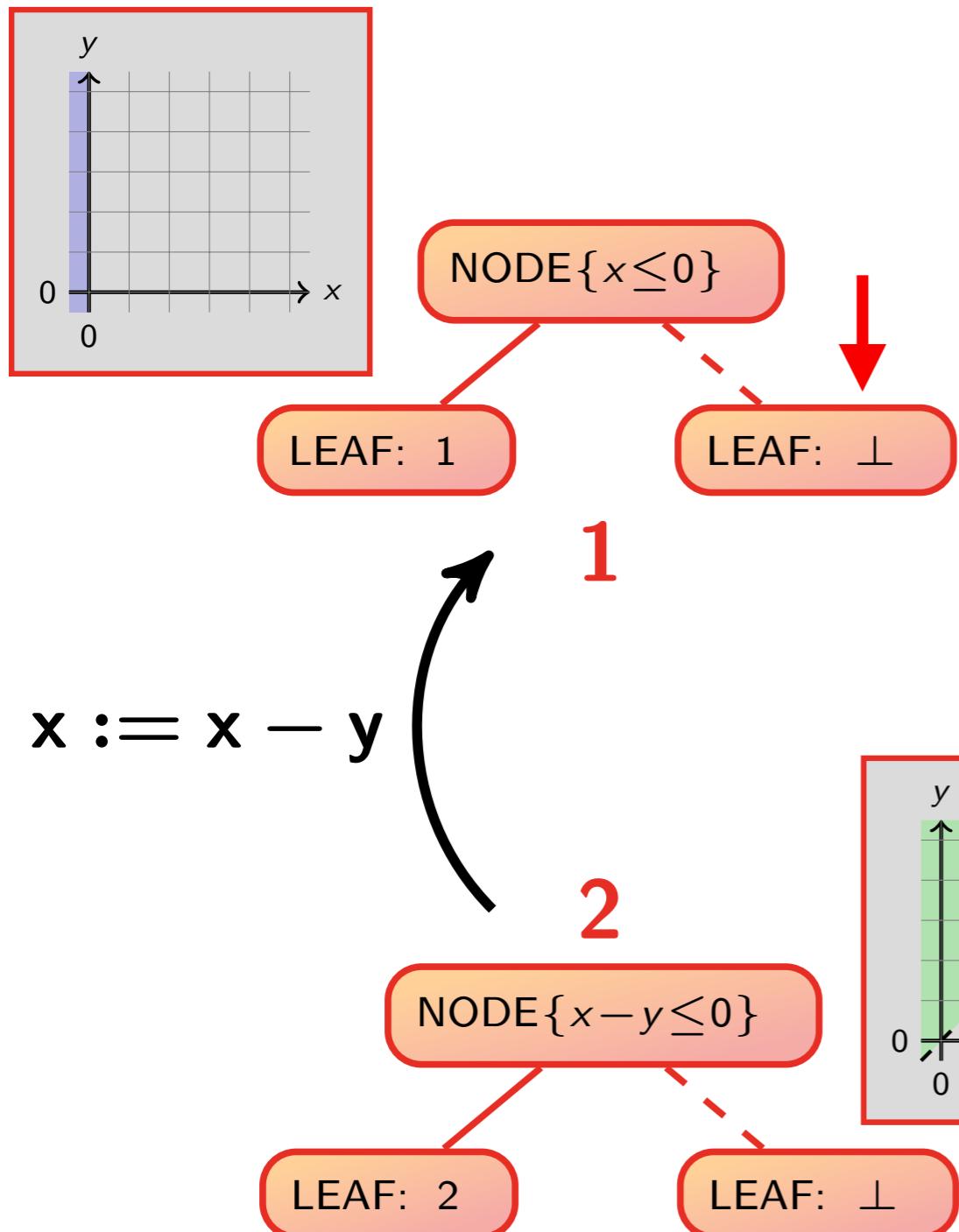


Algorithm 3 : Tree Assign

```

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 ISEMPHY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_T$  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

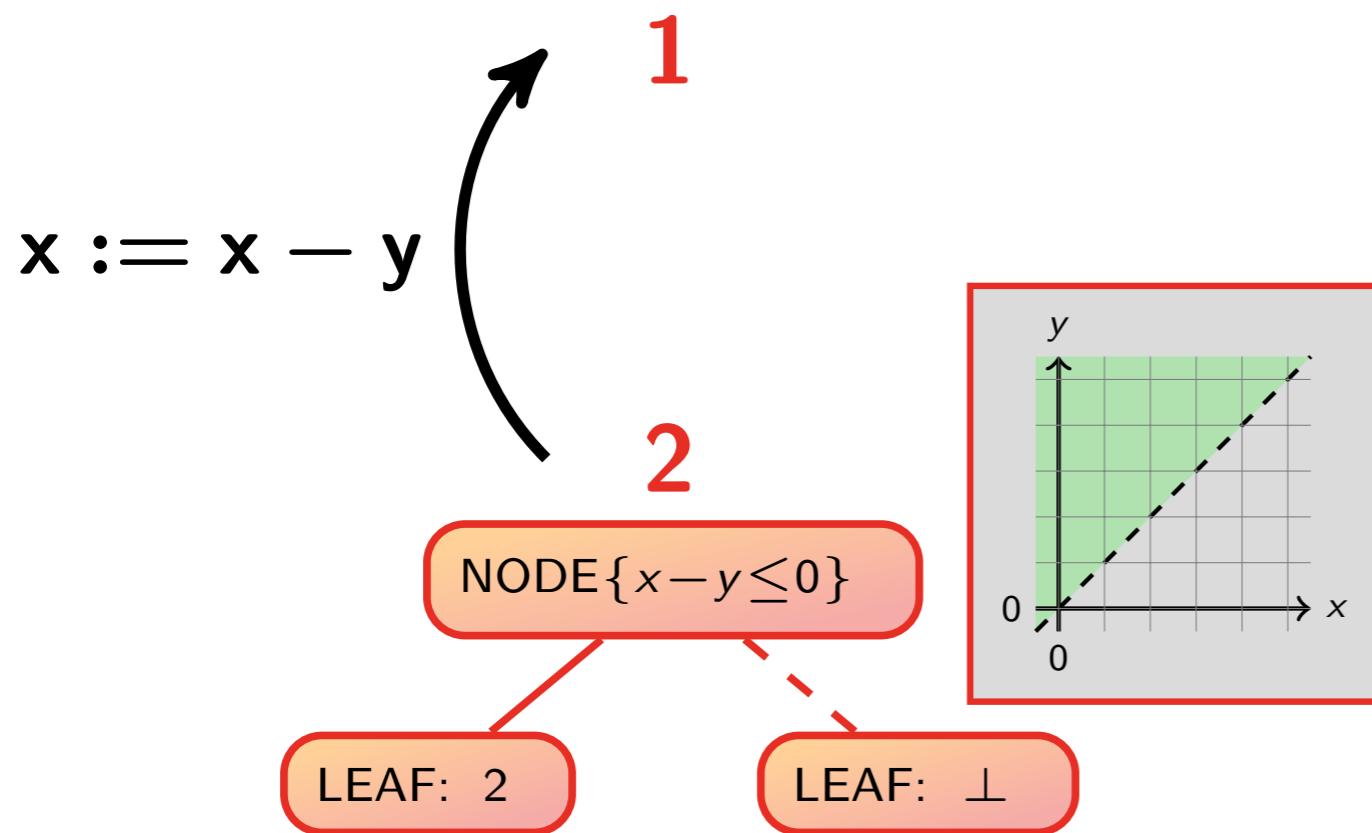
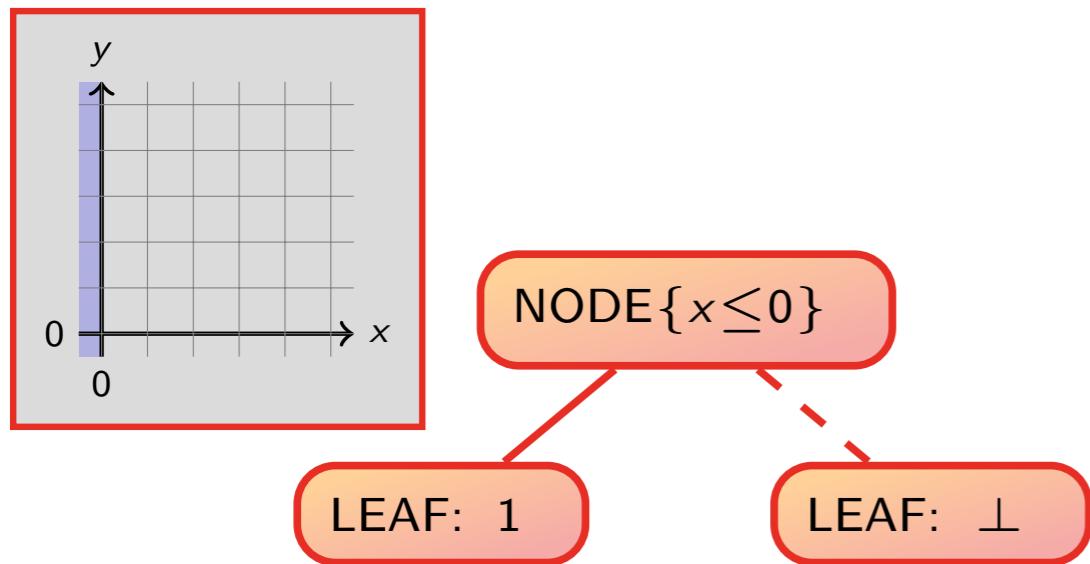


Algorithm 3 : Tree Assign

```

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 ISEMPTRY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_T$  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

```

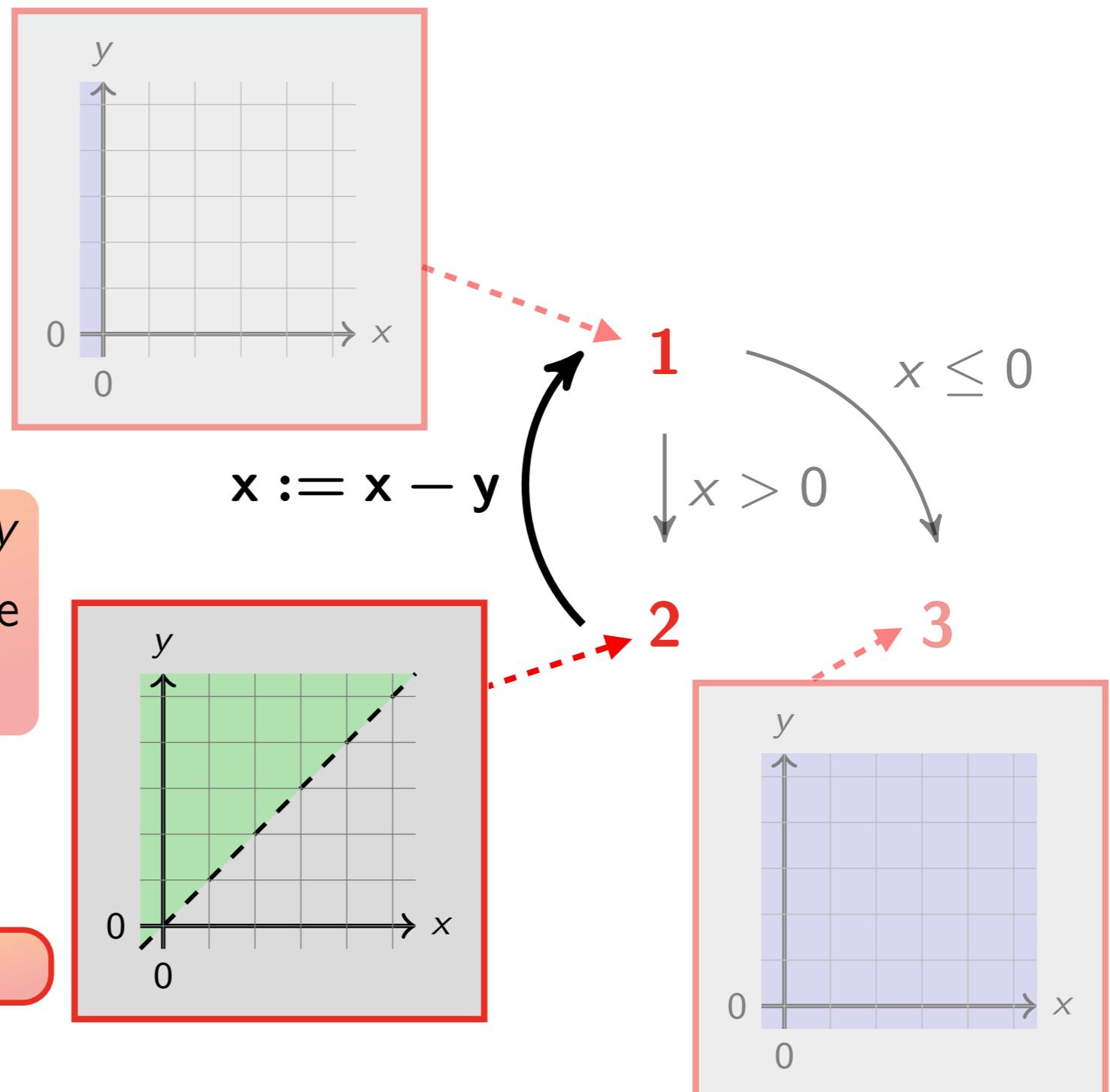
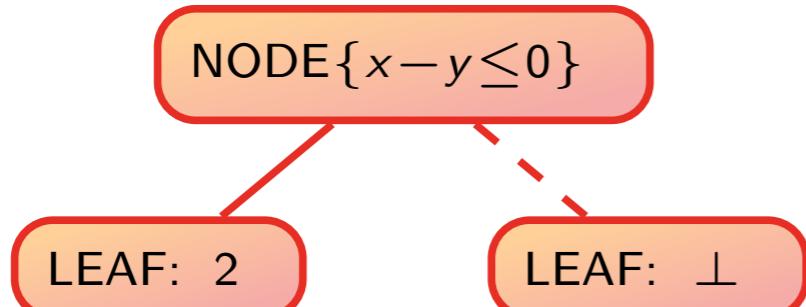
1: function ASSIGN( $t, x := a$ )
2:   if ISLEAF( $t$ ) then return LEAF : ASSIGNF( $f, x := a$ )           /*  $t \triangleq \text{LEAF} : f$  */
3:   else
4:      $C \leftarrow \text{ASSIGN}_L(t.c, x := a)$ 
5:     if ISEMPTRY( $C$ ) then return ASSIGN( $t.l, x := a$ )  $\sqcup_T$  ASSIGN( $t.r, x := a$ )
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 $\{l.c\}$  :  $l; r$ 

```

Example

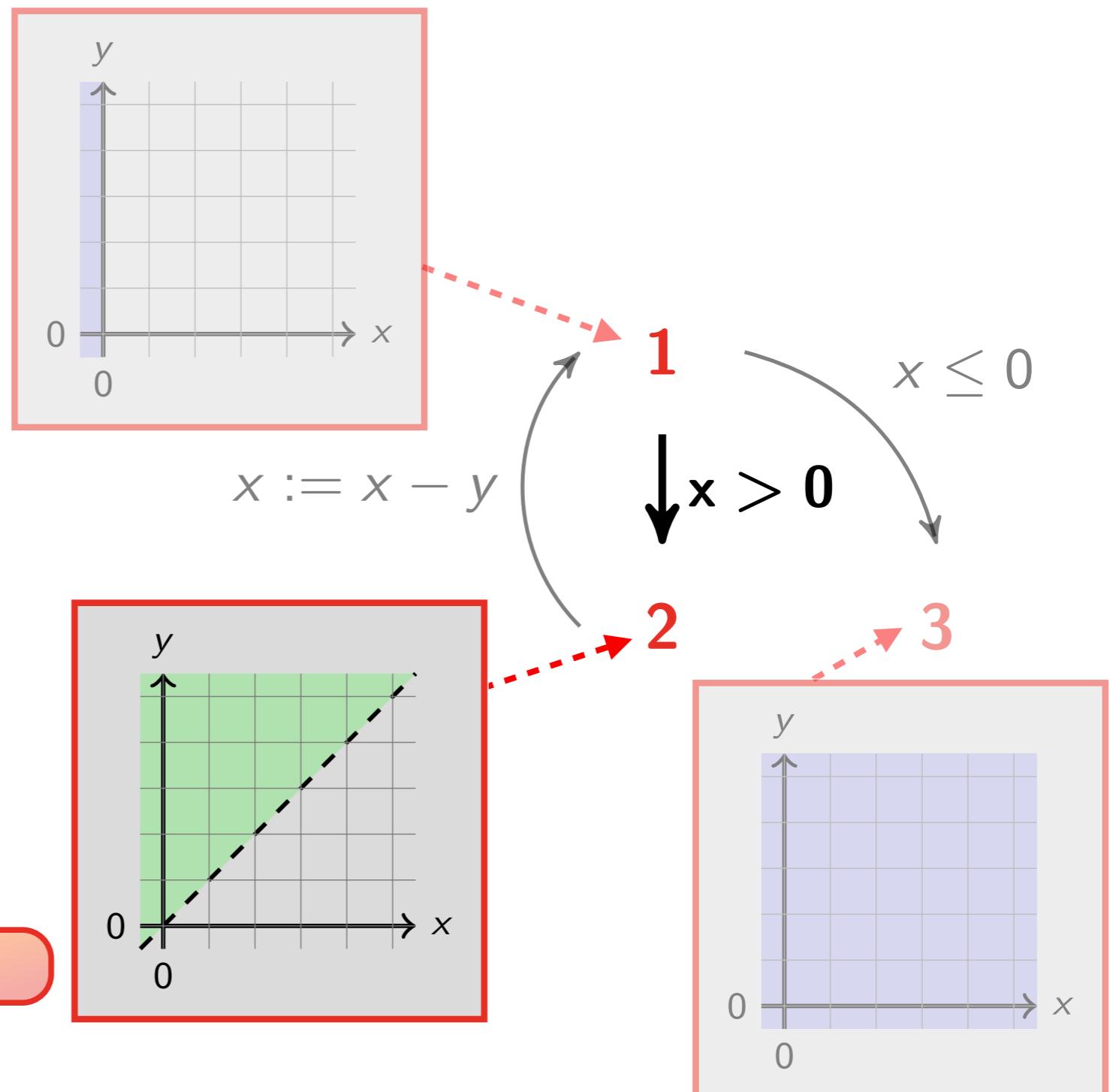
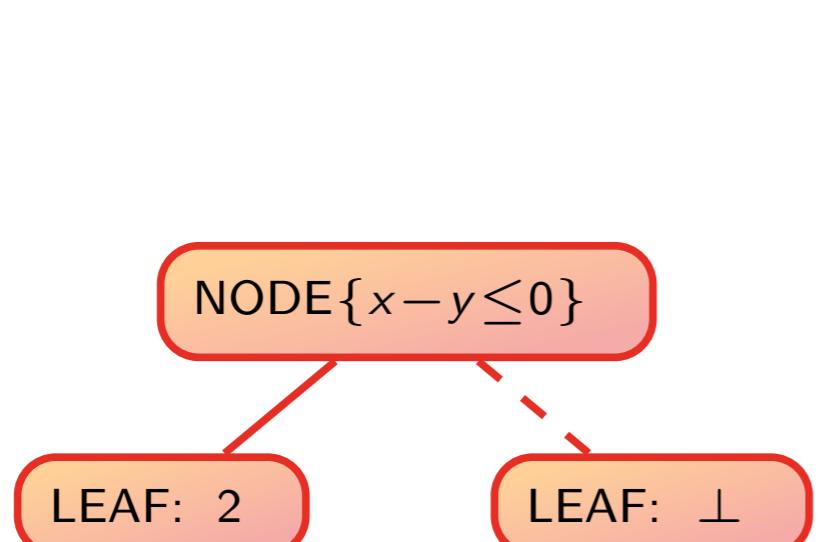
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

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
```

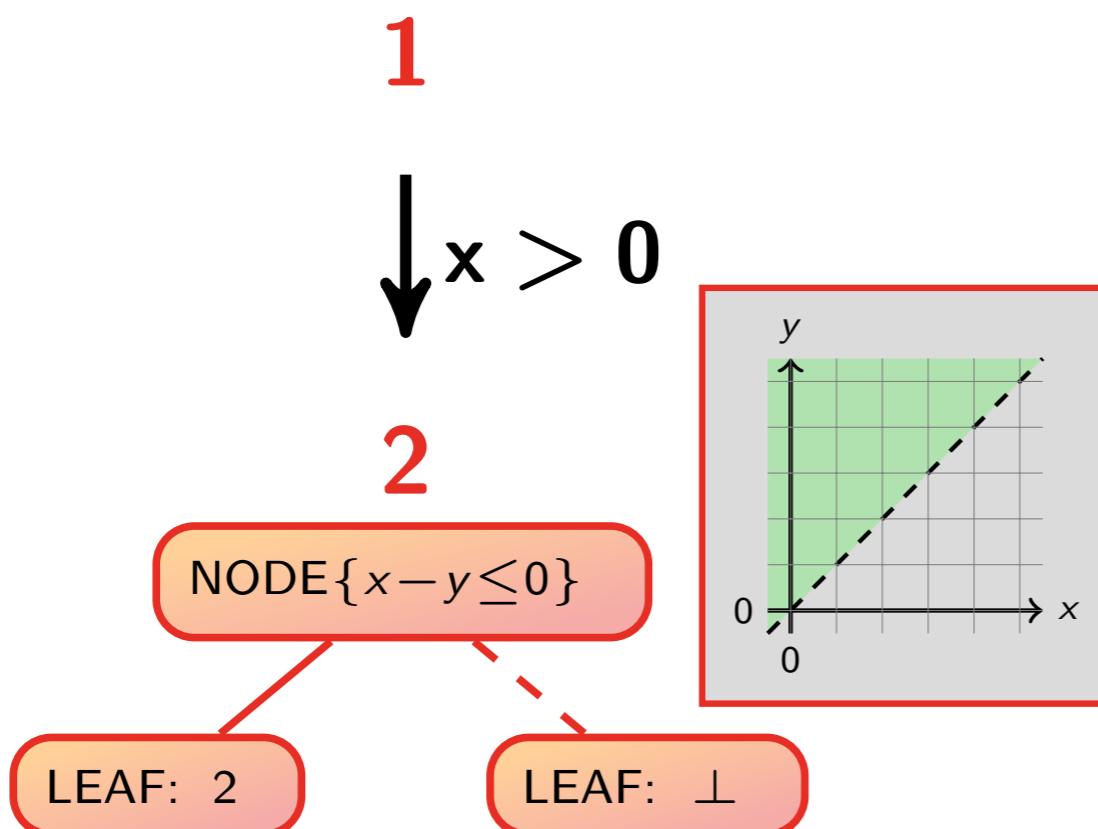


Tests

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( $t, c$ ),  $C$ )
  
```

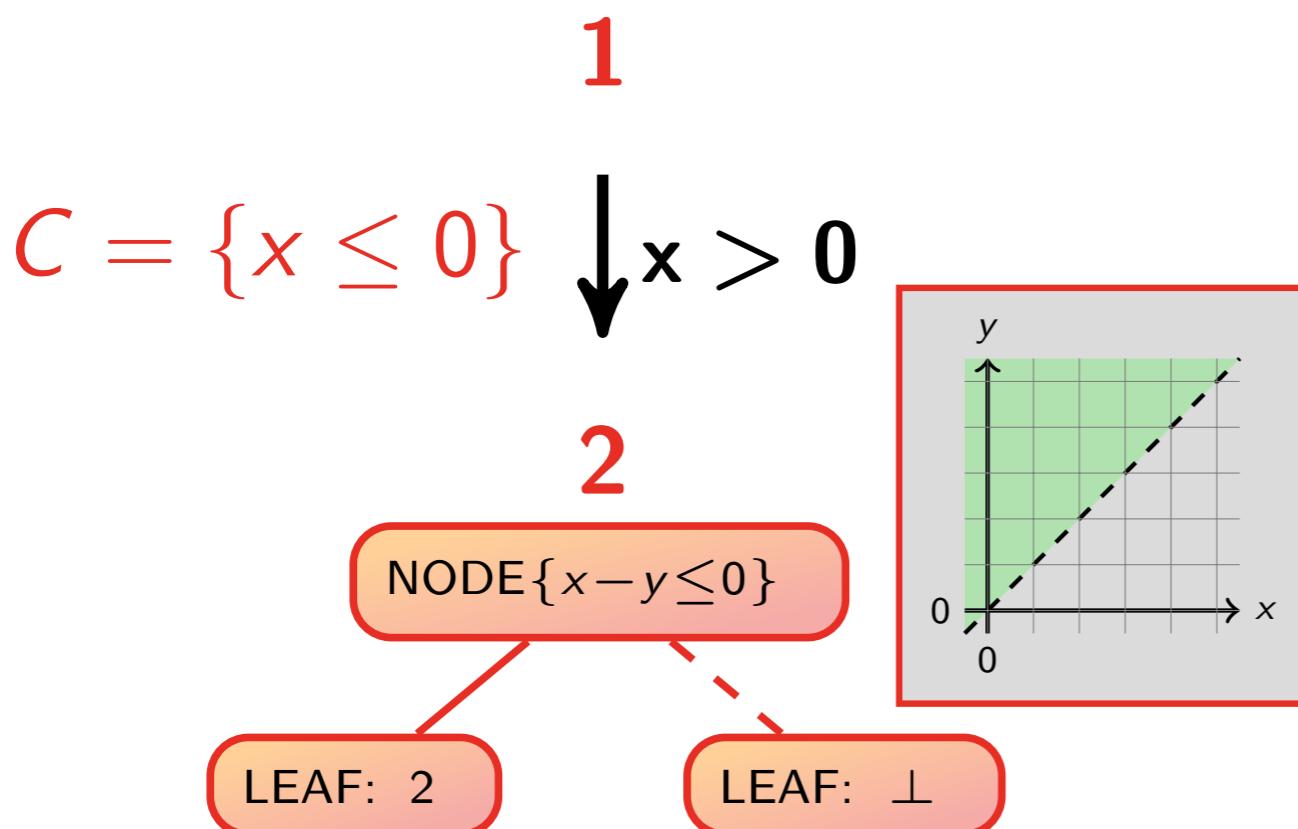


Tests

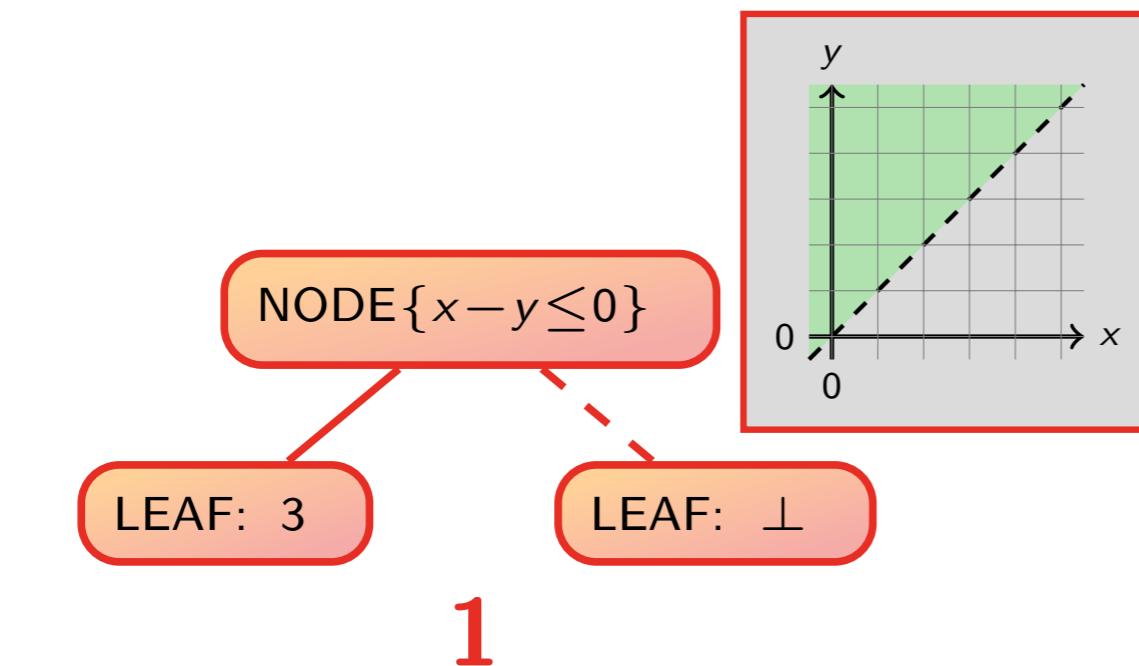
Algorithm 4 : Tree Filter

```

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( $t, c$ ),  $C$ )
  
```



Tests



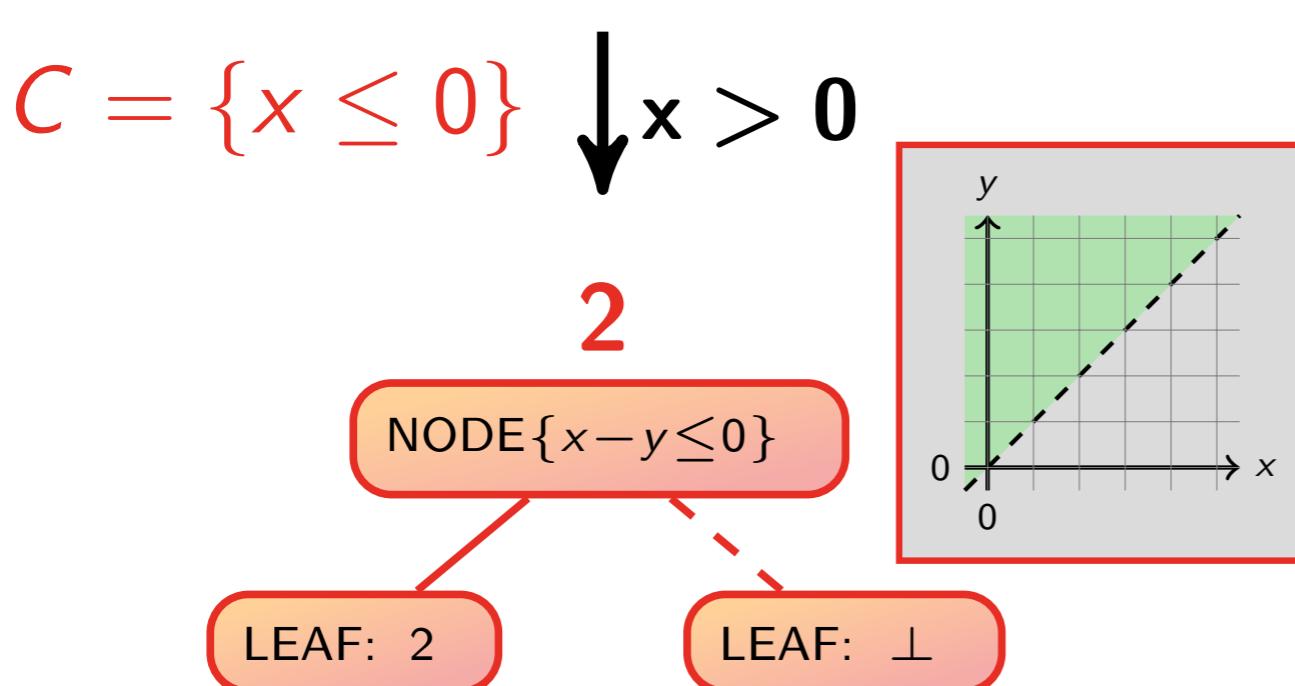
thm 4 : Tree Filter

unction FILTER-AUX(t, c)

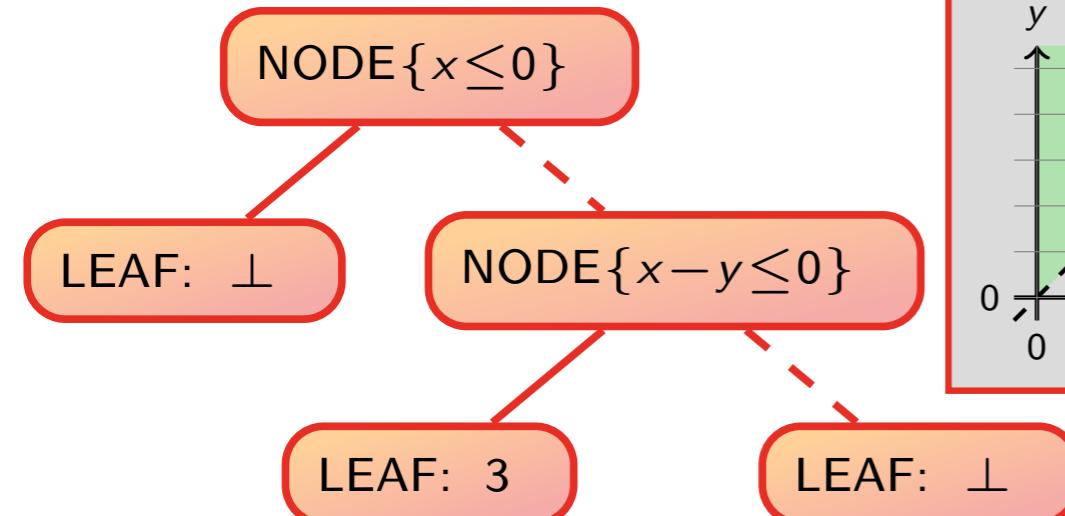
if ISLEAF(t) **then return** LEAF : FILTER_F(f, c) /* $t \triangleq \text{LEAF} : f$ */
else return NODE $\{t.c\}$: FILTER-AUX($t.l, c$); FILTER-AUX($t.r, c$)

unction FILTER(t, c)

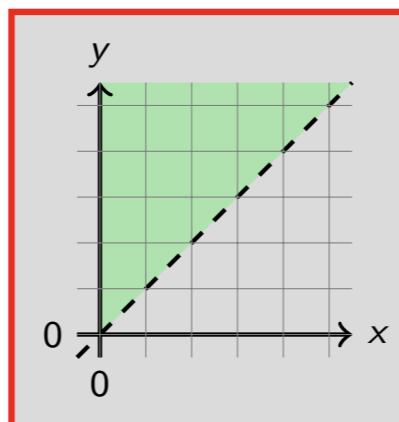
$C \leftarrow \text{FILTER}_L(c)$
return AUGMENT(FILTER-AUX(t, c), C)



Tests



1



thm 4 : Tree Filter

unction FILTER-AUX(t, c)

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

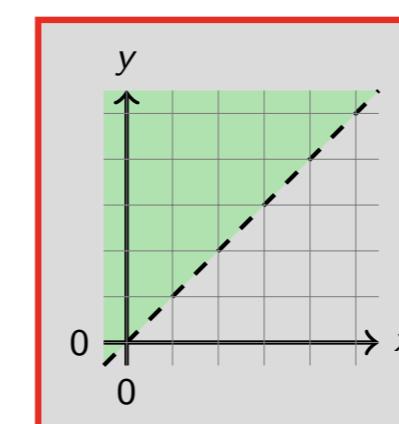
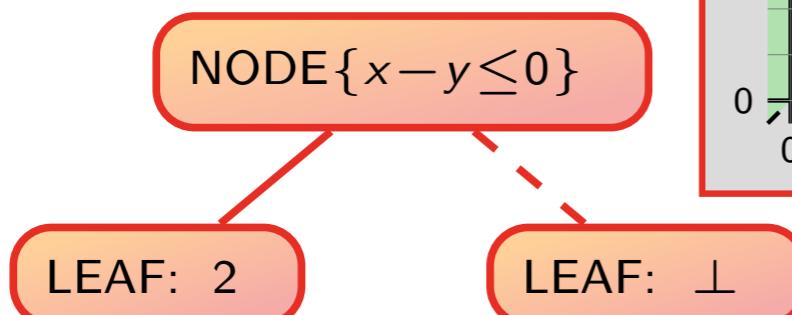
unction FILTER(t, c)

$C \leftarrow \text{FILTER}_L(c)$
return AUGMENT(FILTER-AUX(t, c), C)

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

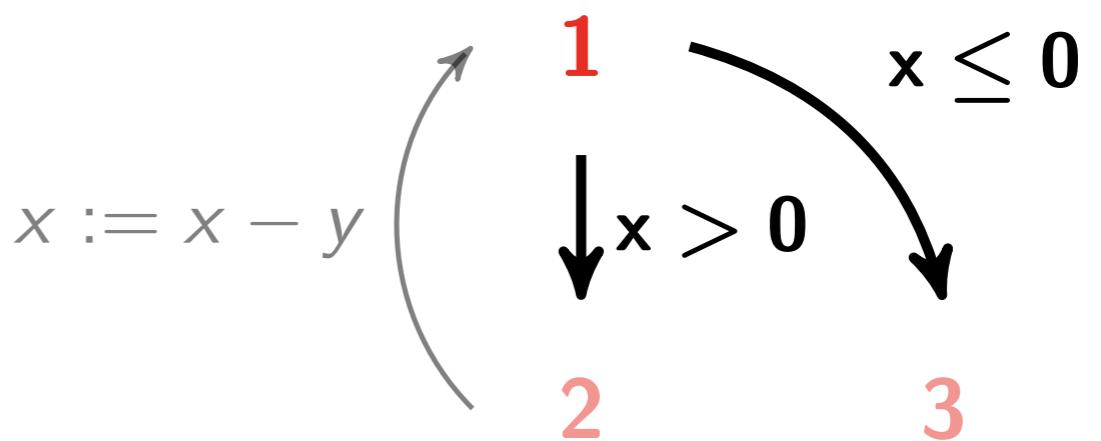
$\downarrow x > 0$

2

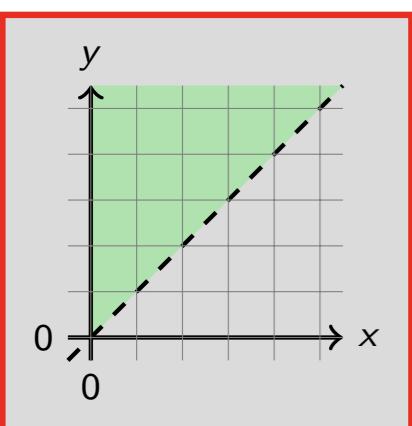


Example

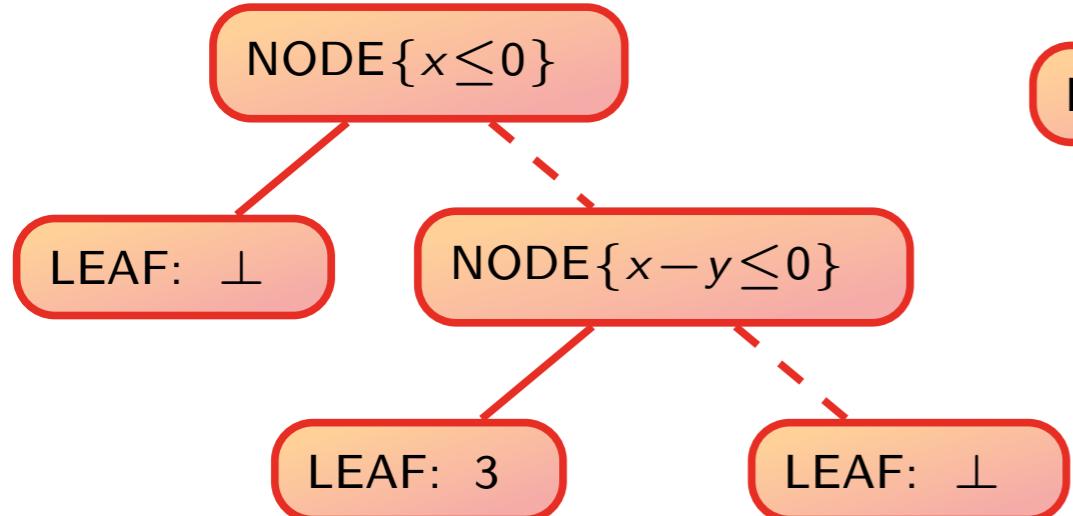
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



Join



1



Algorithm 1 : Tree Unification

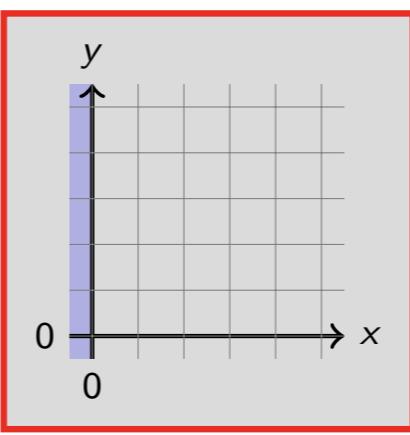
```

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) 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( $t_2$ )  $\wedge$   $t_1.c <_{\mathbb{L}} t_2.c$ ) 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:     ( $r_2, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
  
```

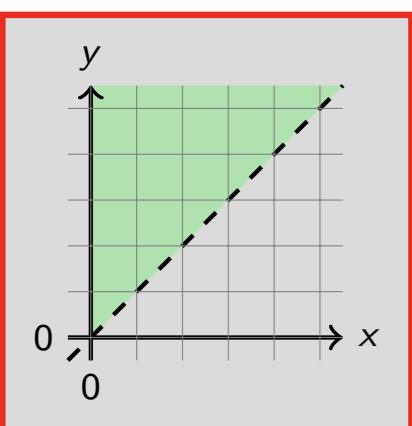
NODE{ $x \leq 0$ }

LEAF: 1

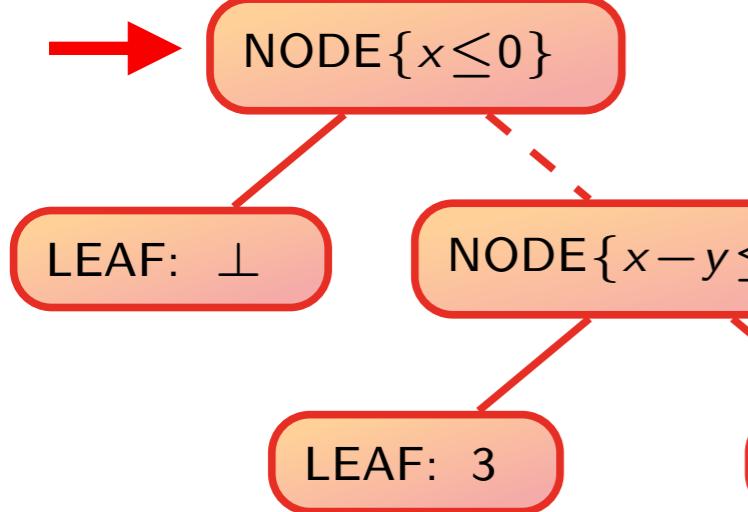
LEAF: ⊥



Join



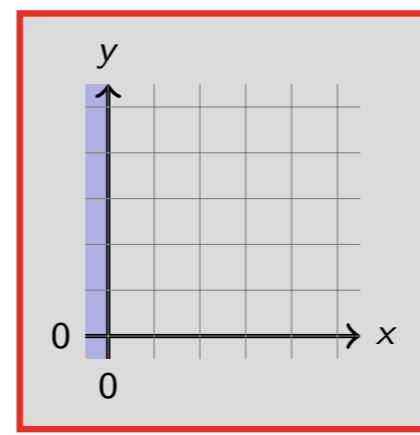
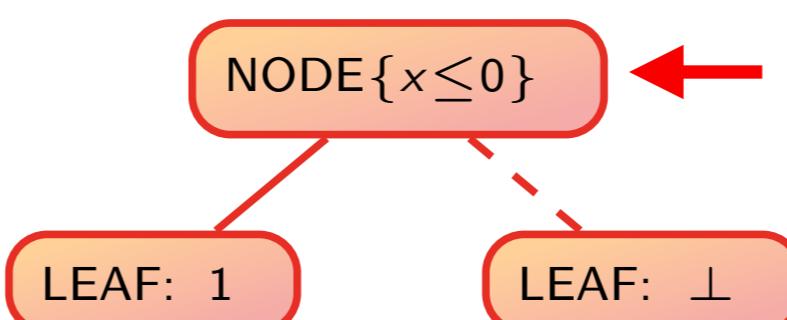
1



Algorithm 1 : Tree Unification

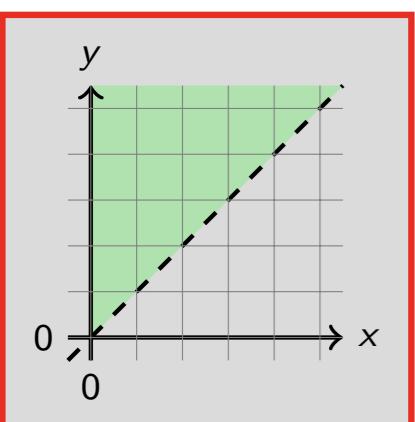
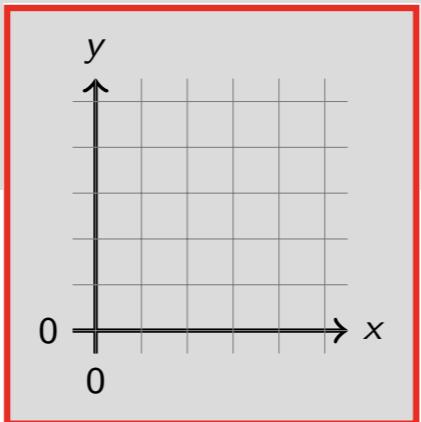
```

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) 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( $t_2$ )  $\wedge$   $t_1.c <_{\mathbb{L}} t_2.c$ ) 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:     ( $r_2, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE{ $t_1.c$ } :  $l_1; r_1$ , NODE{ $t_2.c$ } :  $l_2; r_2$ )
  
```

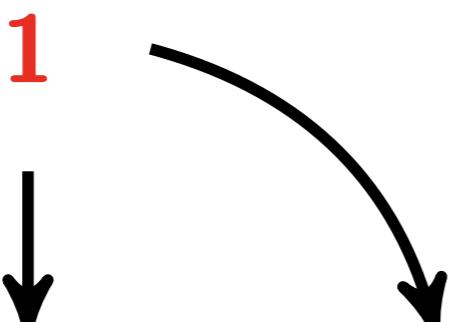


Join

NODE $\{x \leq 0\}$



1



NODE $\{x \leq 0\}$

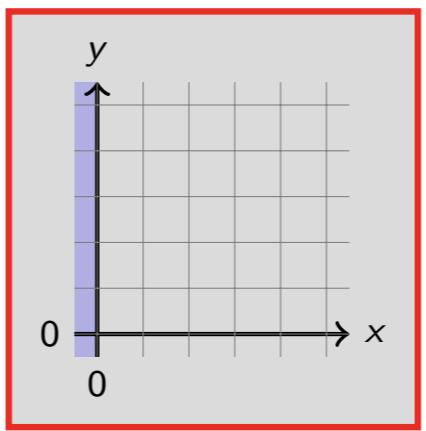
LEAF: \perp

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

LEAF: 3

LEAF: 1

LEAF: \perp



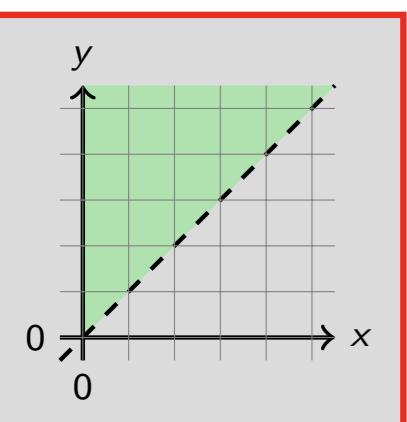
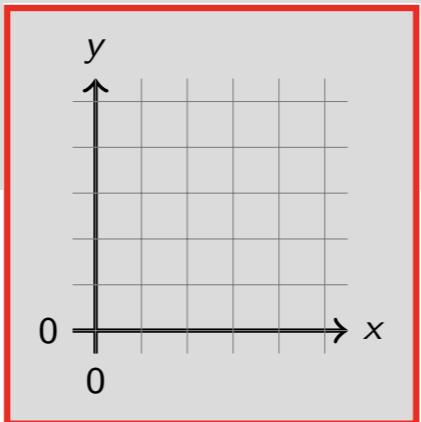
Algorithm 1 : Tree Unification

```

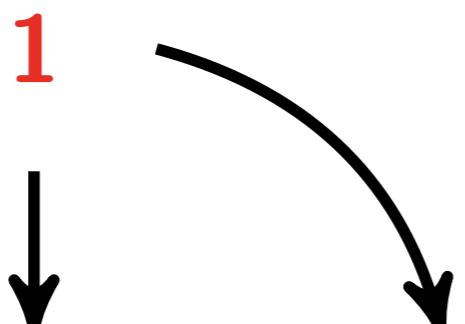
1: function UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) then
5:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1, t_2.l$ )
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( $t_2$ )  $\wedge$   $t_1.c <_{\mathbb{L}} t_2.c$ ) 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:     ( $r_2, r_2$ )  $\leftarrow$  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 $\{x \leq 0\}$



1



NODE $\{x \leq 0\}$

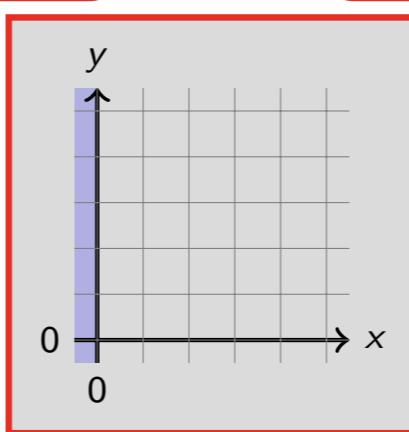
LEAF: \perp

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

LEAF: 3

LEAF: 1

LEAF: \perp

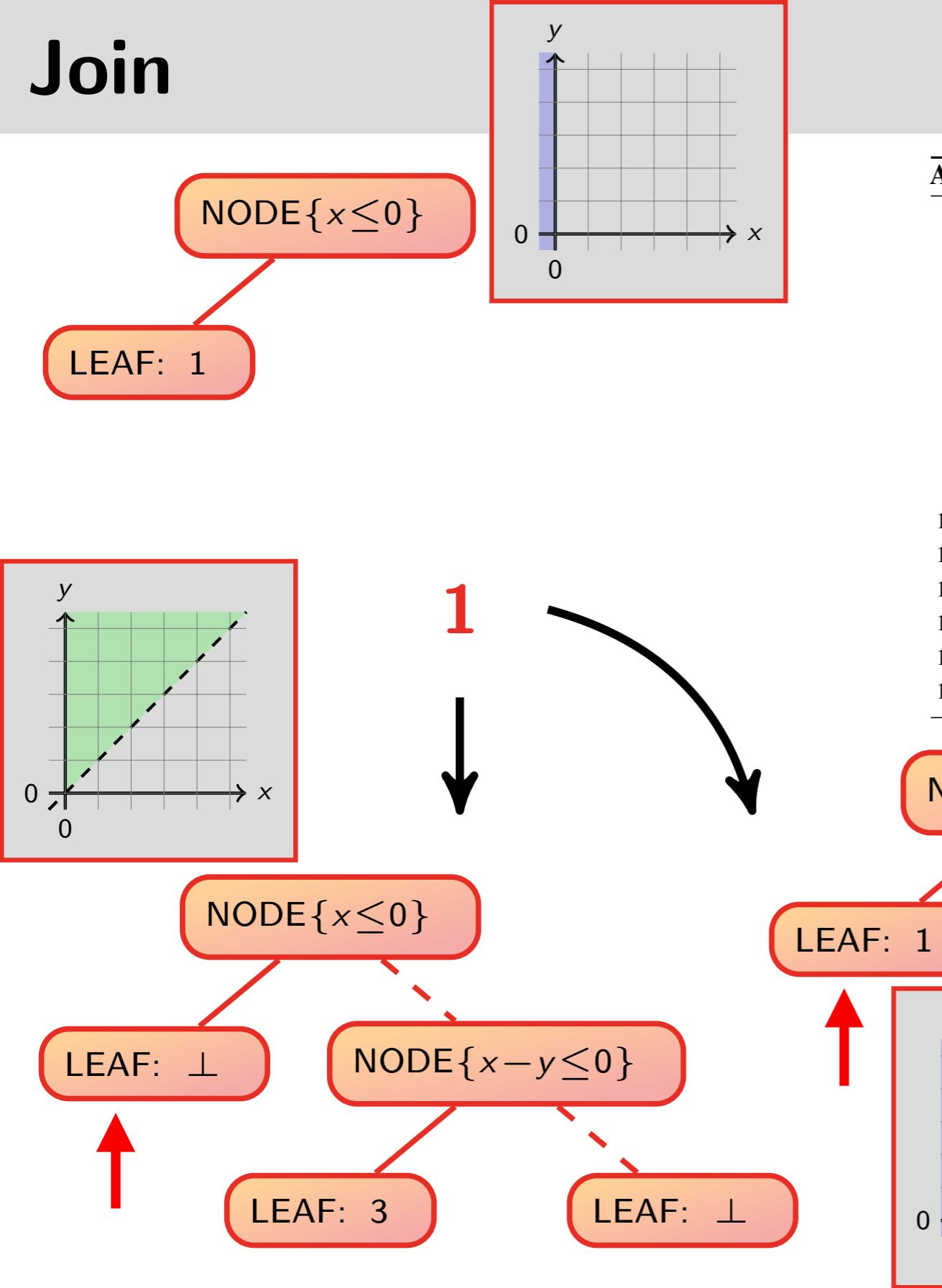


Algorithm 1 : Tree Unification

```

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) 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( $t_2$ )  $\wedge$   $t_1.c <_{\mathbb{L}} t_2.c$ ) 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:     ( $r_2, r_2$ )  $\leftarrow$  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



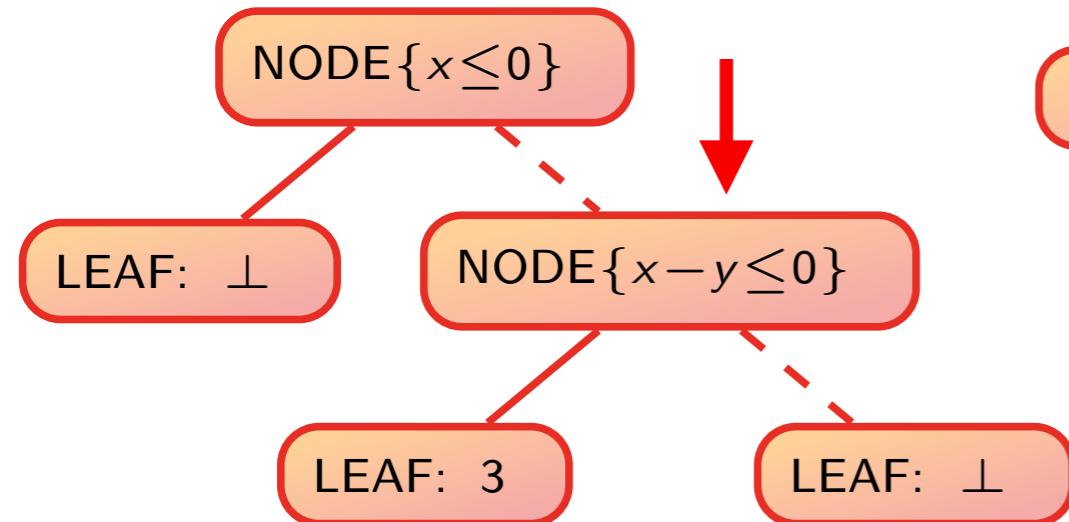
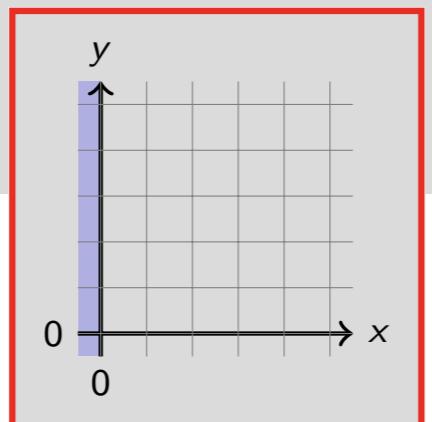
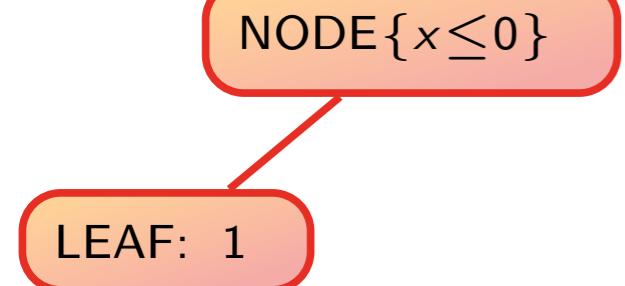
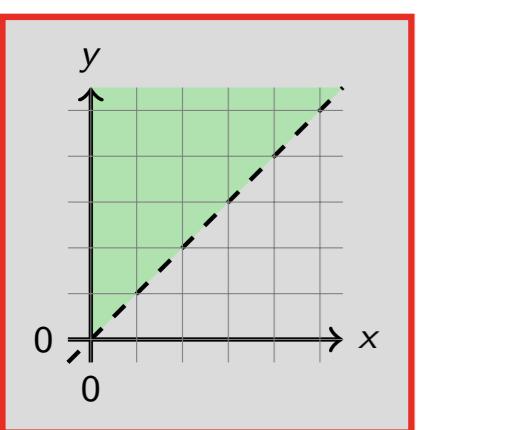
Algorithm 1 : Tree Unification

```

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) 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 ( $\text{NODE}\{t_2.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_{\mathbb{L}} t_2.c$ ) 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 ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_1.c\} : l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_2, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )

```

Join

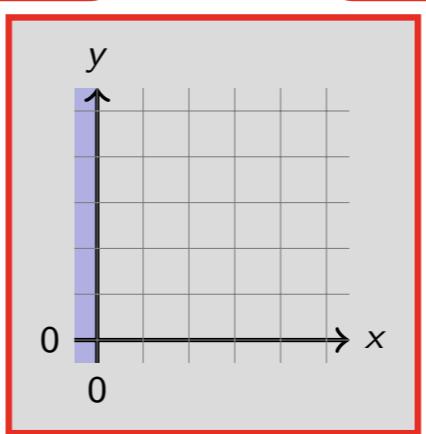
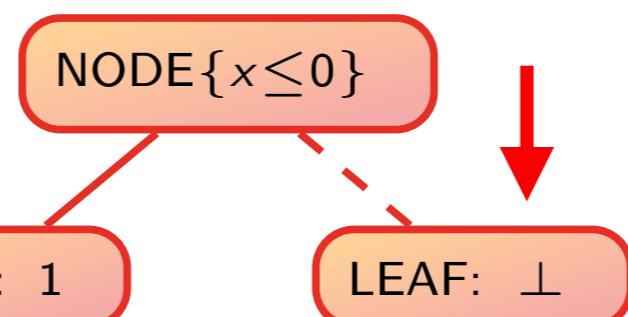


Algorithm 1 : Tree Unification

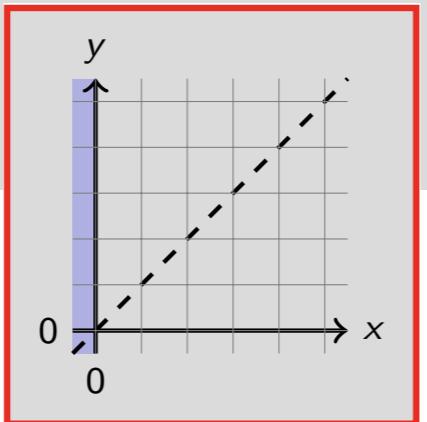
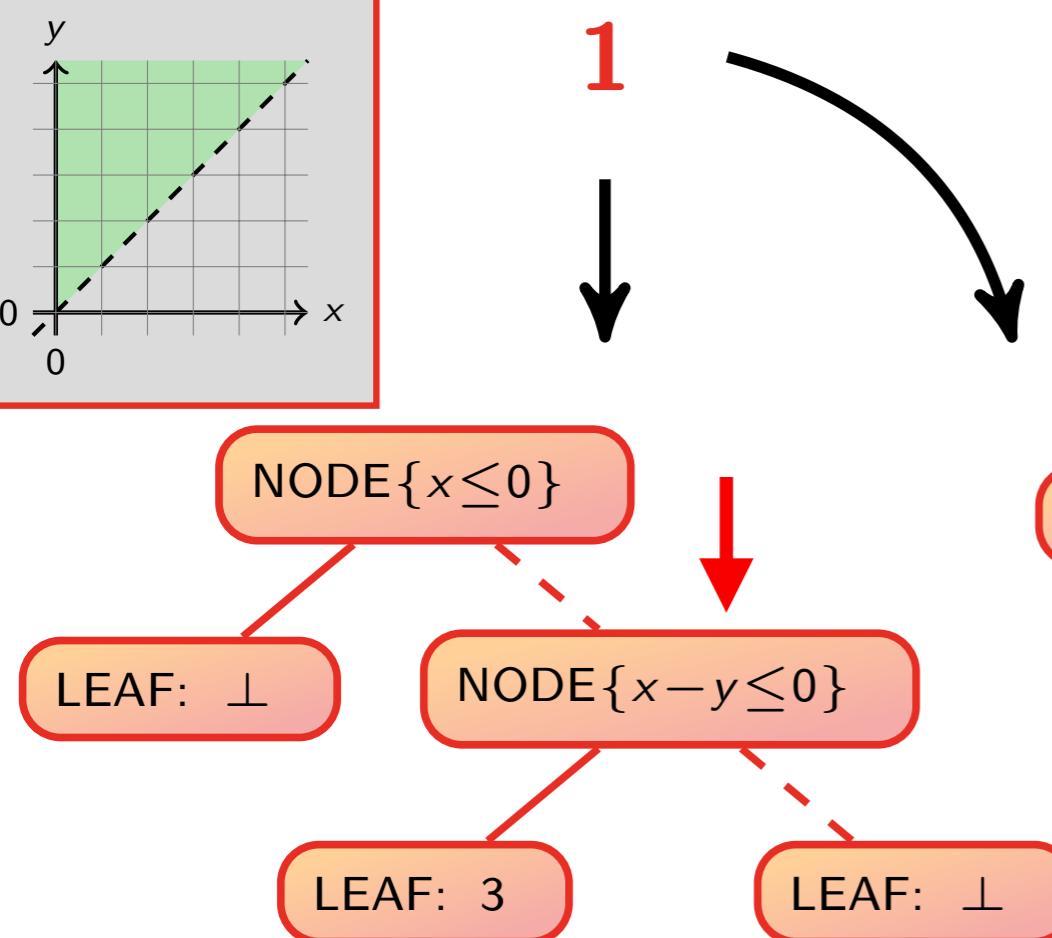
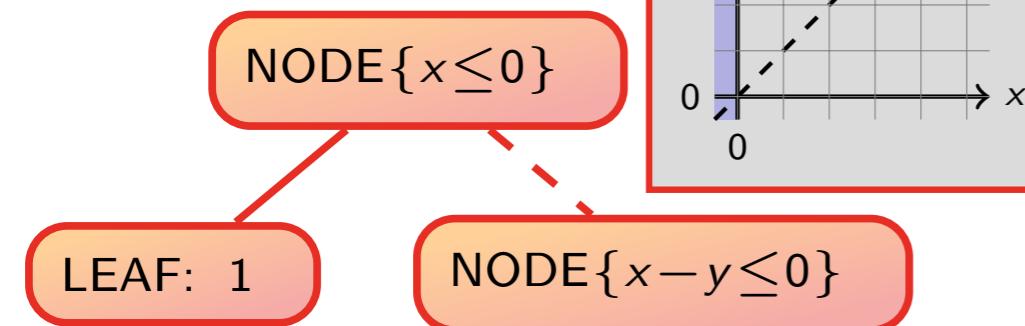
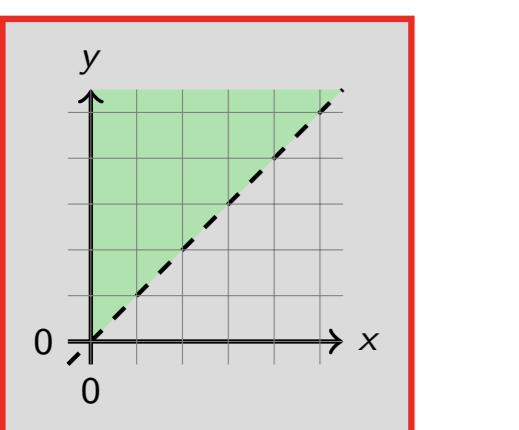
```

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( $t_2$ )  $\wedge$   $t_2.c <_{\perp} t_1.c$ ) 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( $t_2$ )  $\wedge$   $t_1.c <_{\perp} t_2.c$ ) 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:     ( $r_2, r_2$ )  $\leftarrow$  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

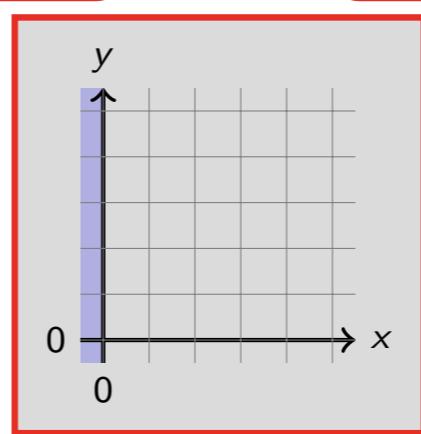
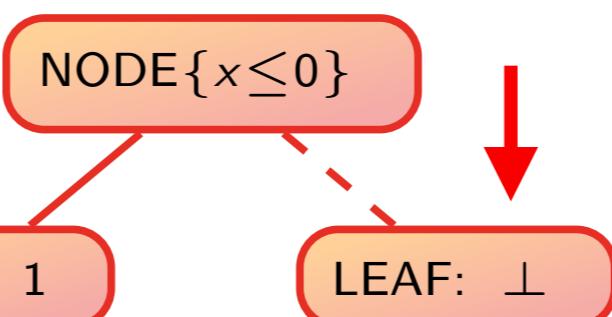


Algorithm 1 : Tree Unification

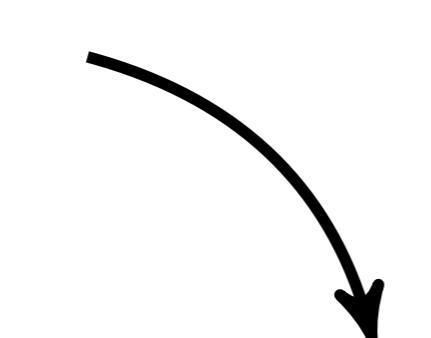
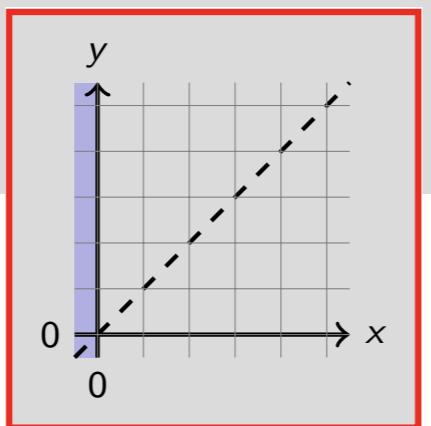
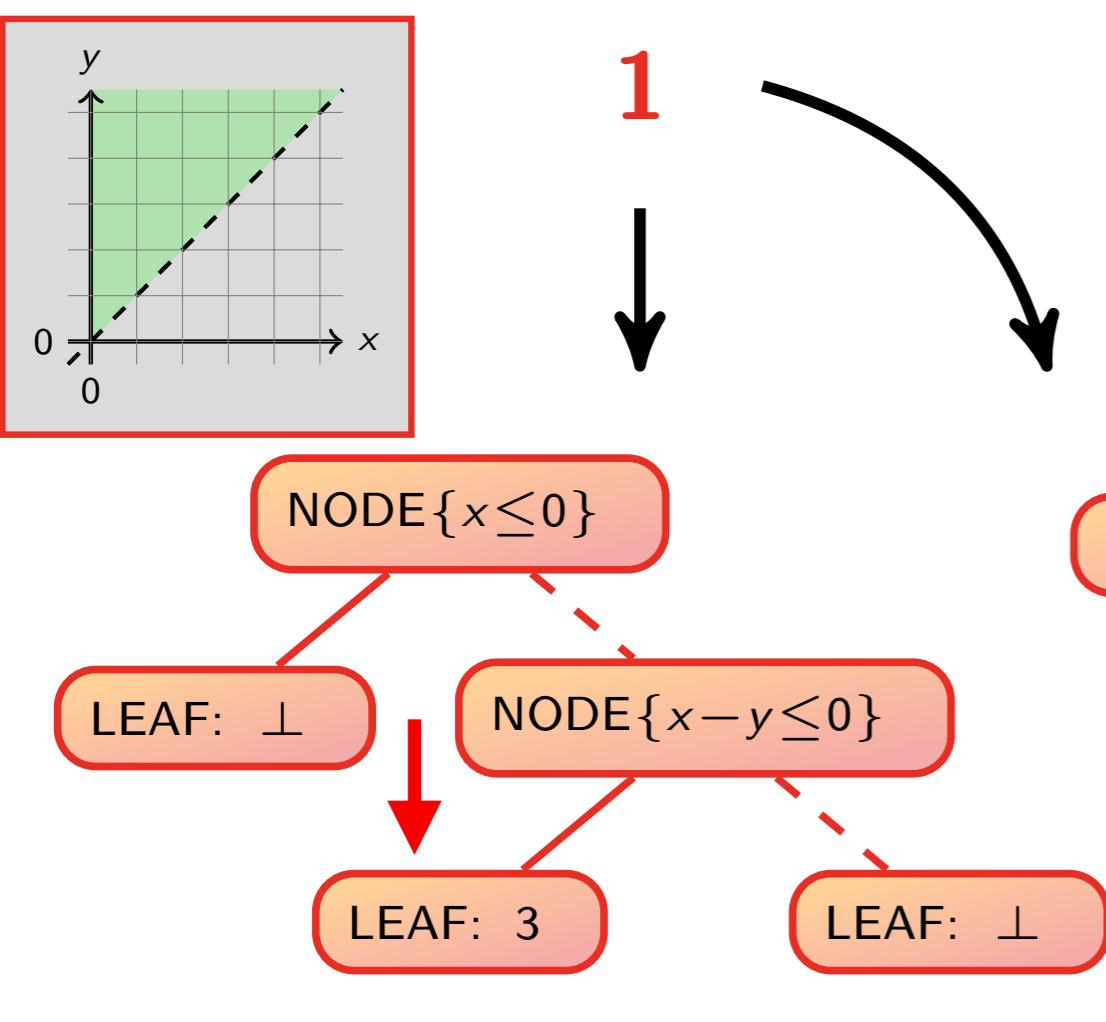
```

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:     ( $r_2, r_2$ )  $\leftarrow$  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

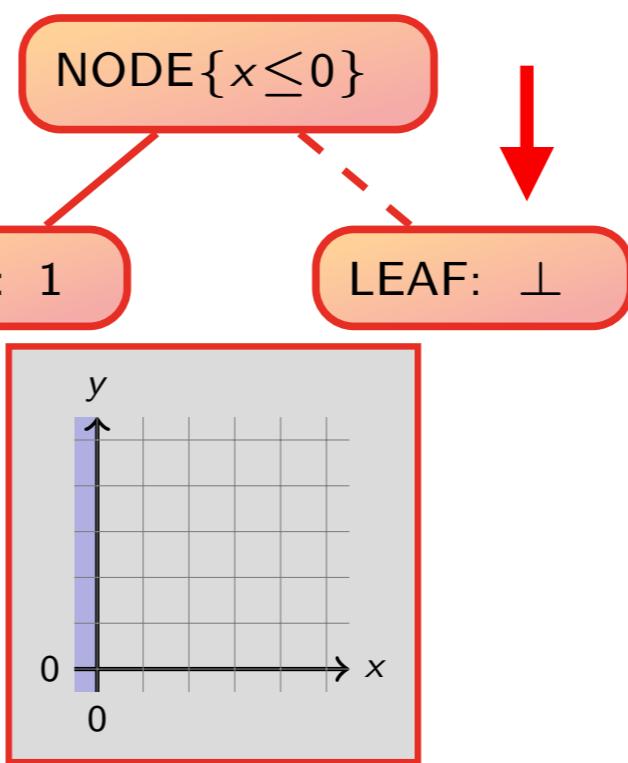


Algorithm 1 : Tree Unification

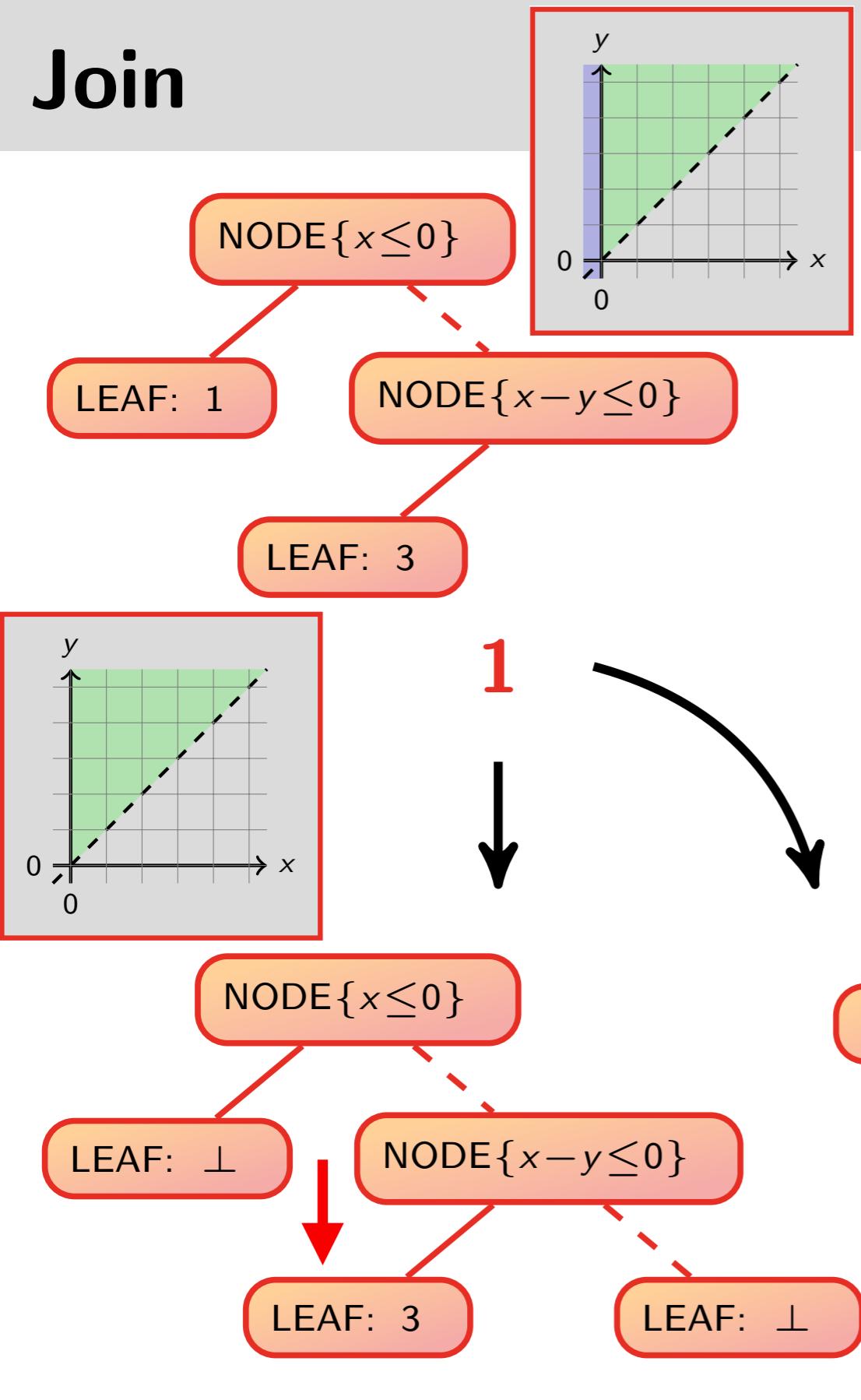
```

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) 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 ( $\text{NODE}\{t_2.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_{\mathbb{L}} t_2.c$ ) 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 ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_1.c\} : l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_2, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )

```



Join

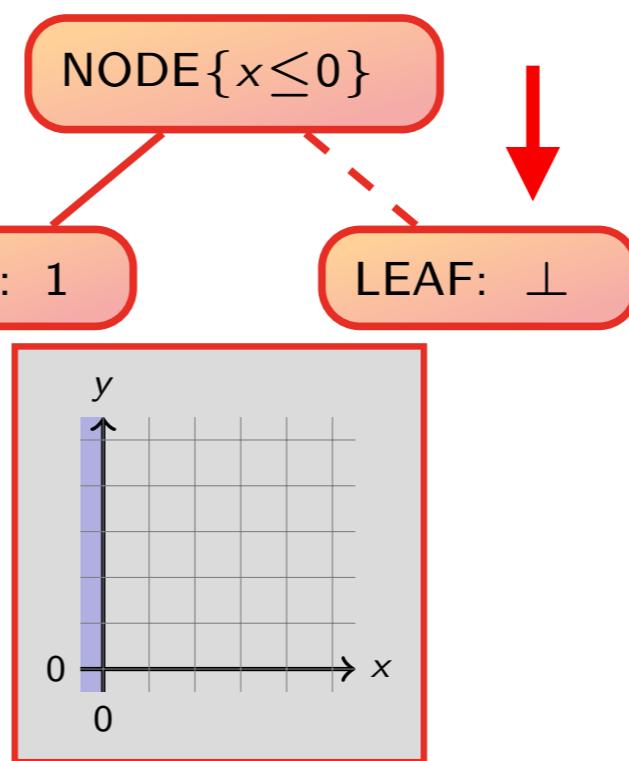


Algorithm 1 : Tree Unification

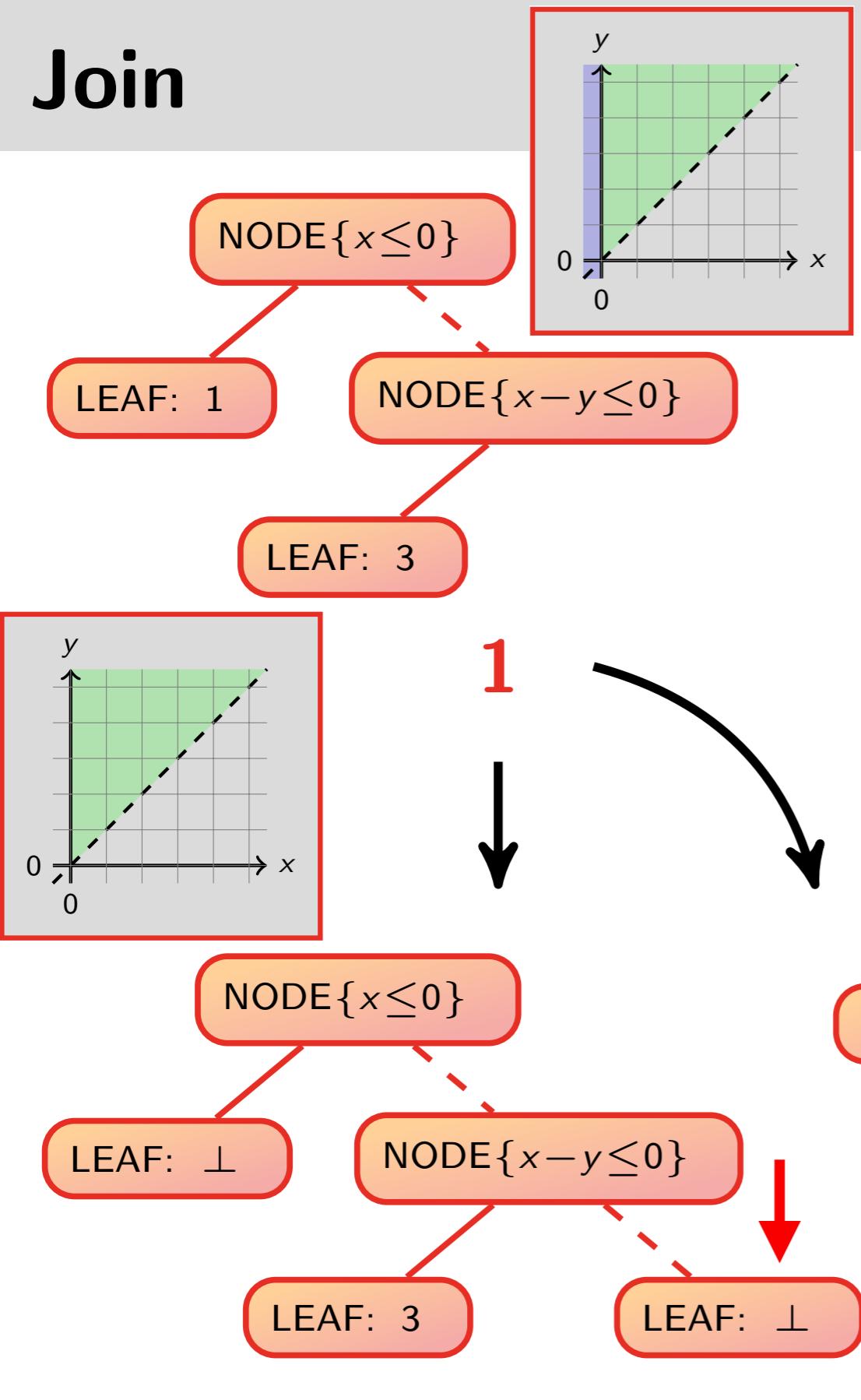
```

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( $t_2$ )  $\wedge$   $t_2.c <_{\perp} t_1.c$ ) 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( $t_2$ )  $\wedge$   $t_1.c <_{\perp} t_2.c$ ) 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:     ( $r_2, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return (NODE $\{t_1.c\} : l_1; r_1$ , NODE $\{t_2.c\} : l_2; r_2$ )

```



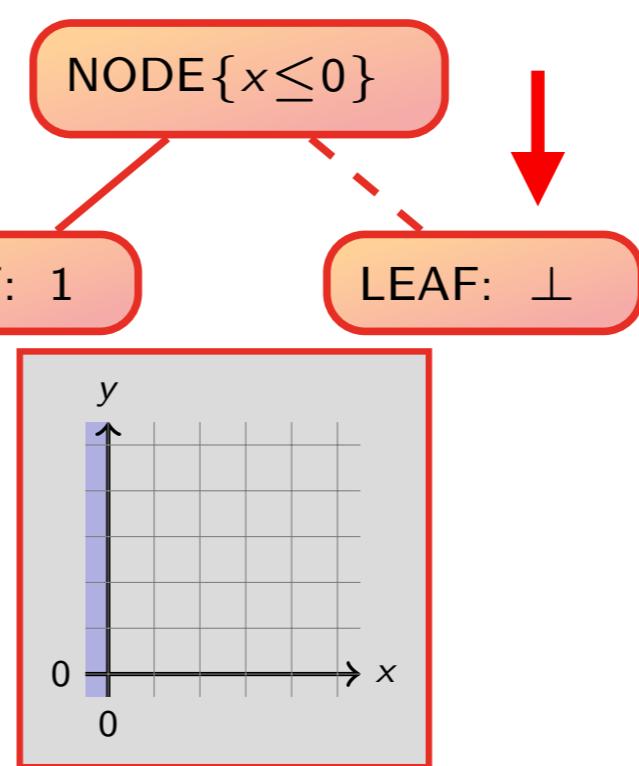
Join



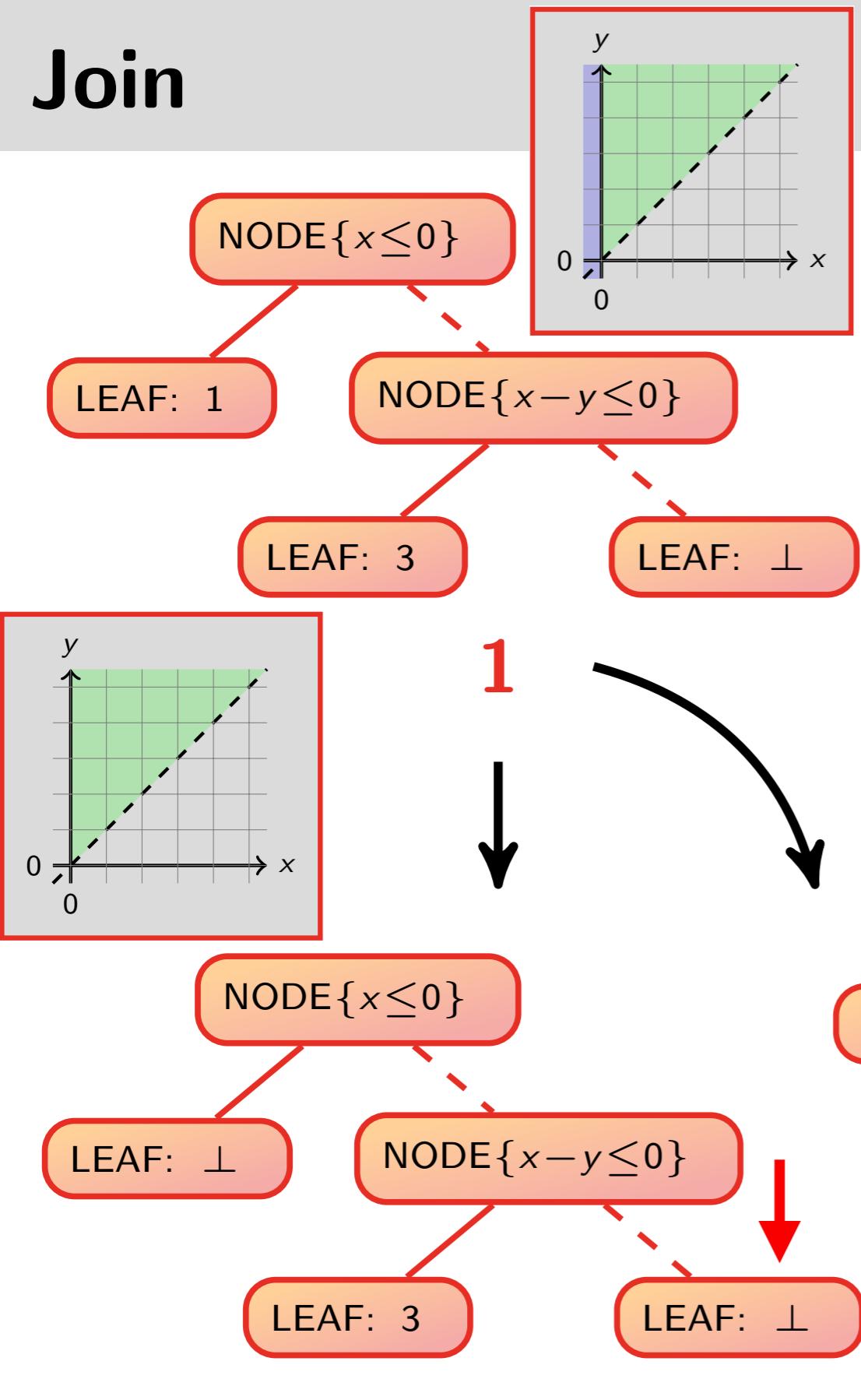
Algorithm 1 : Tree Unification

```

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( $t_2$ )  $\wedge t_2.c <_{\mathbb{L}} t_1.c$ ) 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 ( $\text{NODE}\{t_2.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge t_1.c <_{\mathbb{L}} t_2.c$ ) 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 ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_1.c\} : l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_2, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )
  
```



Join

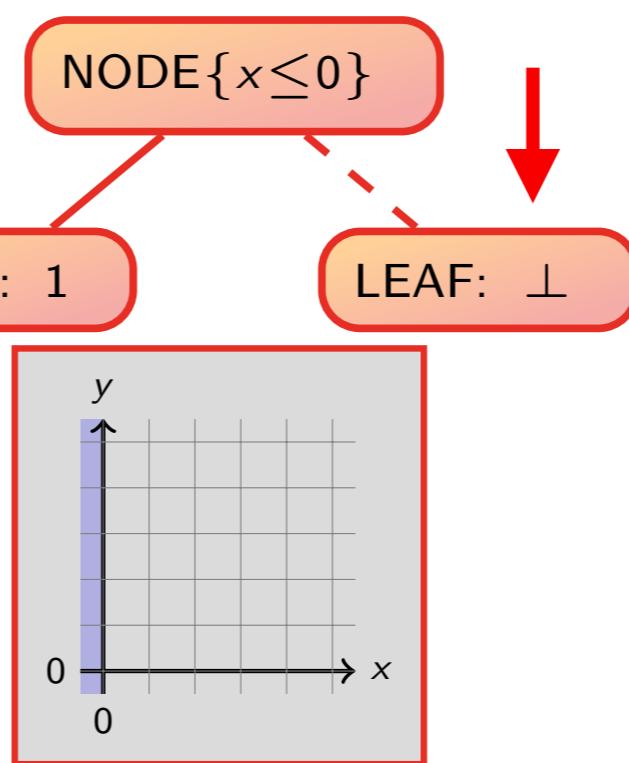


Algorithm 1 : Tree Unification

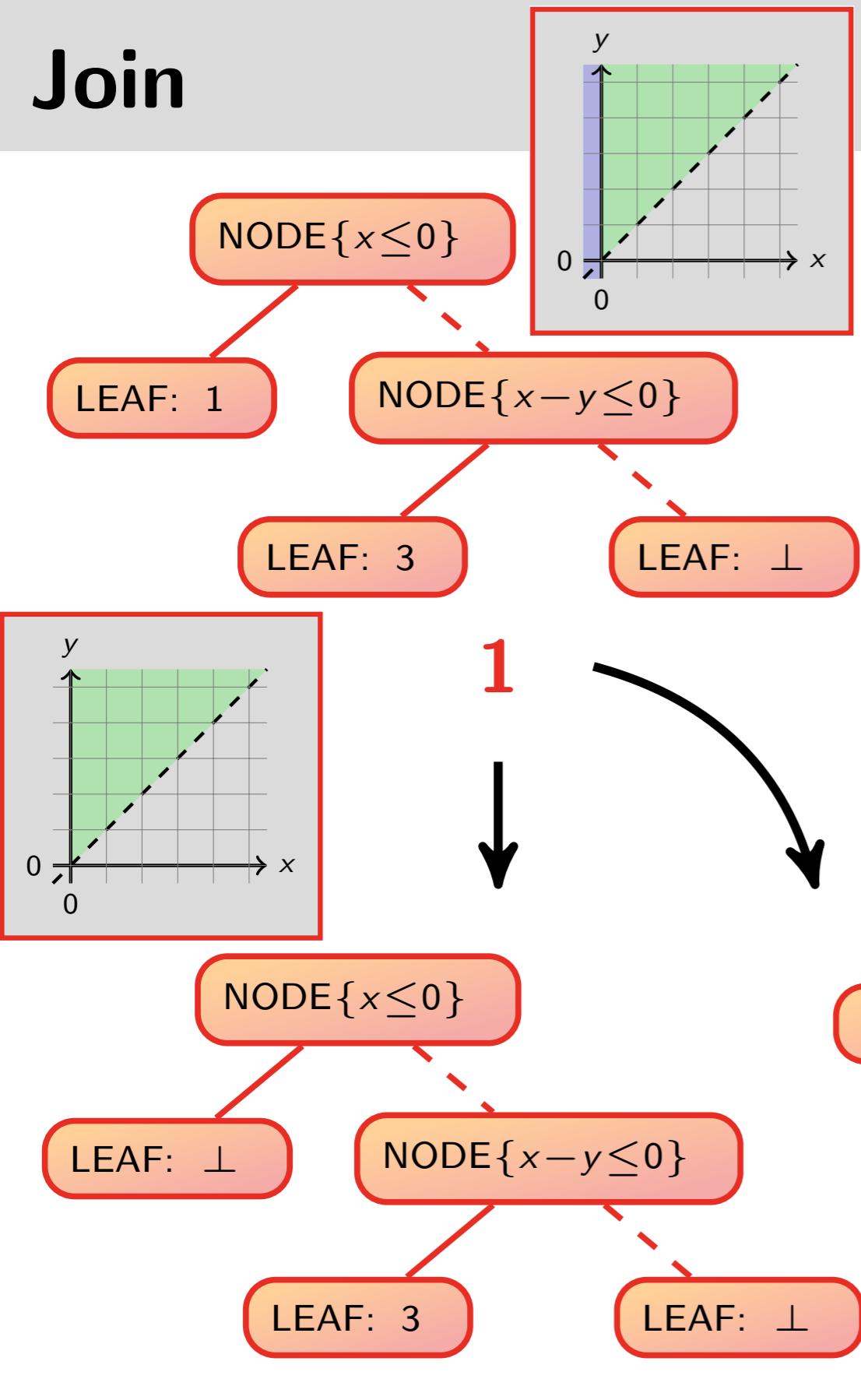
```

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 ( $\text{NODE}\{t_2.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )
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 ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_1.c\} : l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_2, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )

```



Join

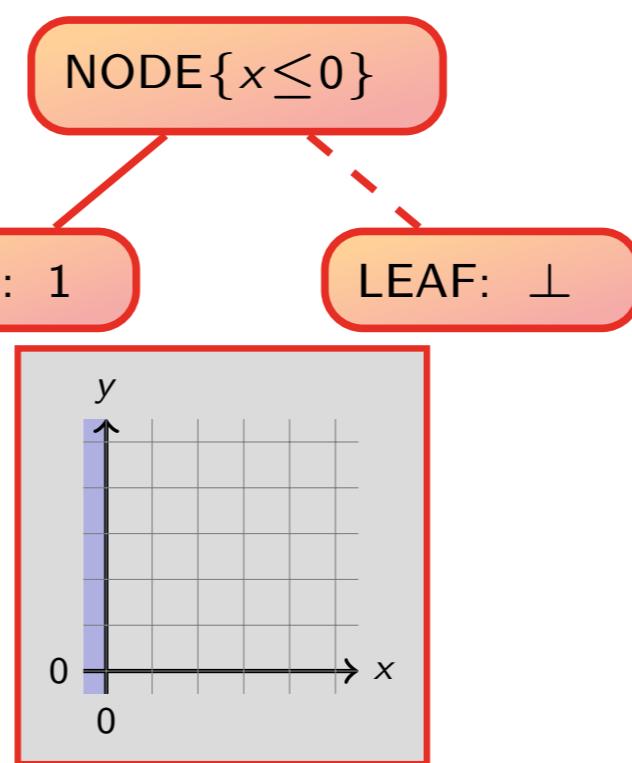


Algorithm 1 : Tree Unification

```

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) 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 ( $\text{NODE}\{t_2.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )
8:   else if ISLEAF( $t_2$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_1.c <_{\mathbb{L}} t_2.c$ ) 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 ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_1.c\} : l_2; r_2$ )
12:   else
13:     ( $l_1, l_2$ )  $\leftarrow$  UNIFICATION( $t_1.l, t_2.l$ )
14:     ( $r_2, r_2$ )  $\leftarrow$  UNIFICATION( $t_1.r, t_2.r$ )
15:     return ( $\text{NODE}\{t_1.c\} : l_1; r_1, \text{NODE}\{t_2.c\} : l_2; r_2$ )

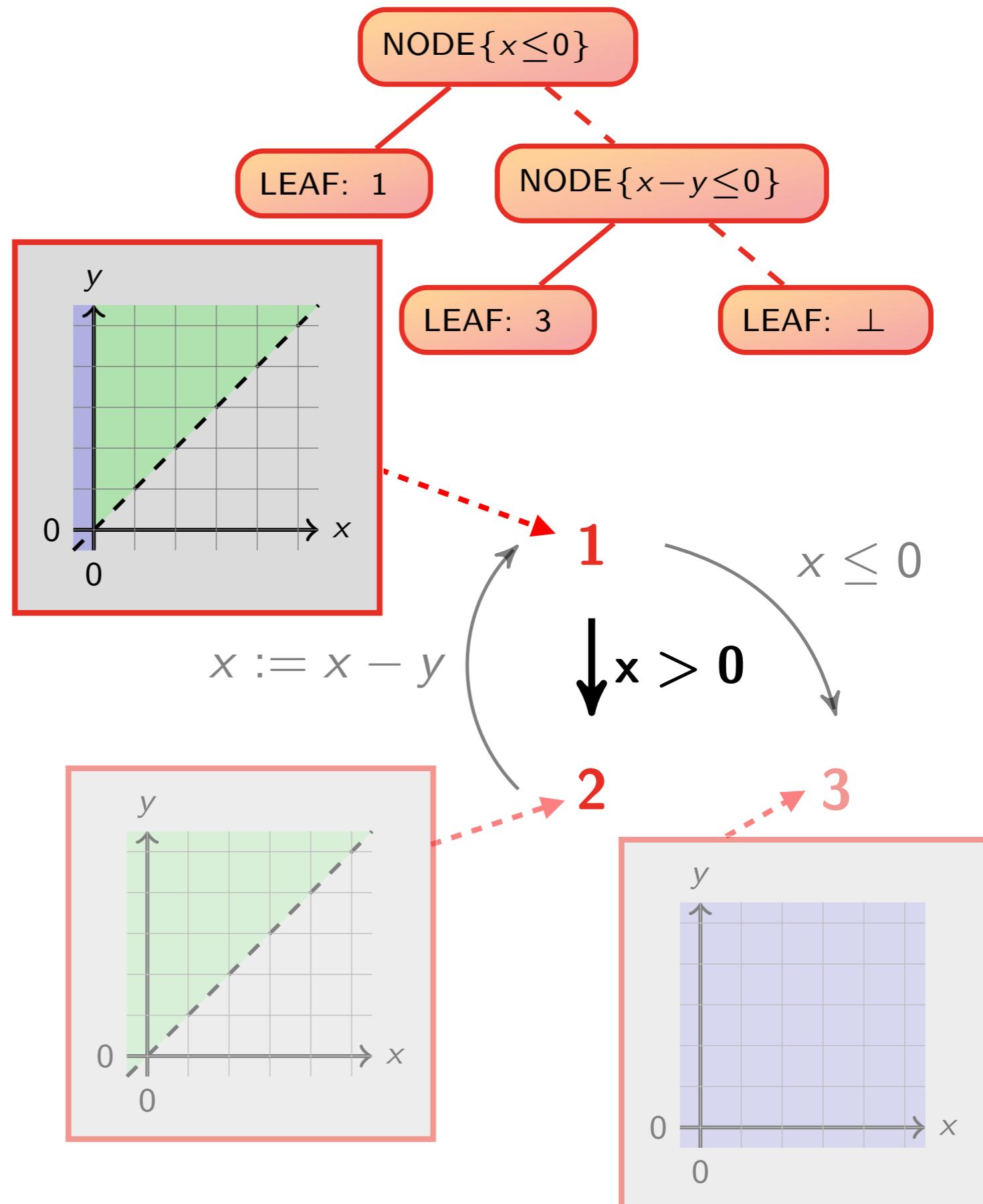
```



Example

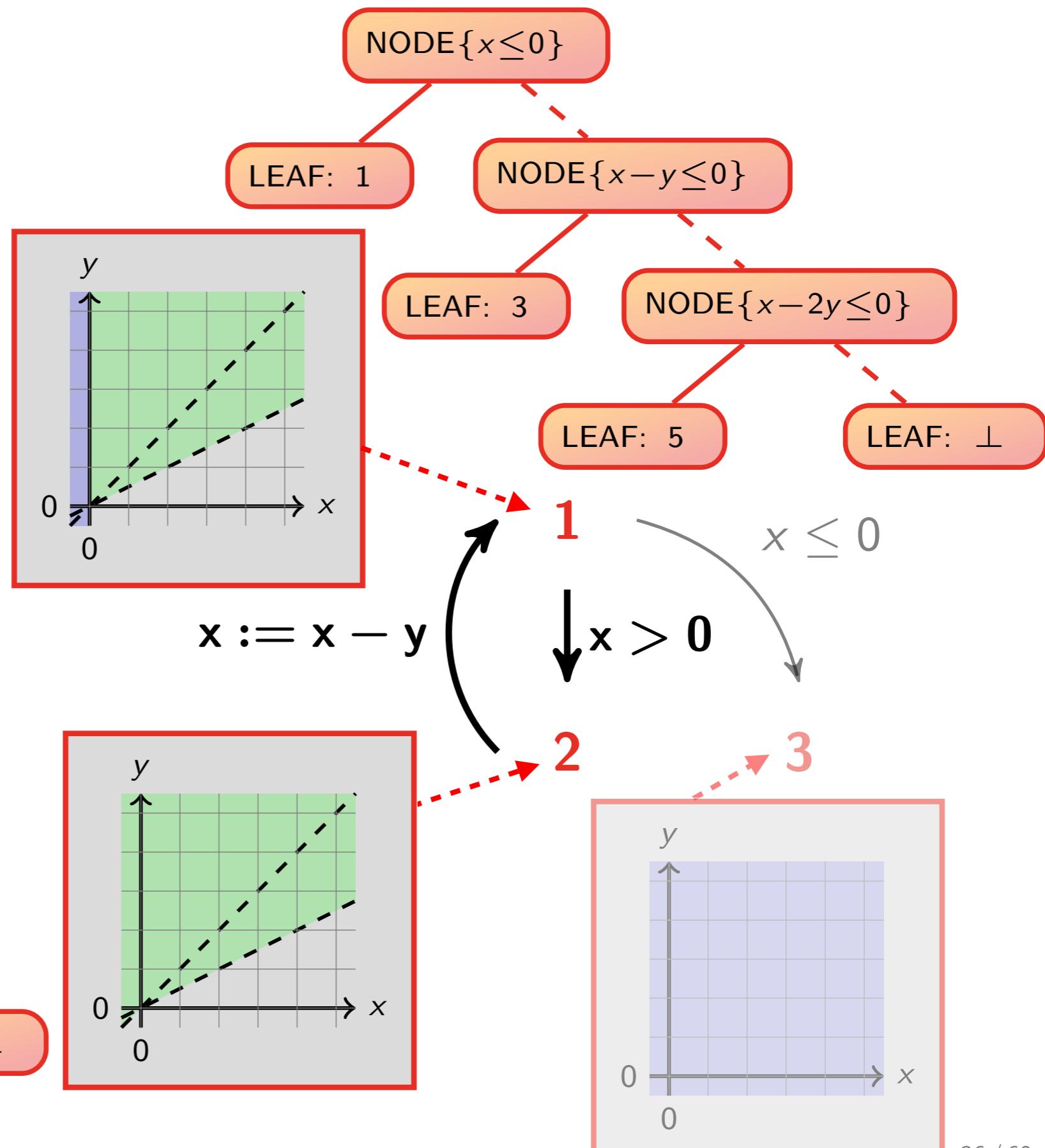
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

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

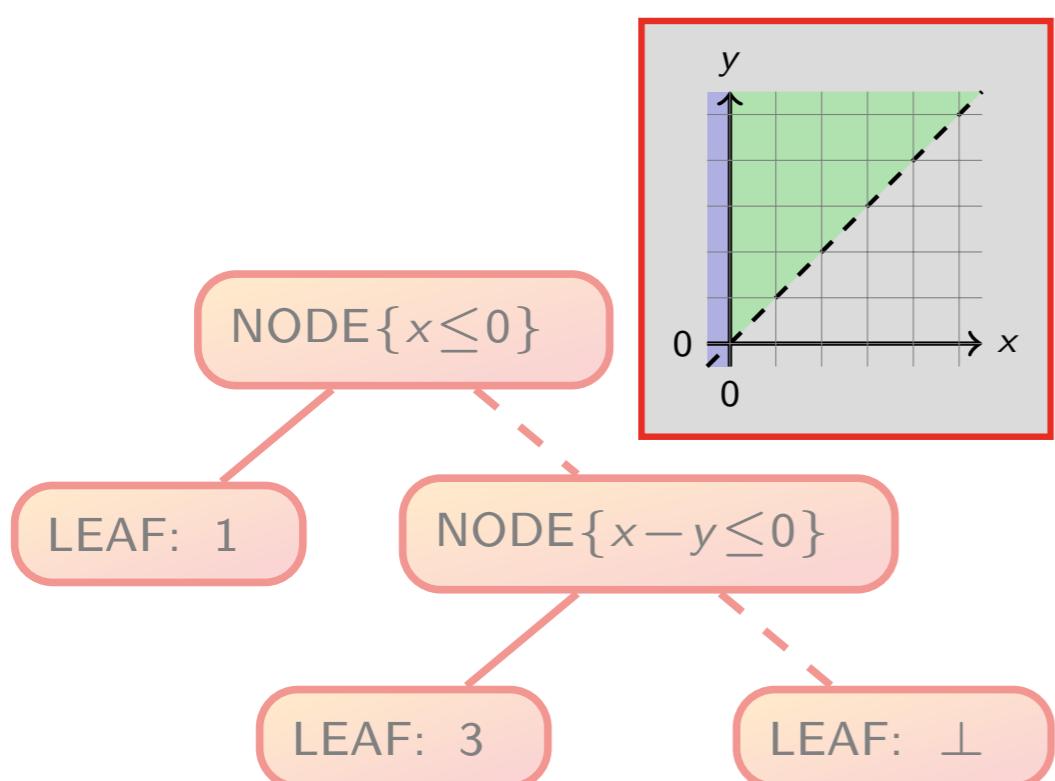


Example

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



Widening: Left Unification

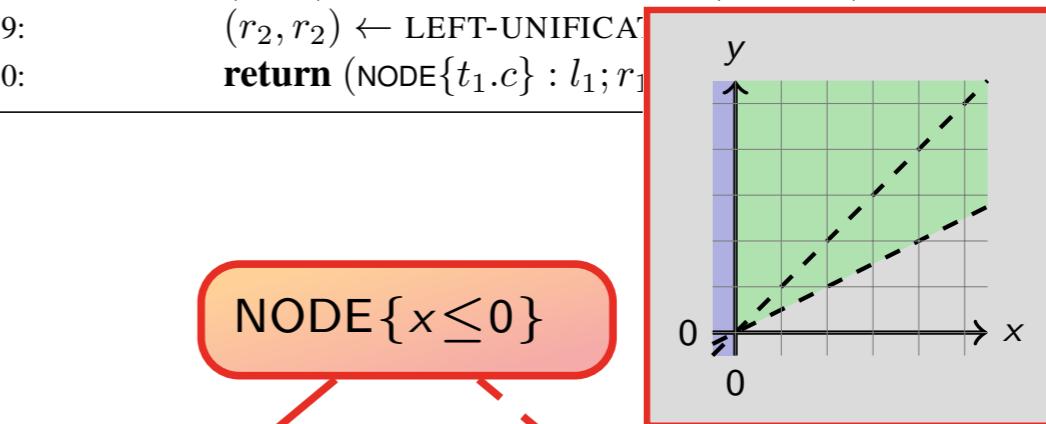


Algorithm 5 : Tree Left Unification

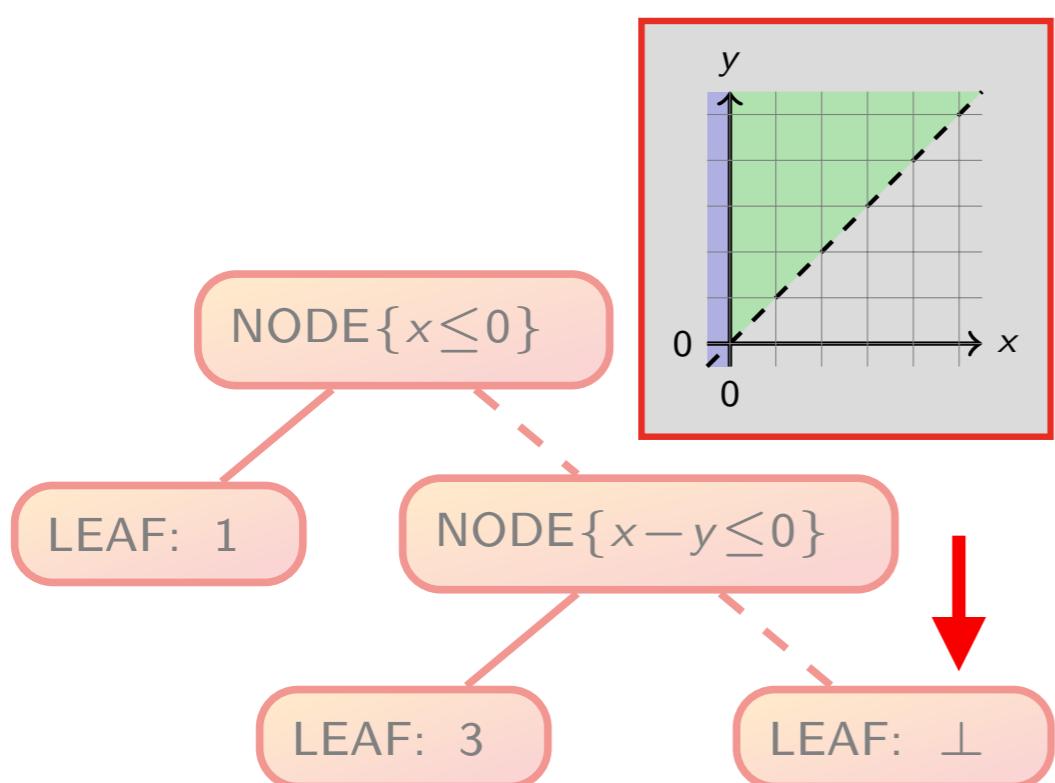
```

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\mathbb{T}} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_2, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:      return (NODE{ $t_1.c$ } :  $l_1; r_2$ )

```



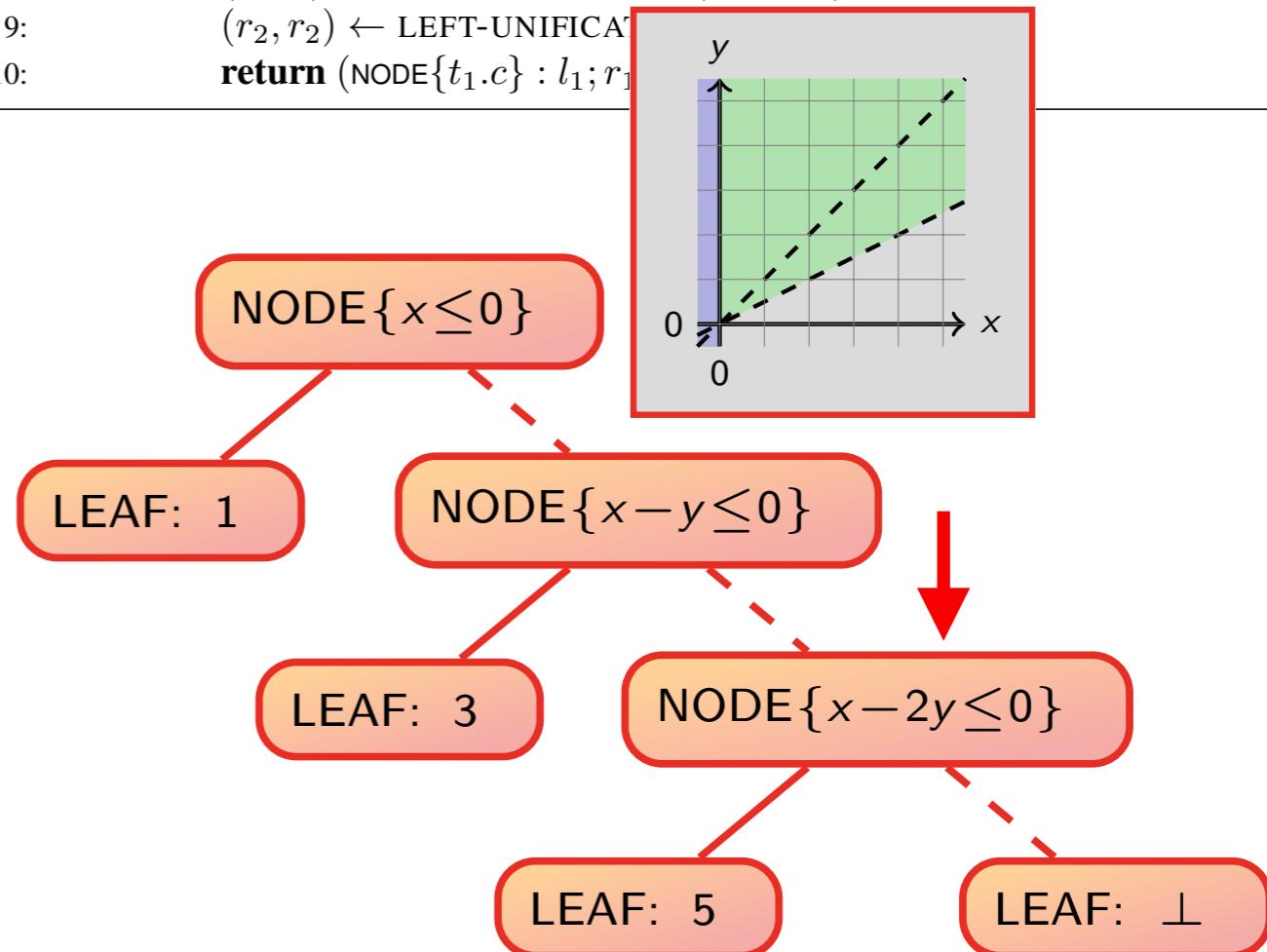
Widening: Left Unification



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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\mathbb{T}} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_2, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:      return (NODE $\{t_1.c\} : l_1 ; r_2$ )
  
```

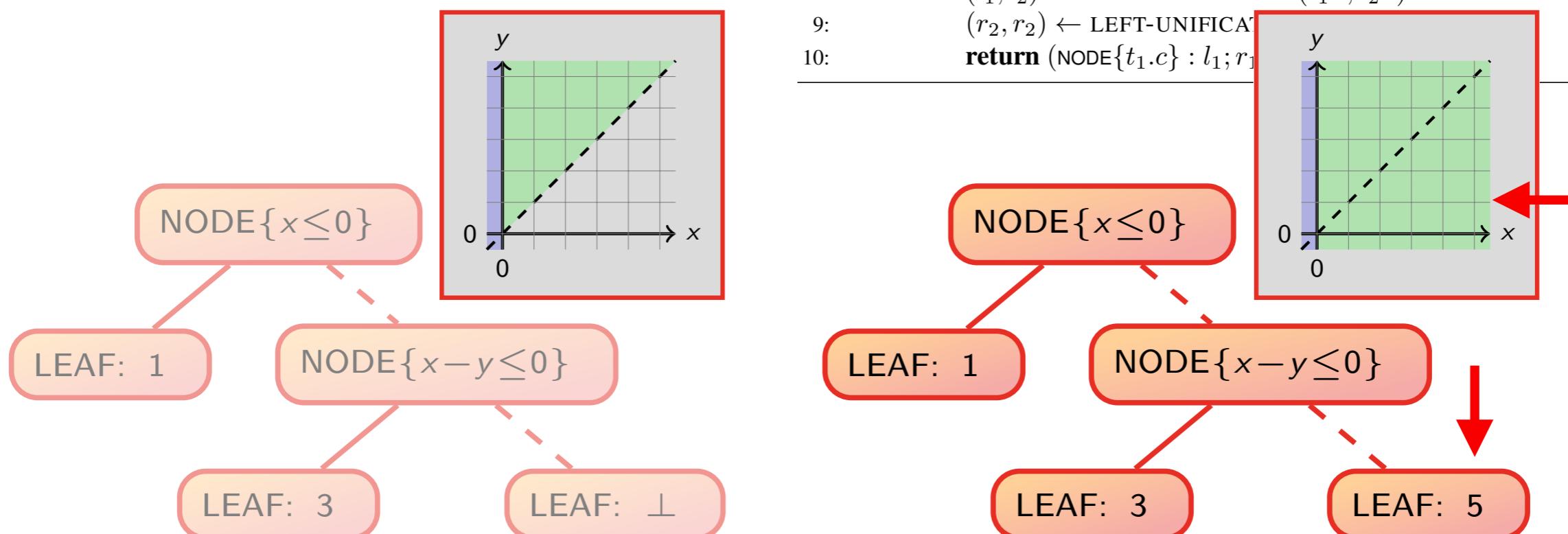


Widening: Left Unification

Algorithm 5 : Tree Left Unification

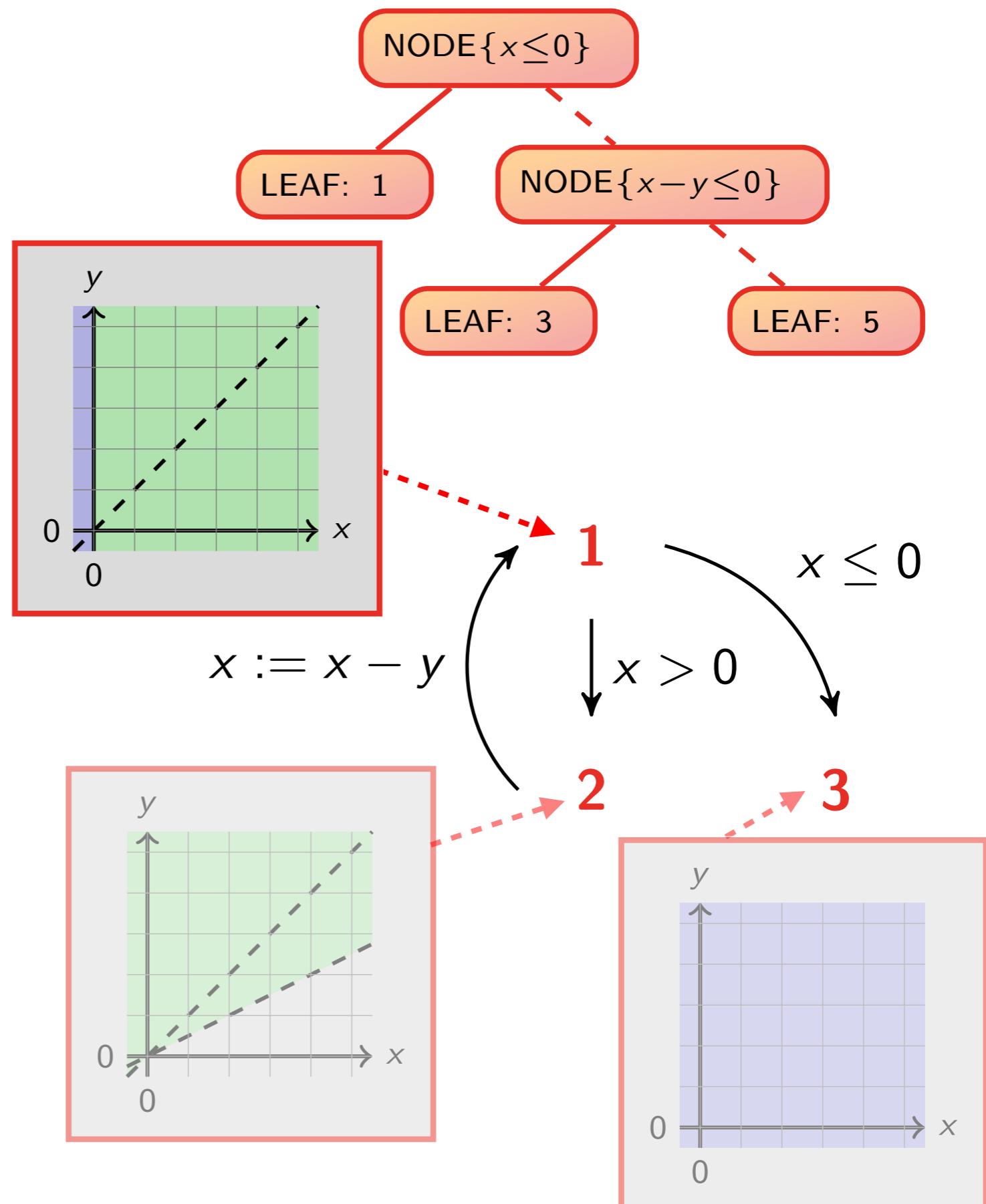
```

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\mathbb{T}} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_2, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:      return (NODE{ $t_1.c$ } :  $l_1; r_2$ )
  
```



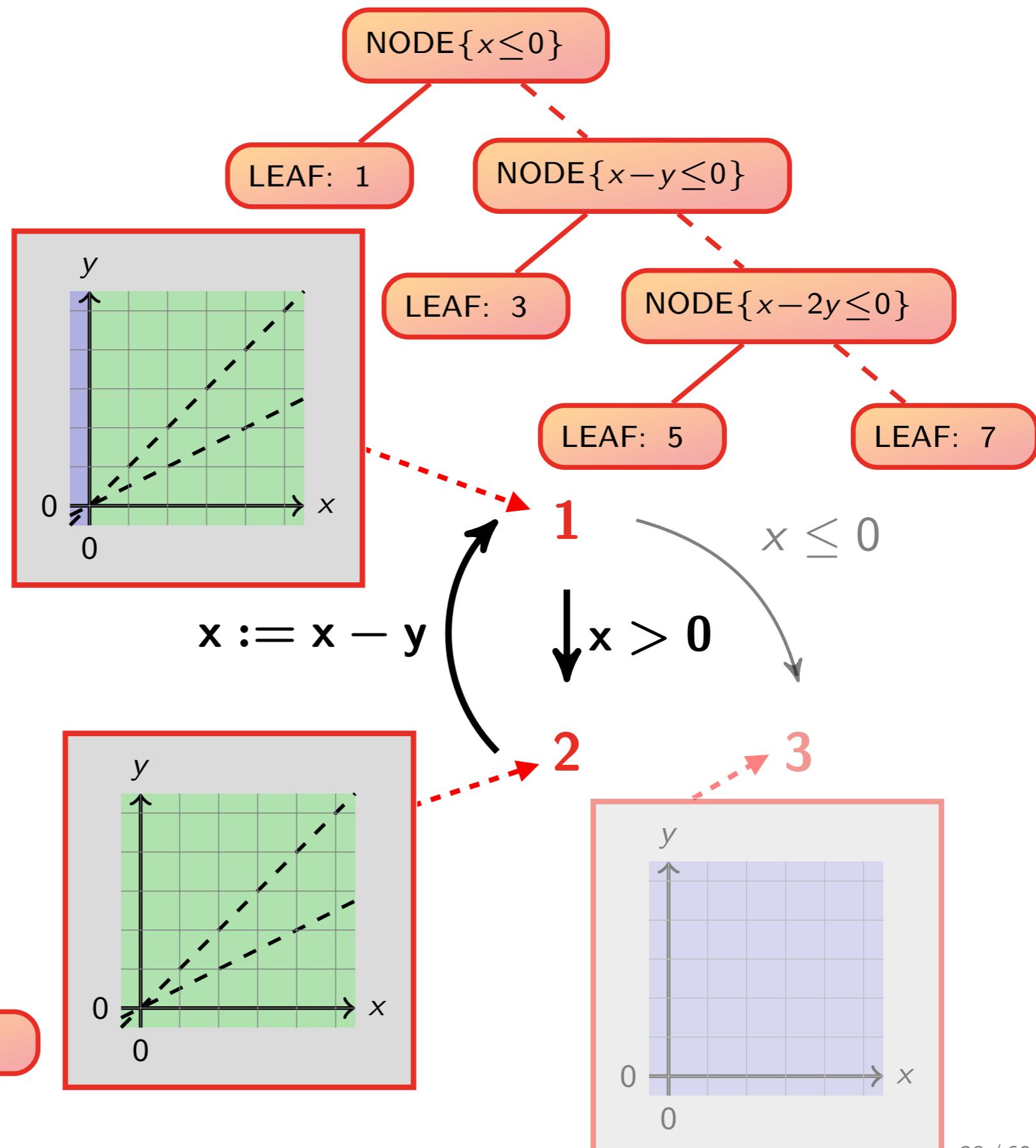
Example

```
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
```

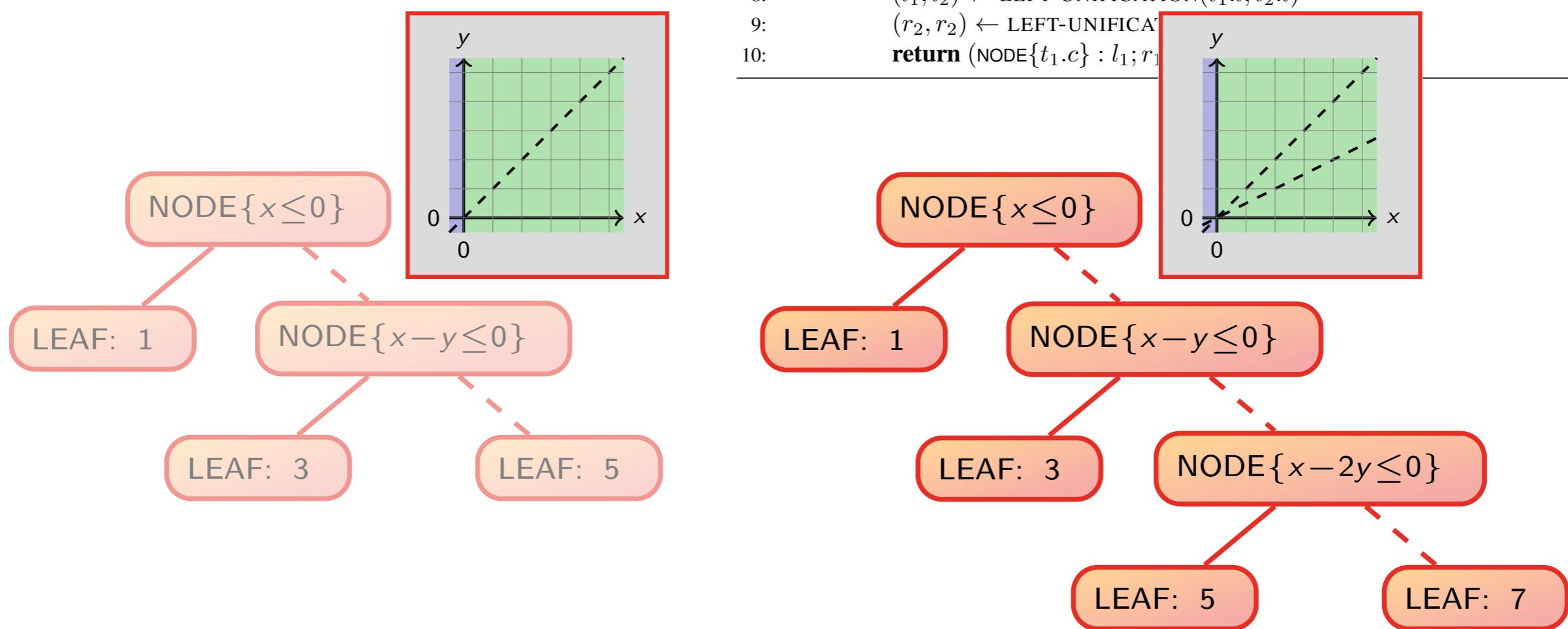


Widening: Domain Over-Approximation

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\mathbb{T}} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_2, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:      return (NODE{ $t_1.c$ } :  $l_1; r_2$ )
  
```

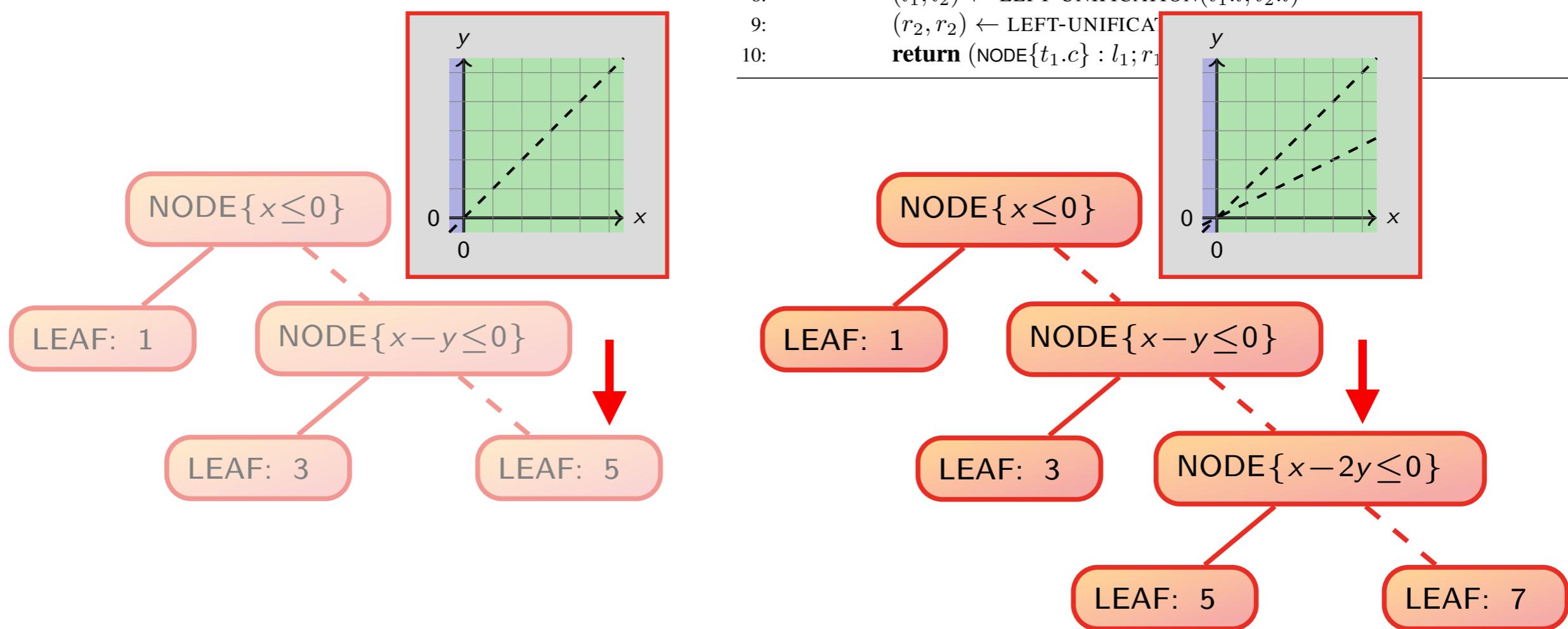


Widening: Domain Over-Approximation

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\mathbb{T}} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_2, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:      return (NODE{ $t_1.c$ } :  $l_1; r_2$ )
  
```

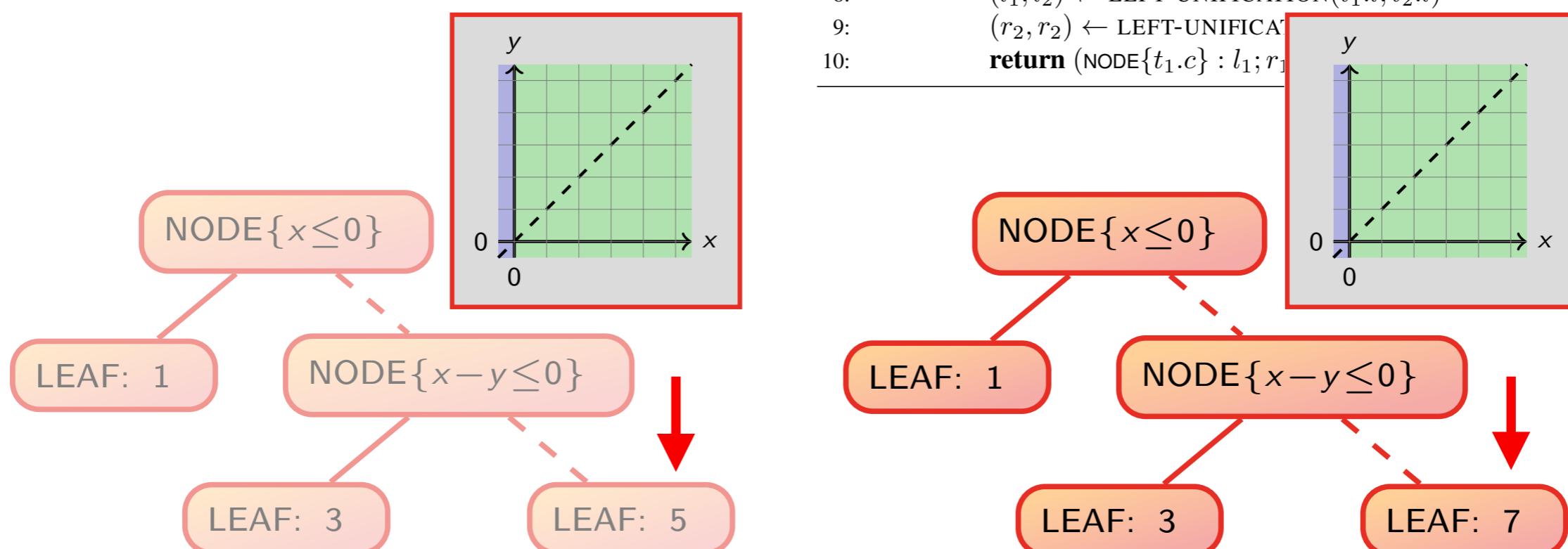


Widening: Domain Over-Approximation

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( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1, t_2.l \sqcup_{\mathbb{T}} t_2.r$ )
7:     else
8:       ( $l_1, l_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.l, t_2.l$ )
9:       ( $r_2, r_2$ )  $\leftarrow$  LEFT-UNIFICATION( $t_1.r, t_2.r$ )
10:      return (NODE{ $t_1.c$ } :  $l_1; r_2$ )
  
```

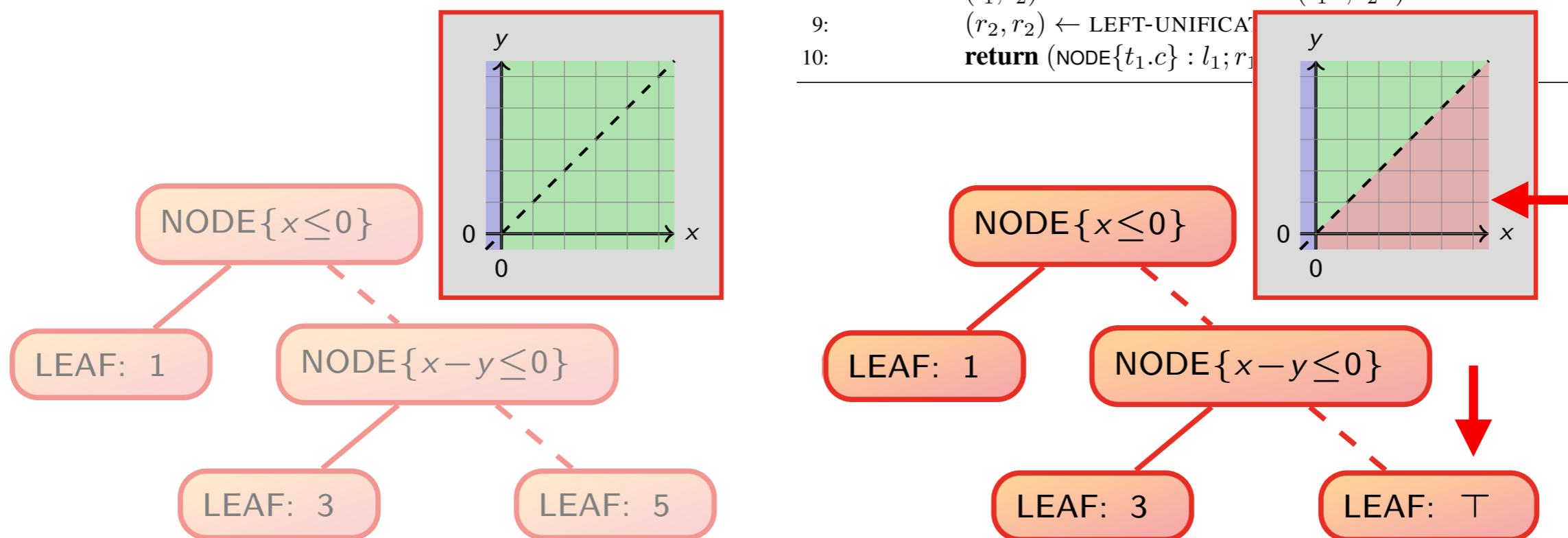


Widening: Domain Over-Approximation

Algorithm 5 : Tree Left Unification

```

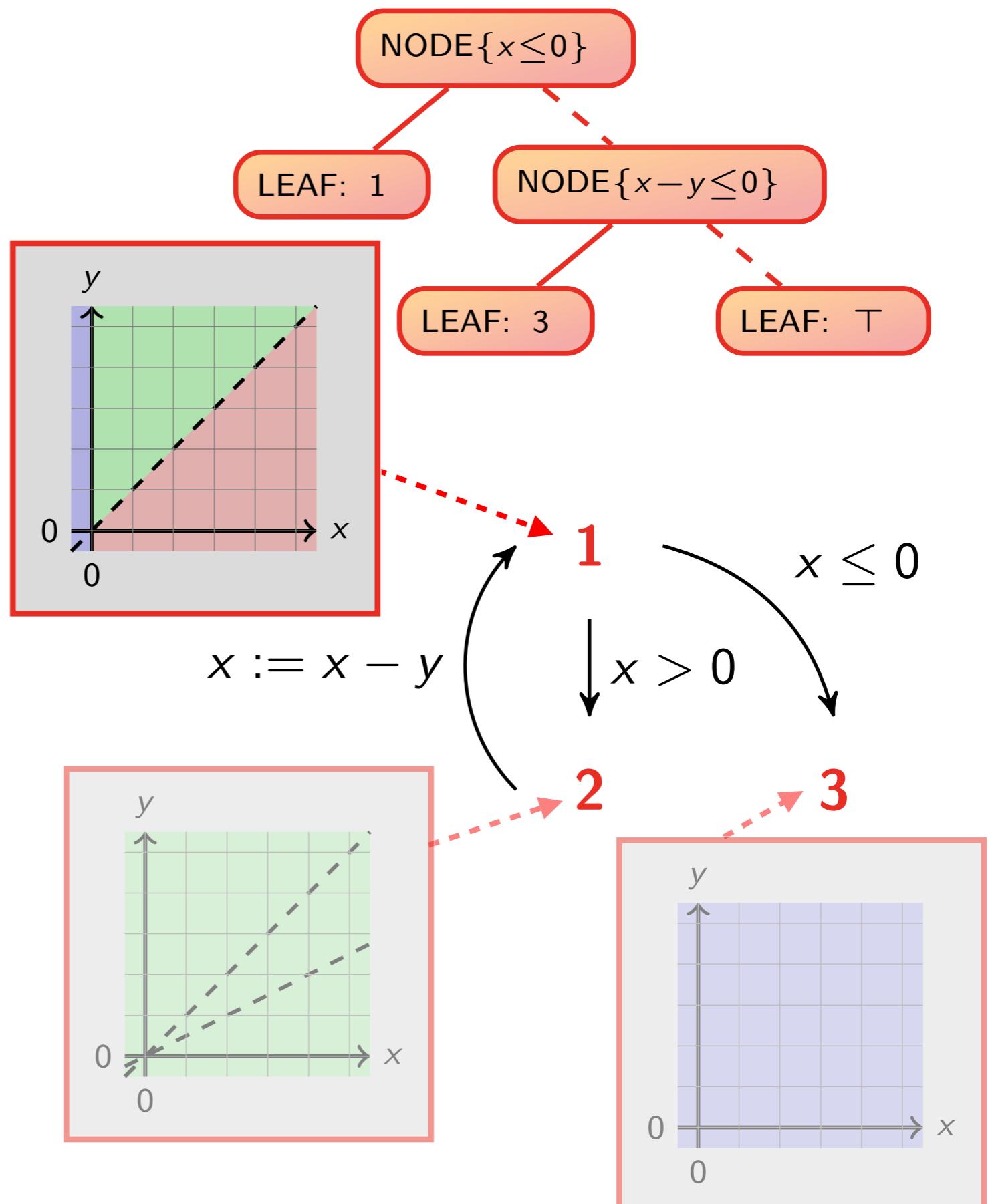
1: function LEFT-UNIFICATION( $t_1, t_2$ )
2:   if ISLEAF( $t_1$ )  $\wedge$  ISLEAF( $t_2$ ) then
3:     return ( $t_1, t_2$ )
4:   else
5:     if ISLEAF( $t_1$ )  $\vee$  (ISNODE( $t_1$ )  $\wedge$  ISNODE( $t_2$ )  $\wedge$   $t_2.c <_{\mathbb{L}} t_1.c$ ) then
6:       return LEFT-UNIFICATION( $t_1,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



Example

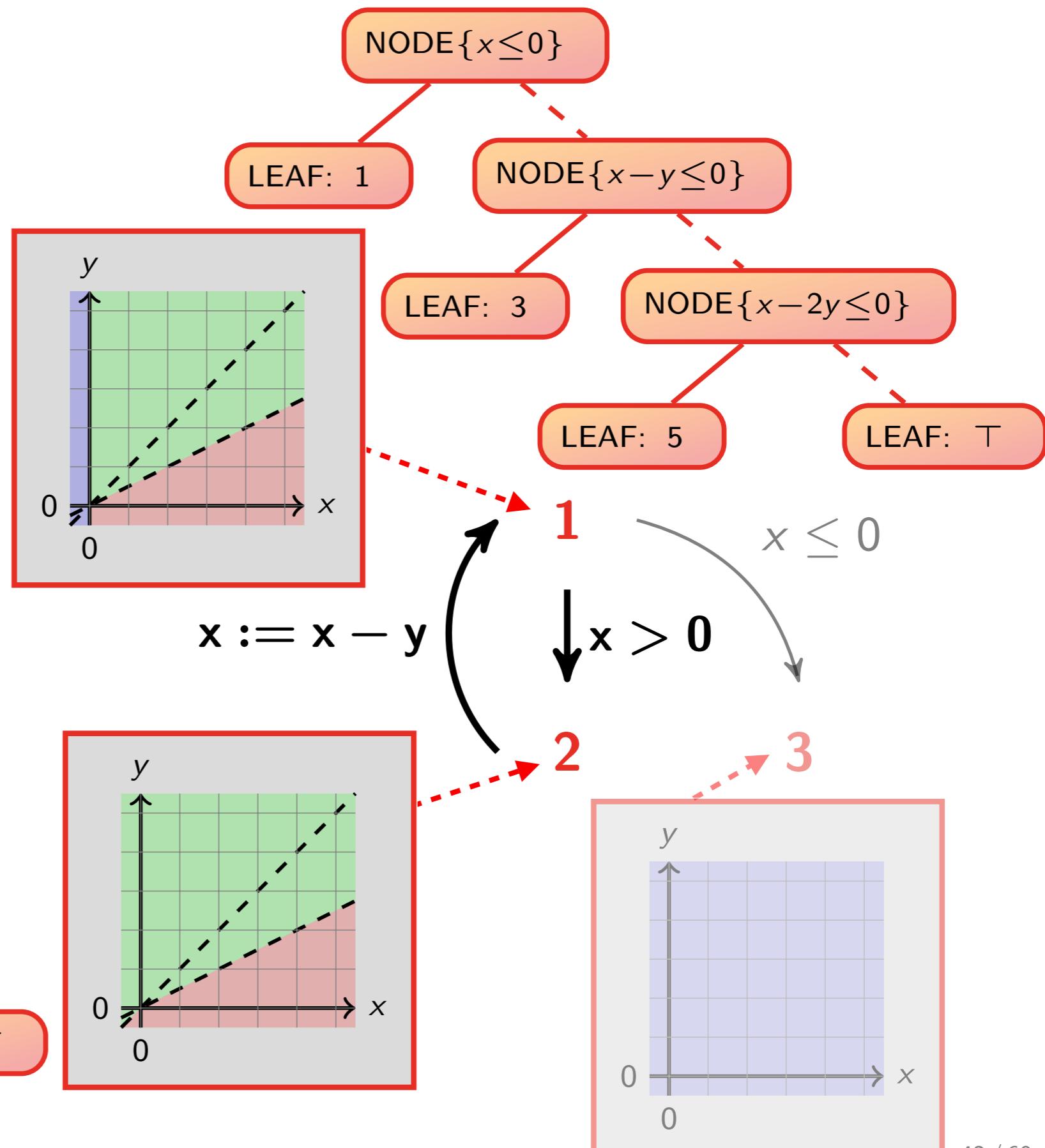
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

→ the widening is **sound!**



Example

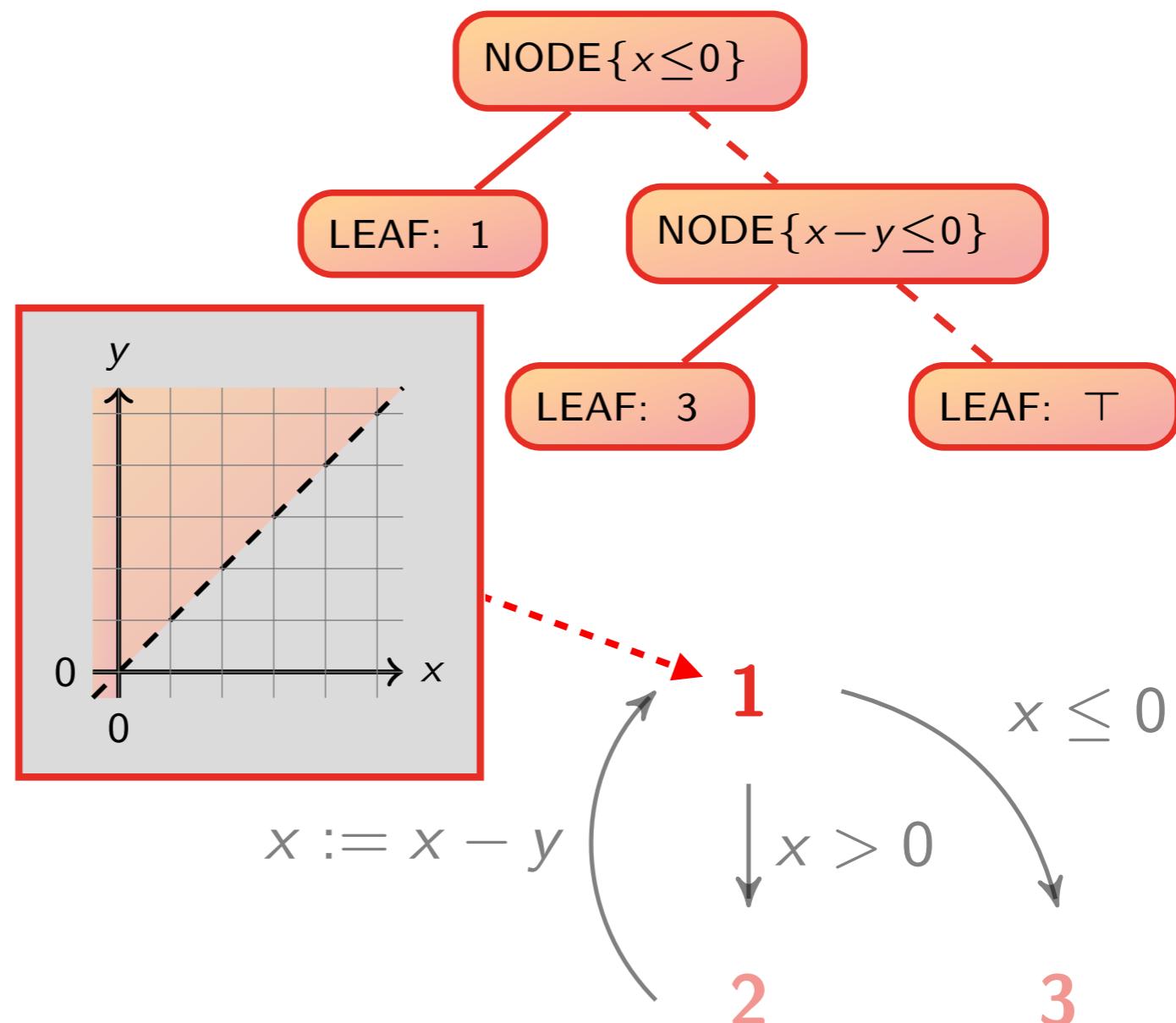
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



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**

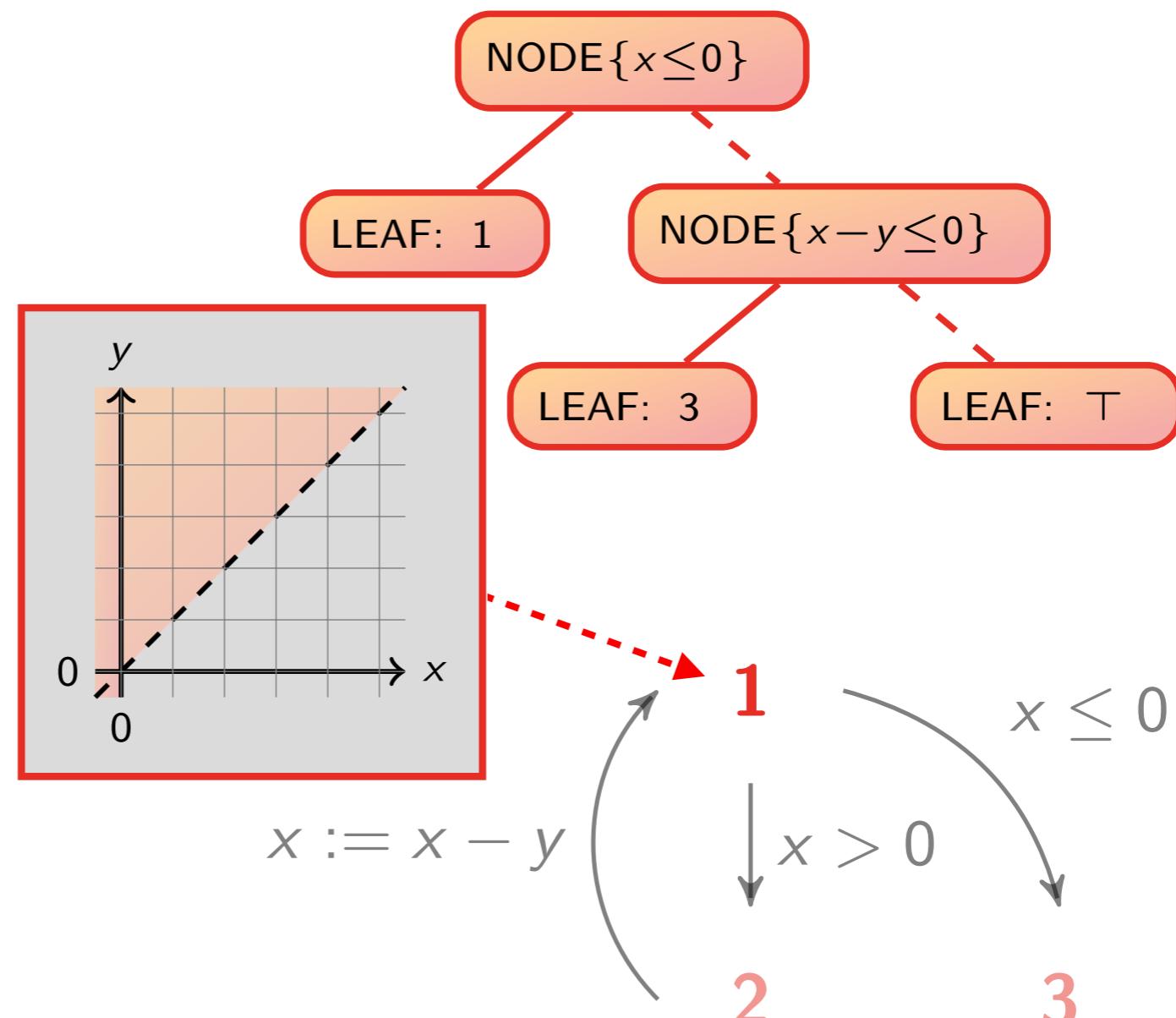


Example

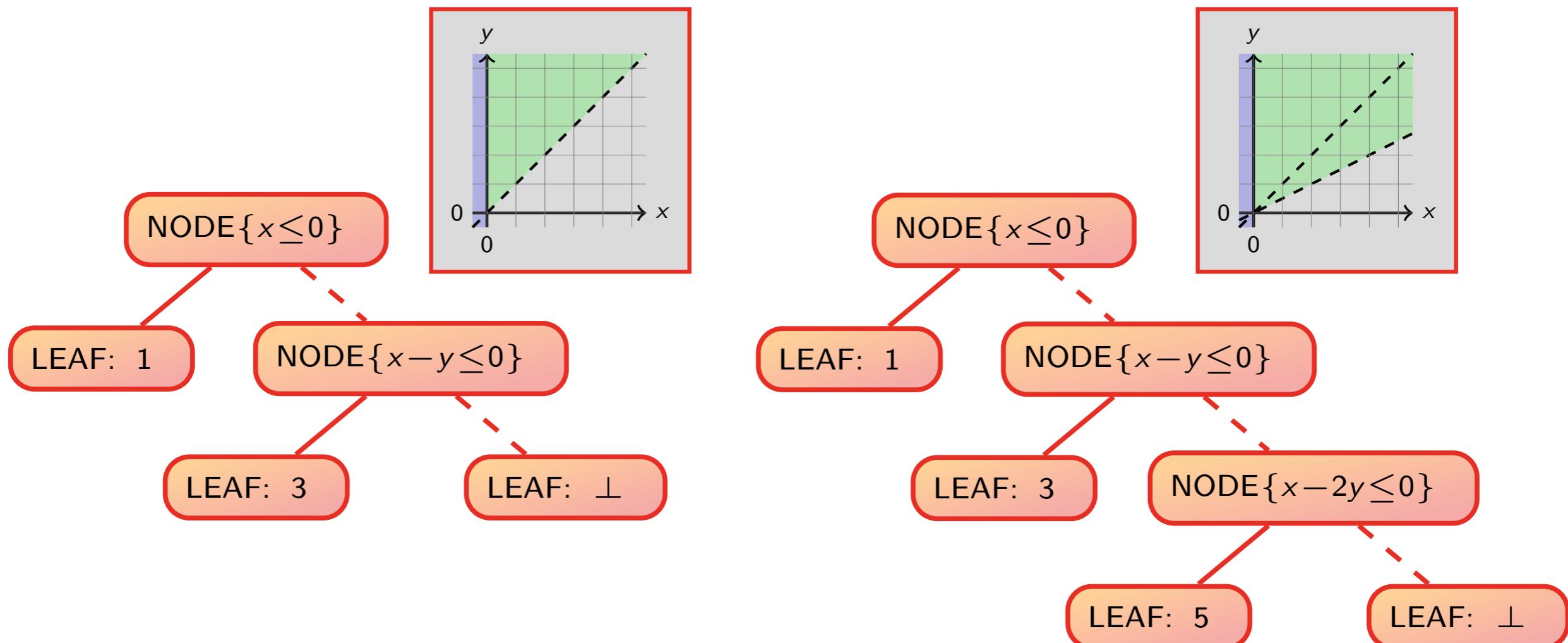
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

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

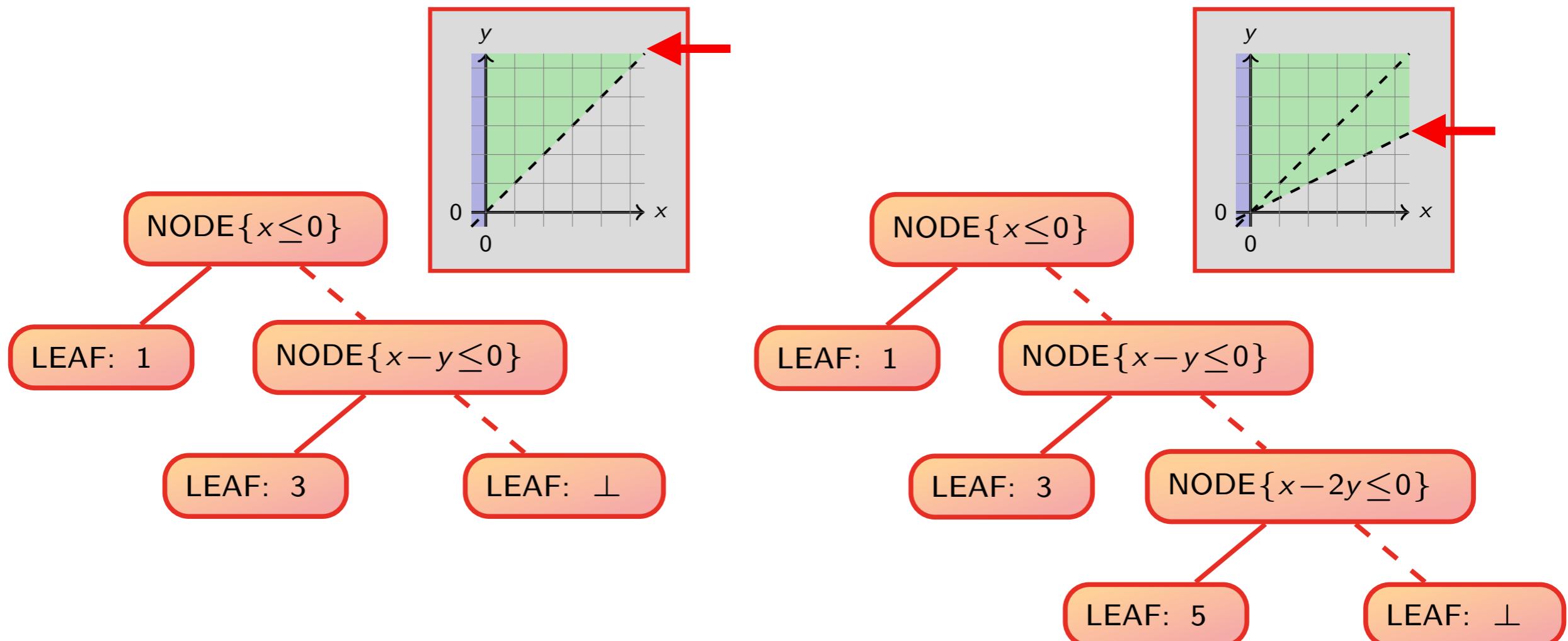
the **weakest precondition**
is $x \leq 0 \vee y > 0$



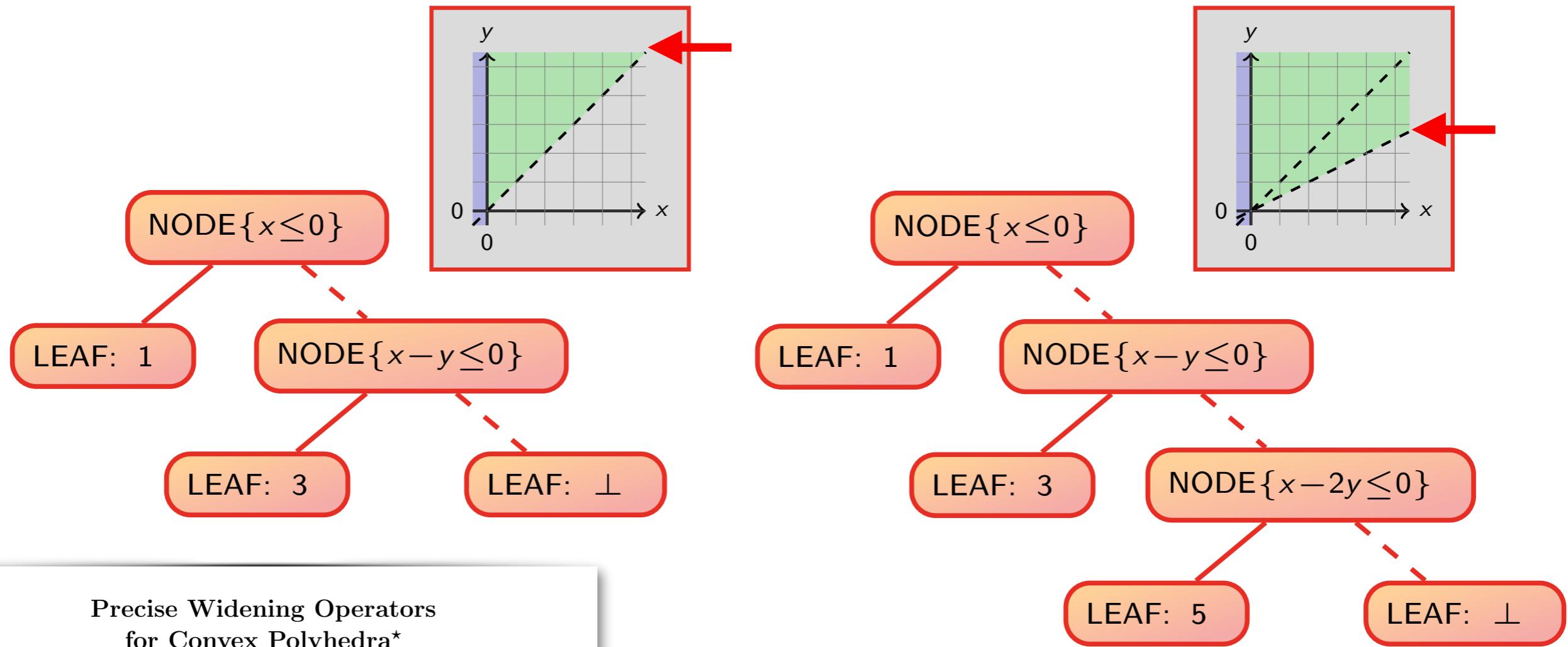
Widening



Widening



Widening



Precise Widening Operators
for Convex Polyhedra*

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

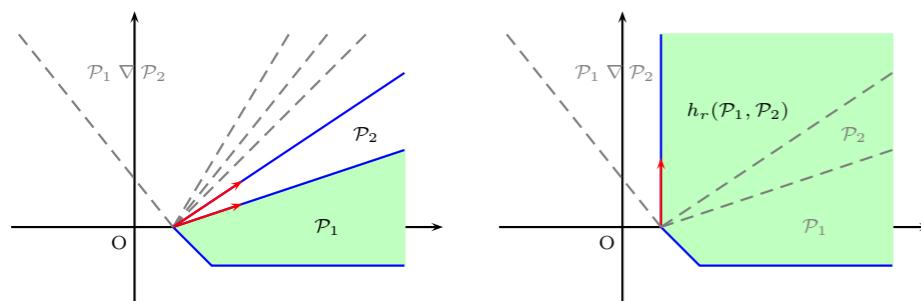
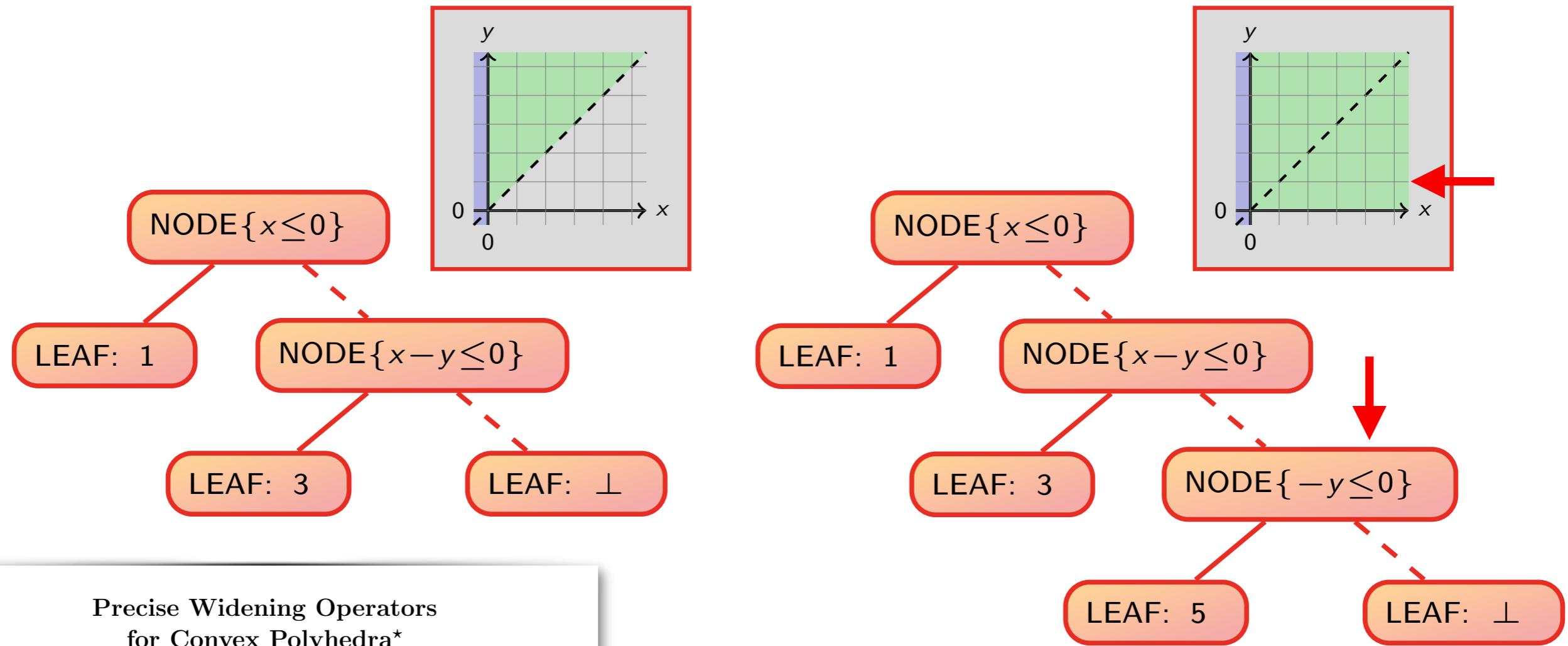


Fig. 2. The heuristics h_r improving on the standard widening.

Widening



Precise Widening Operators
for Convex Polyhedra*

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

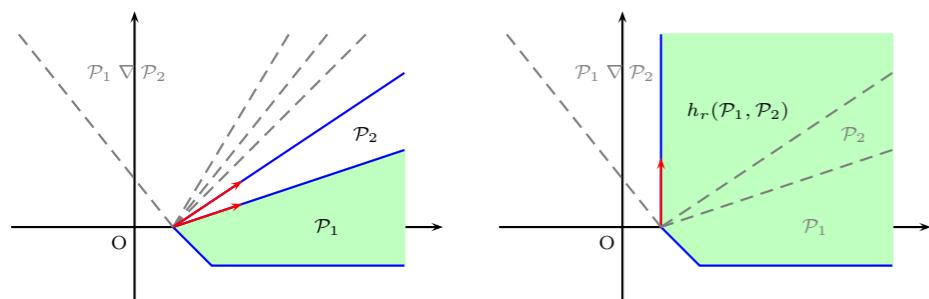
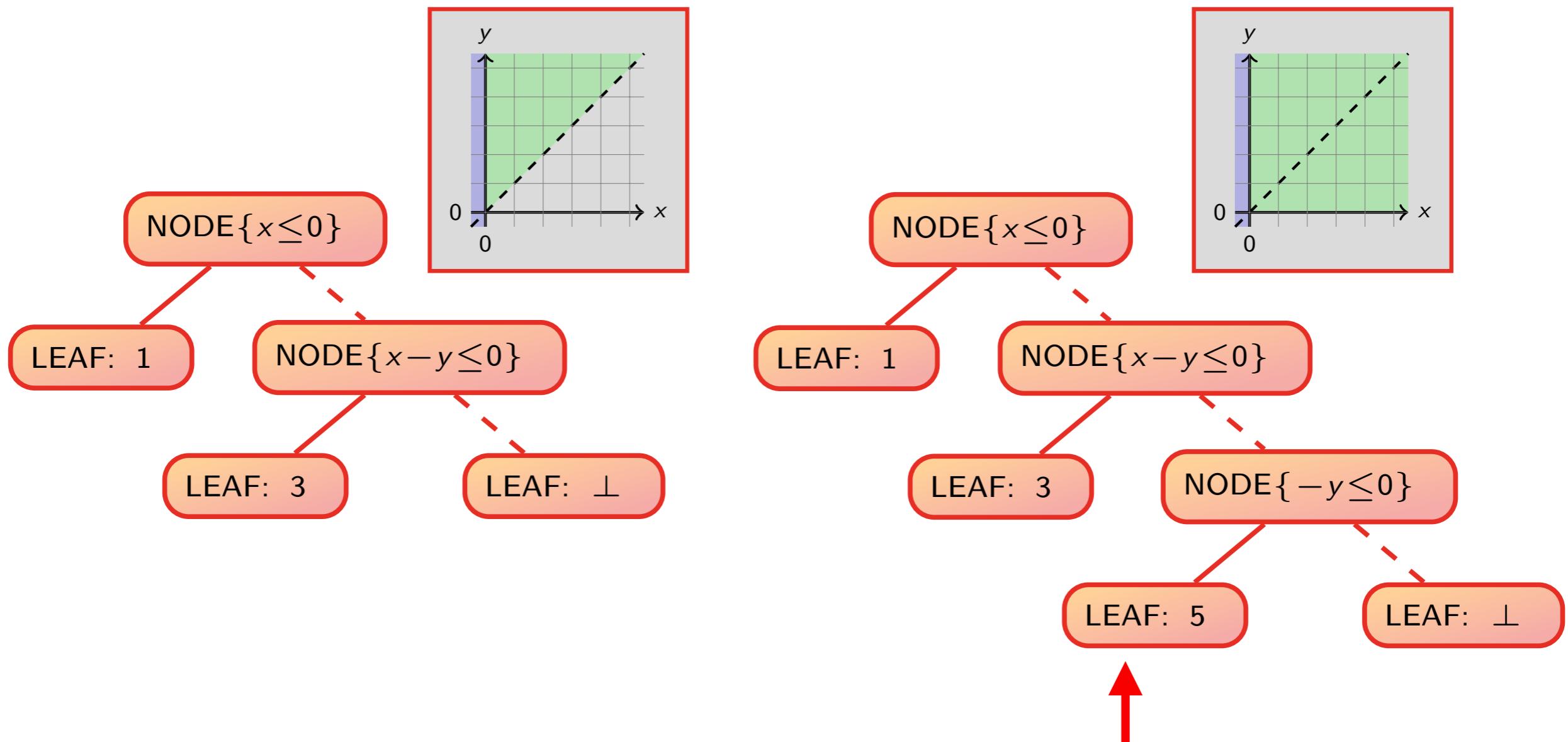


Fig. 2. The heuristics h_r improving on the standard widening.

Widening



Widening

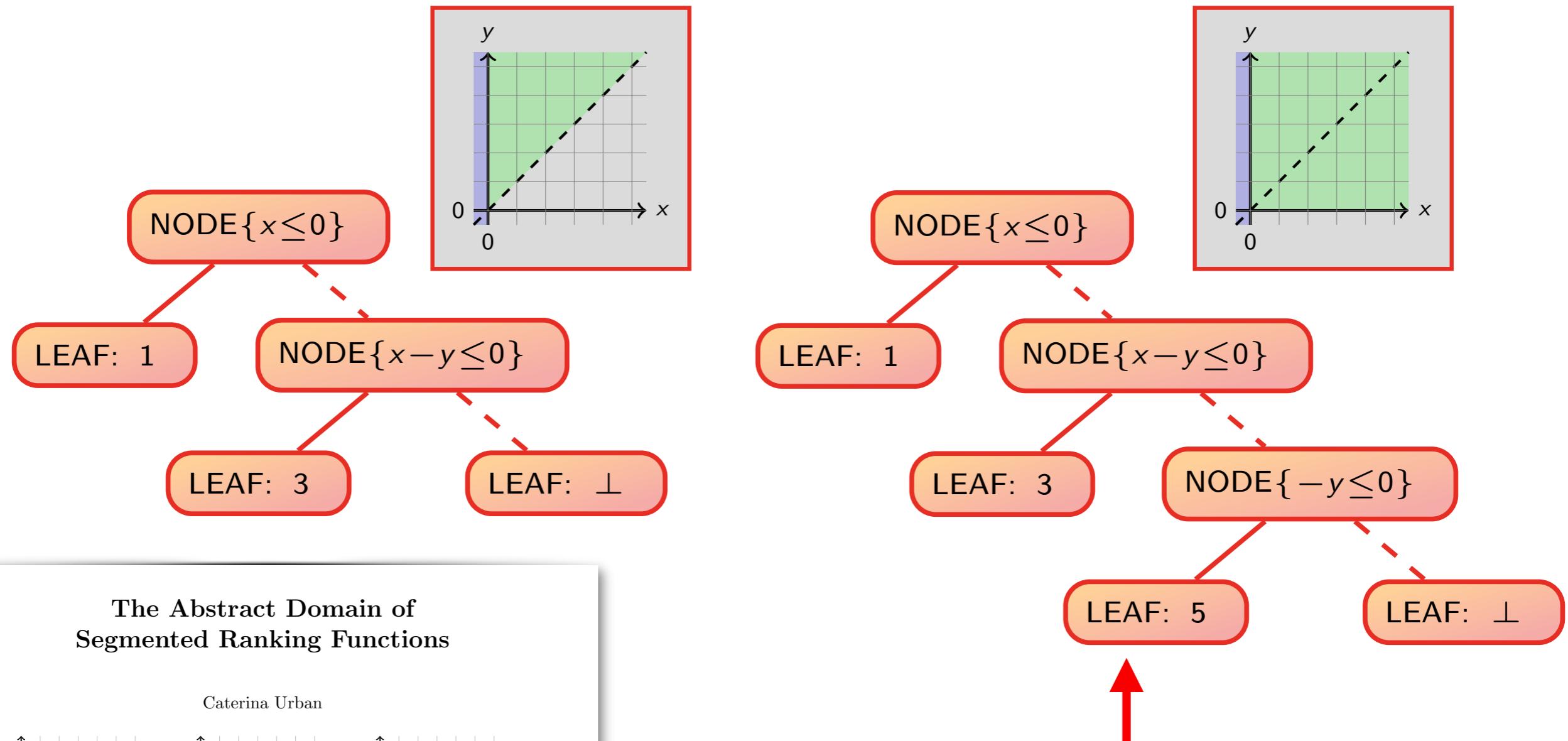


Fig. 7: Example of widening of abstract piecewise-defined ranking functions. The result of widening $v_1^{\#}$ (shown in (a)) with $v_2^{\#}$ (shown in (b)) is shown in (c).

Widening

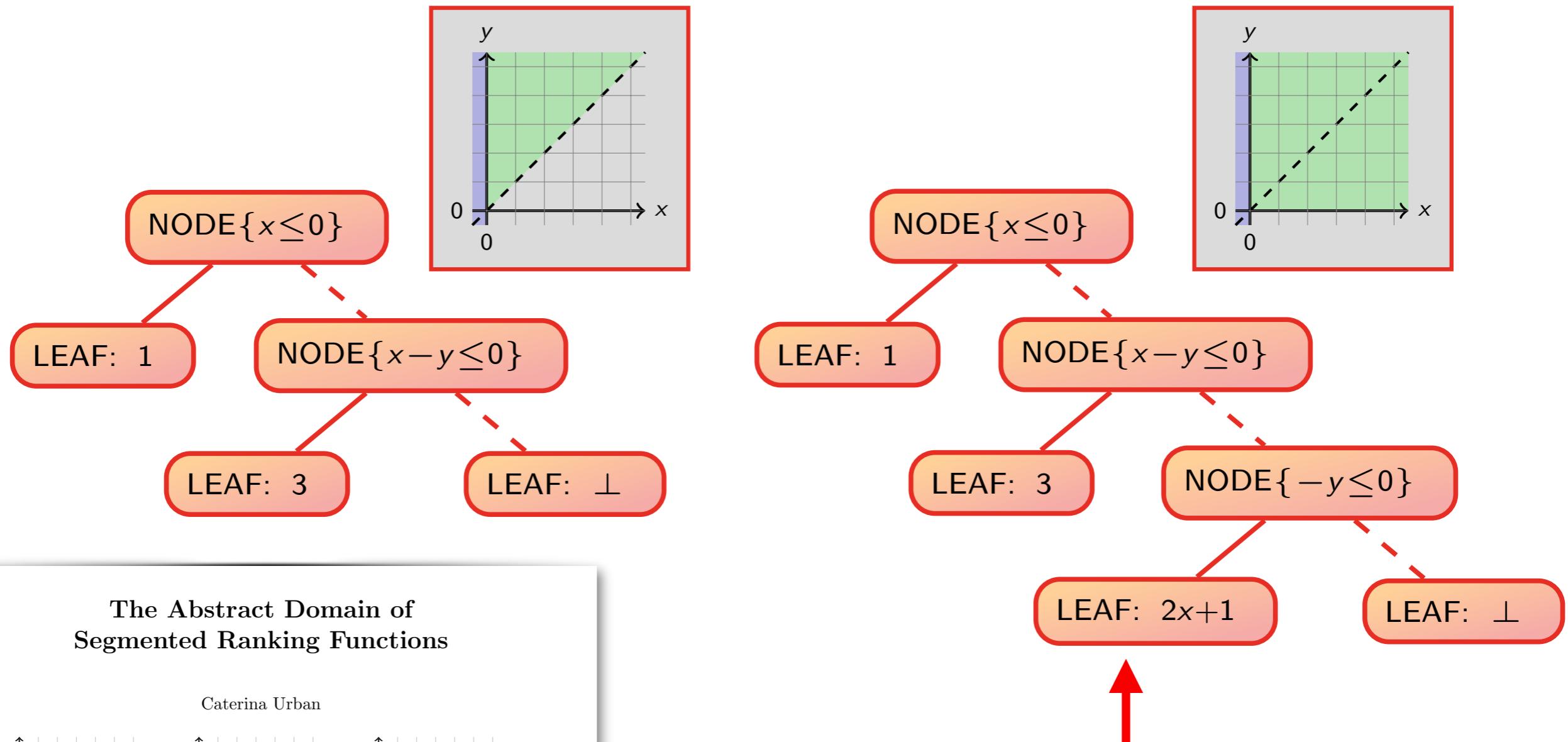
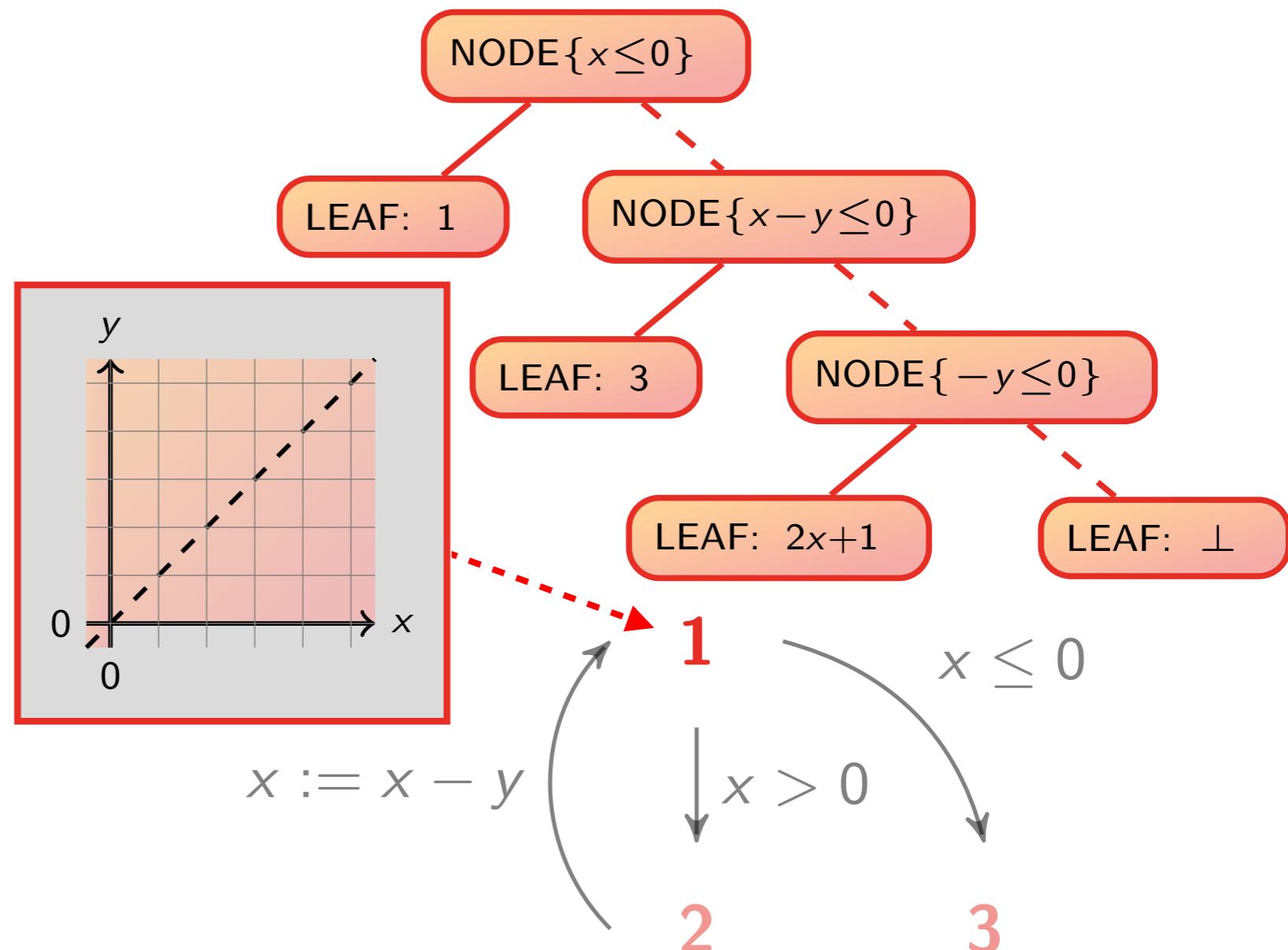


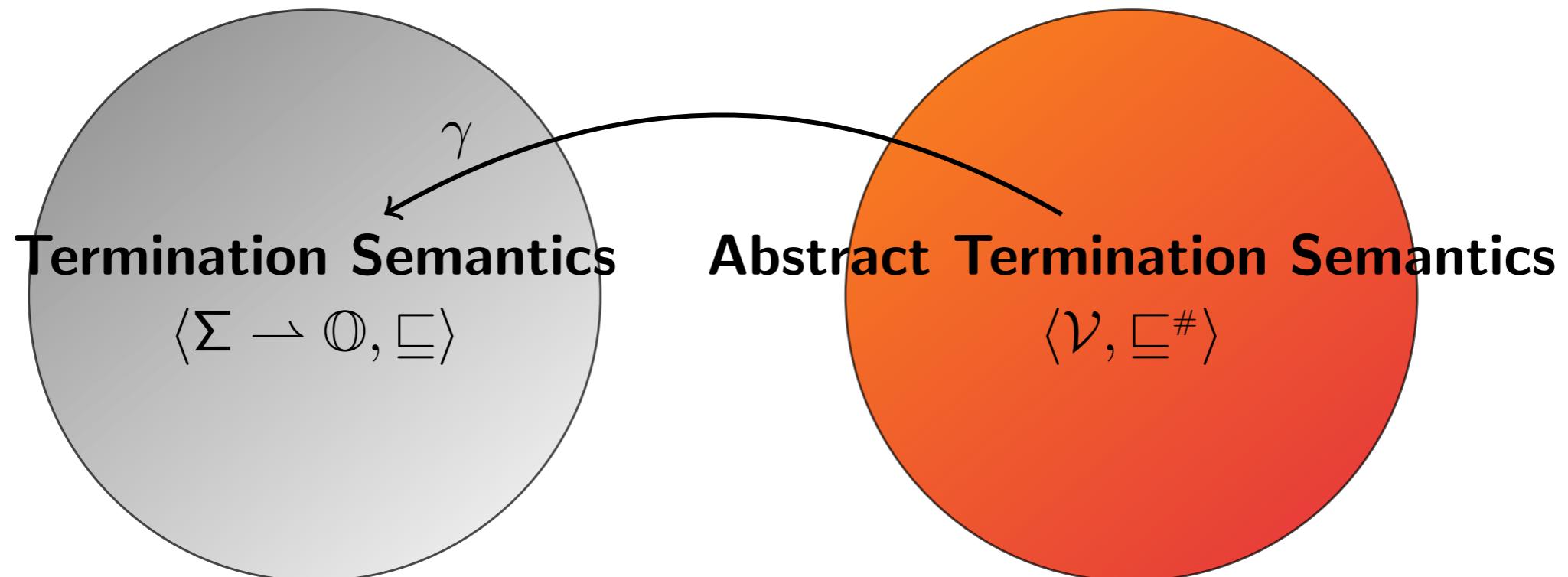
Fig. 7: Example of widening of abstract piecewise-defined ranking functions. The result of widening $v_1^\#$ (shown in (a)) with $v_2^\#$ (shown in (b)) is shown in (c).

Example

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

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





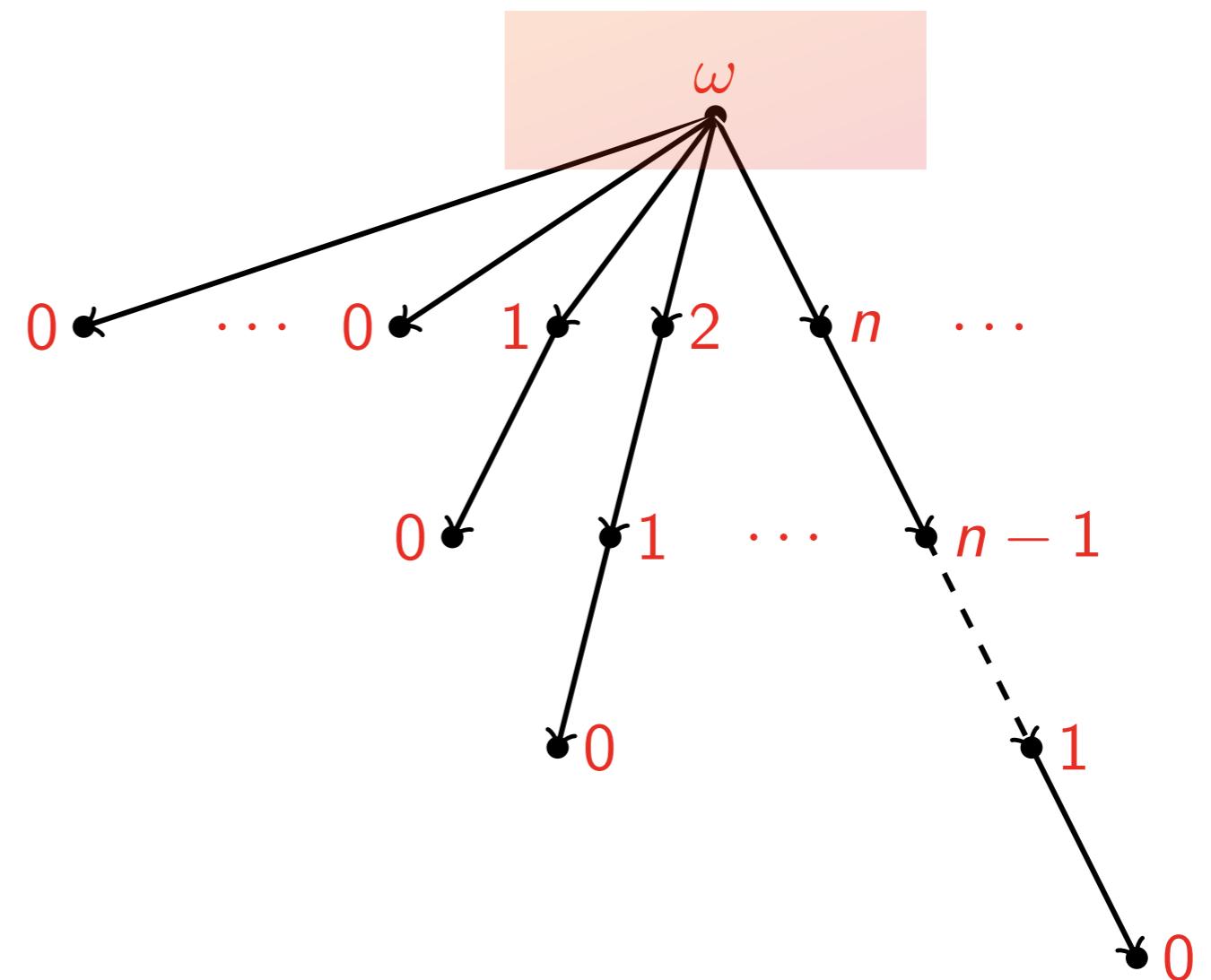
Theorem (Soundness)

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

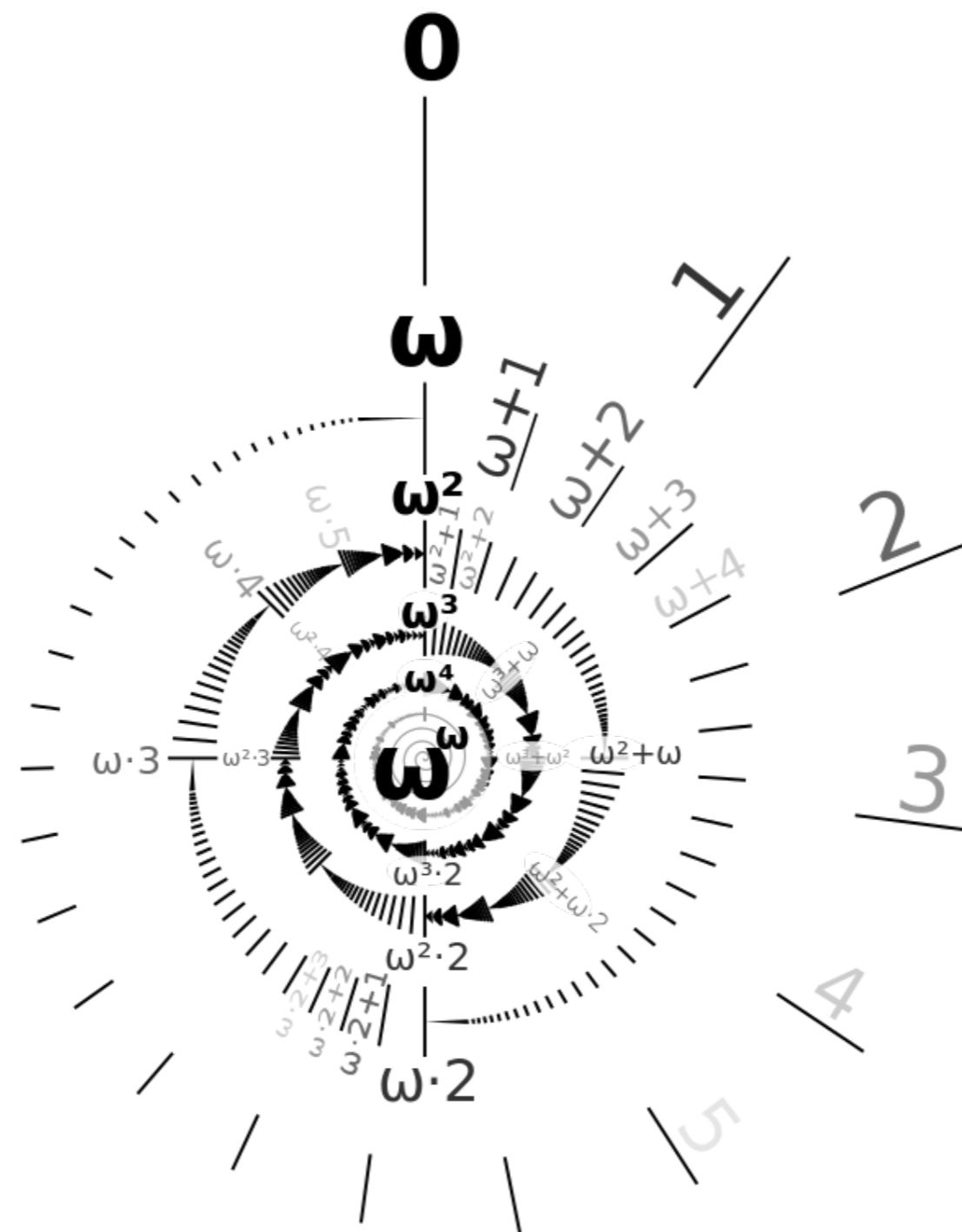
- **remark:** natural-valued ranking functions are **not sufficient!**

Example

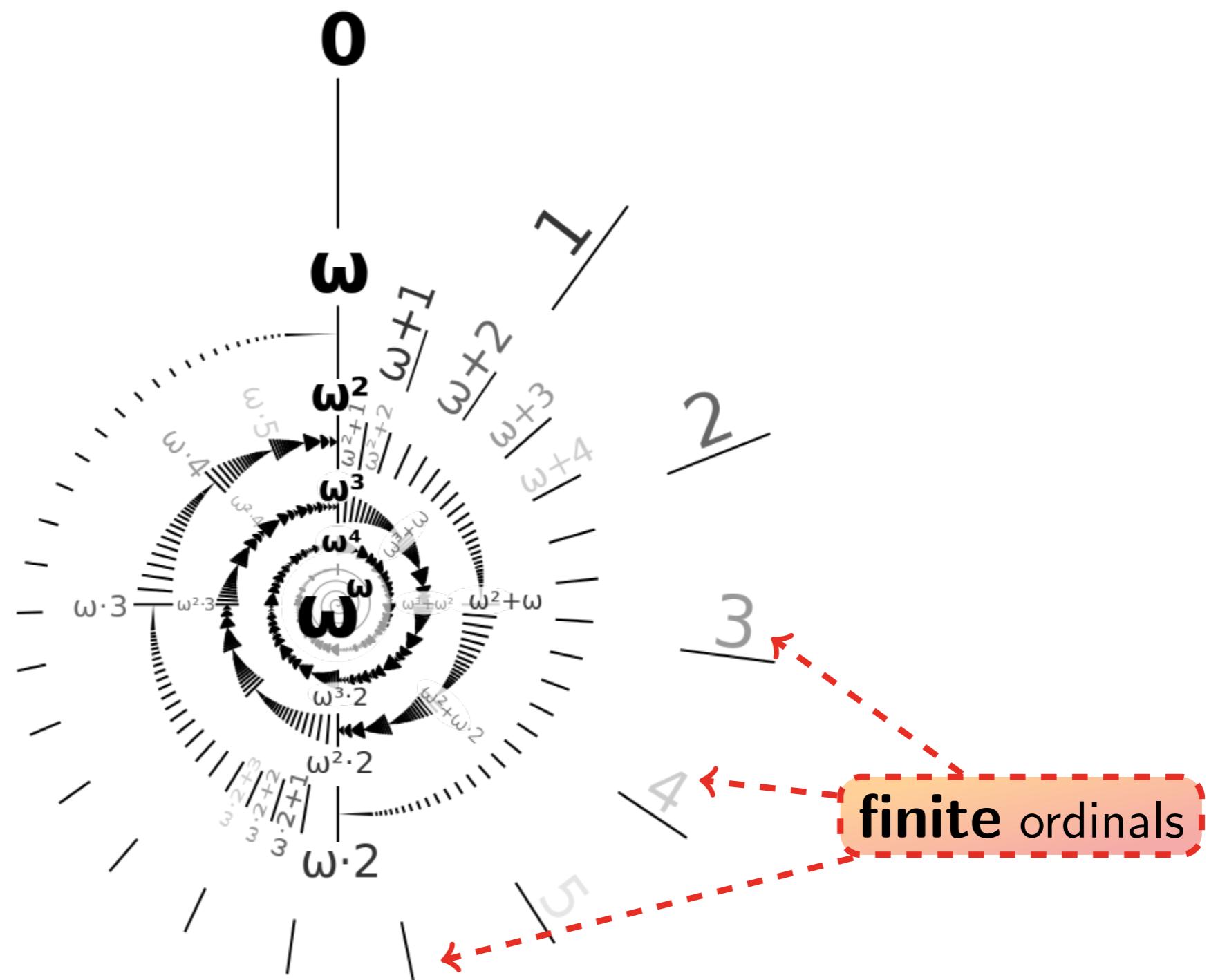
```
int : x
x := ?
while (x > 0) do
  x := x - 1
od
```



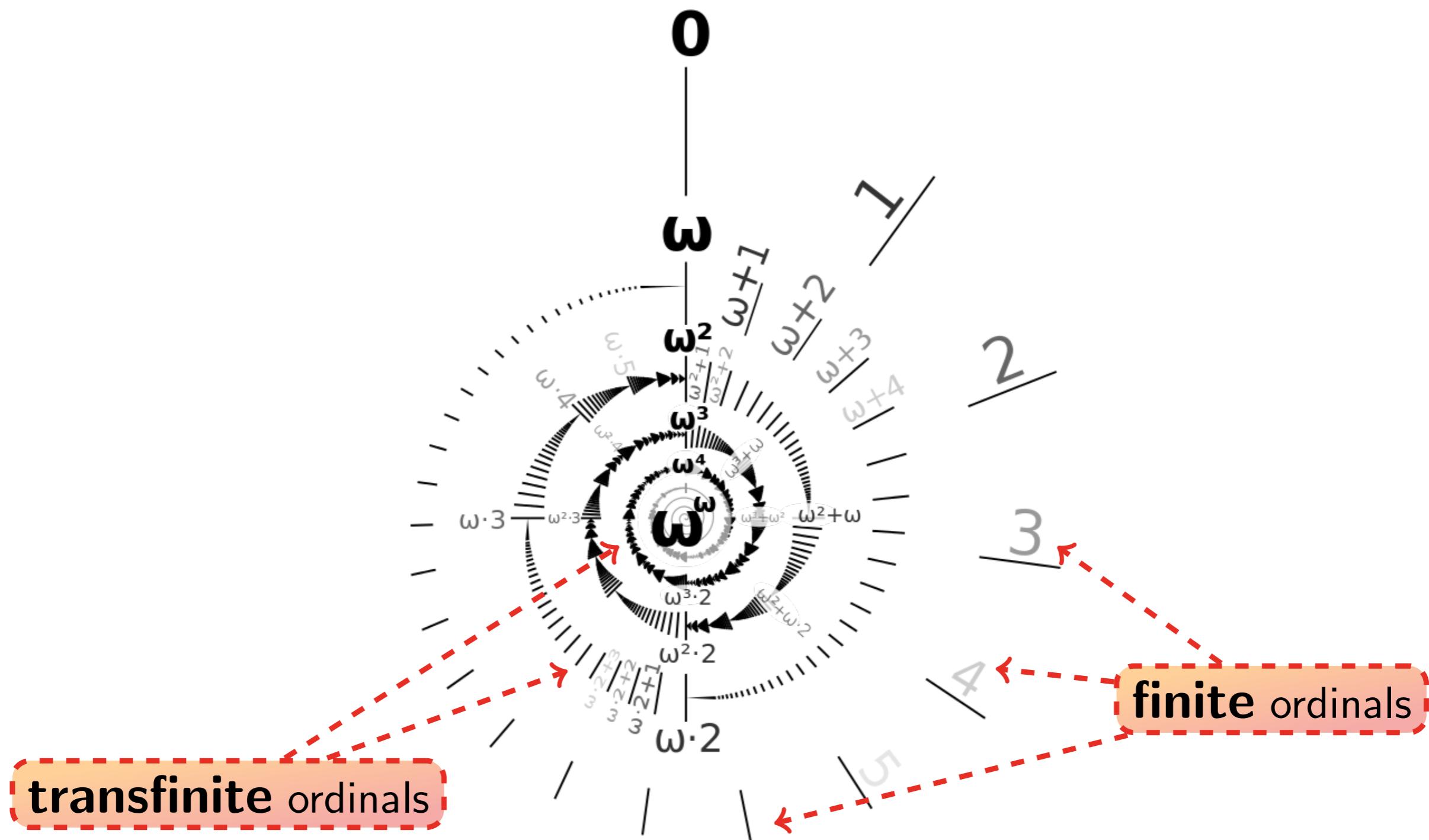
Ordinals



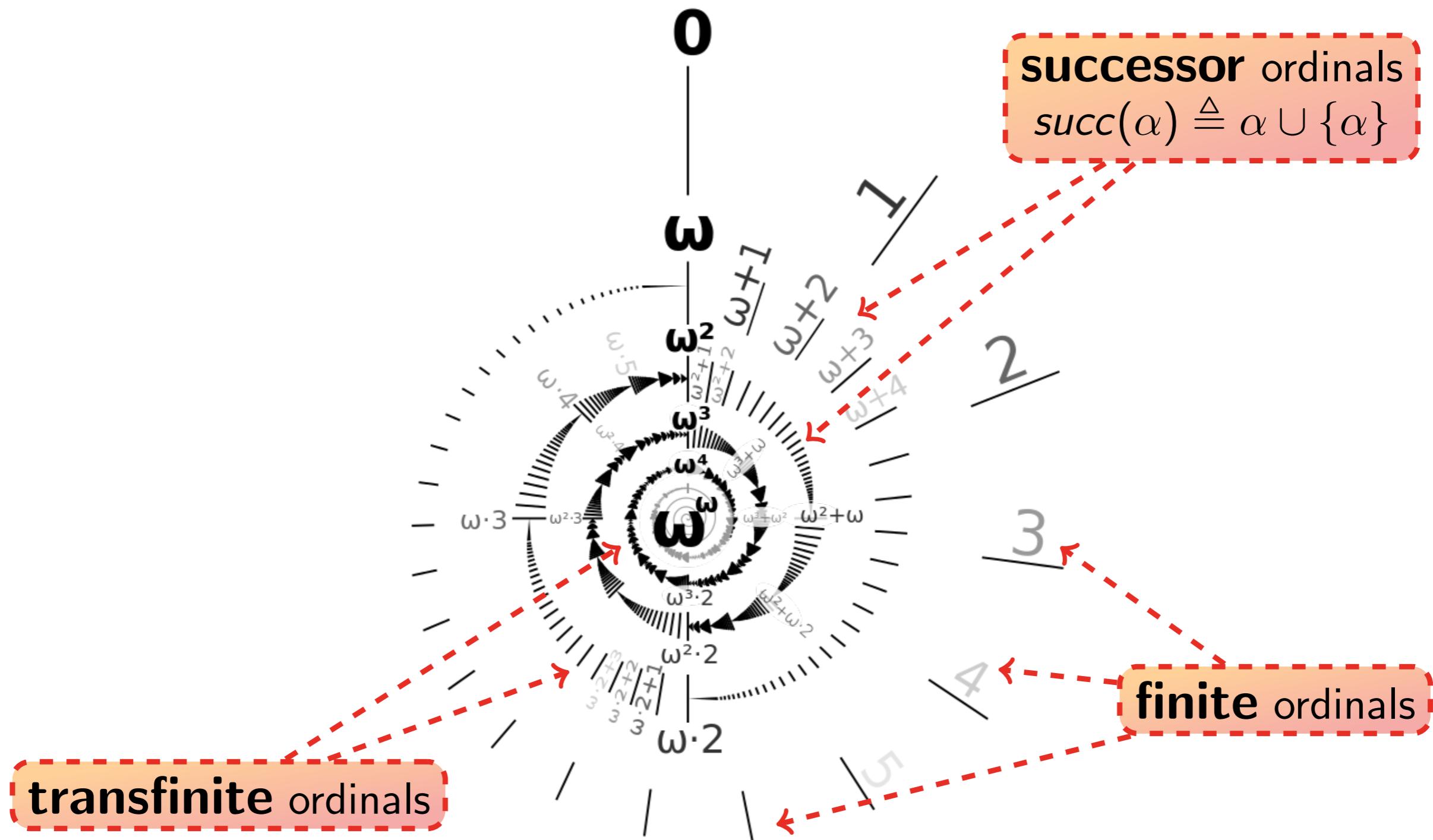
Ordinals



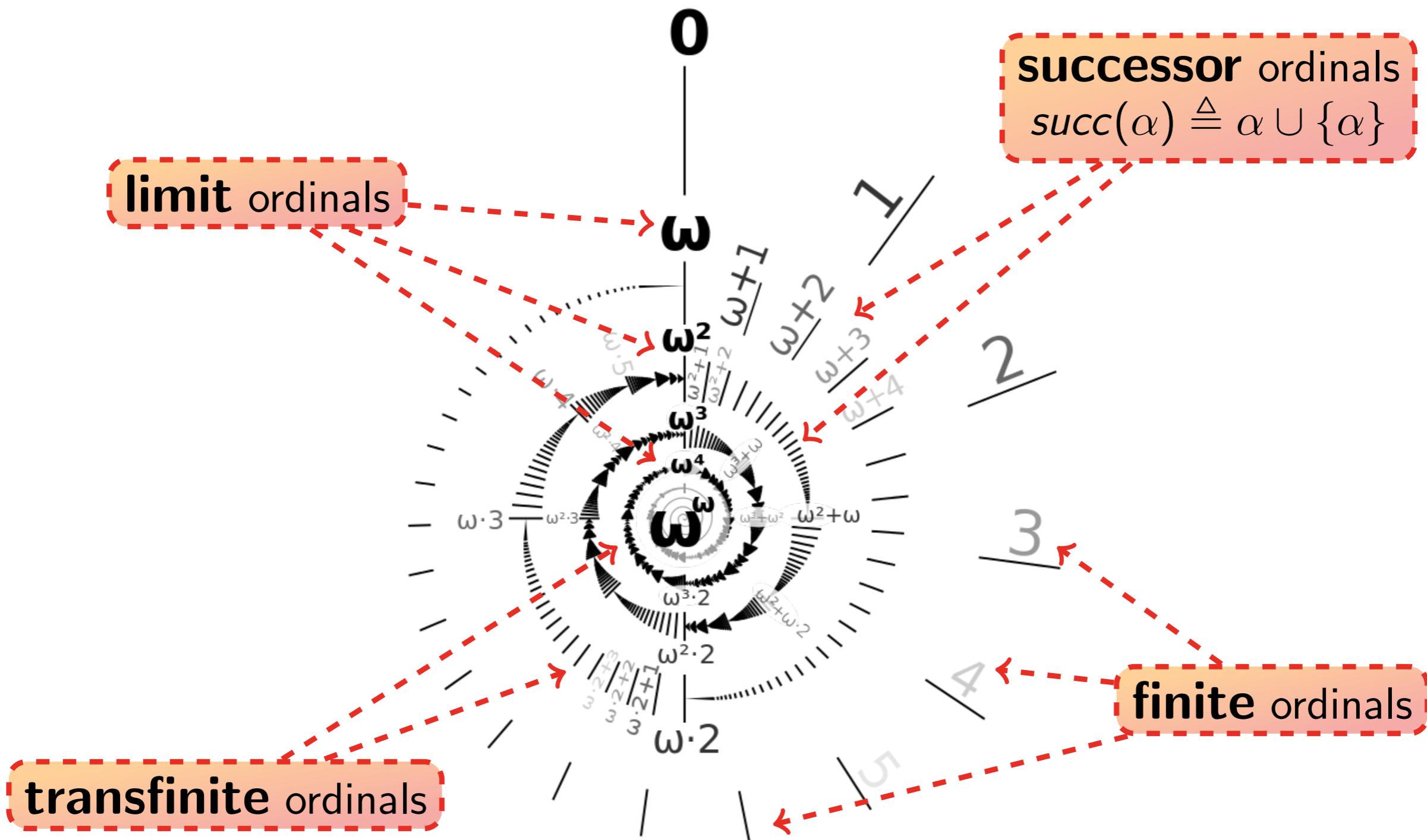
Ordinals



Ordinals



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

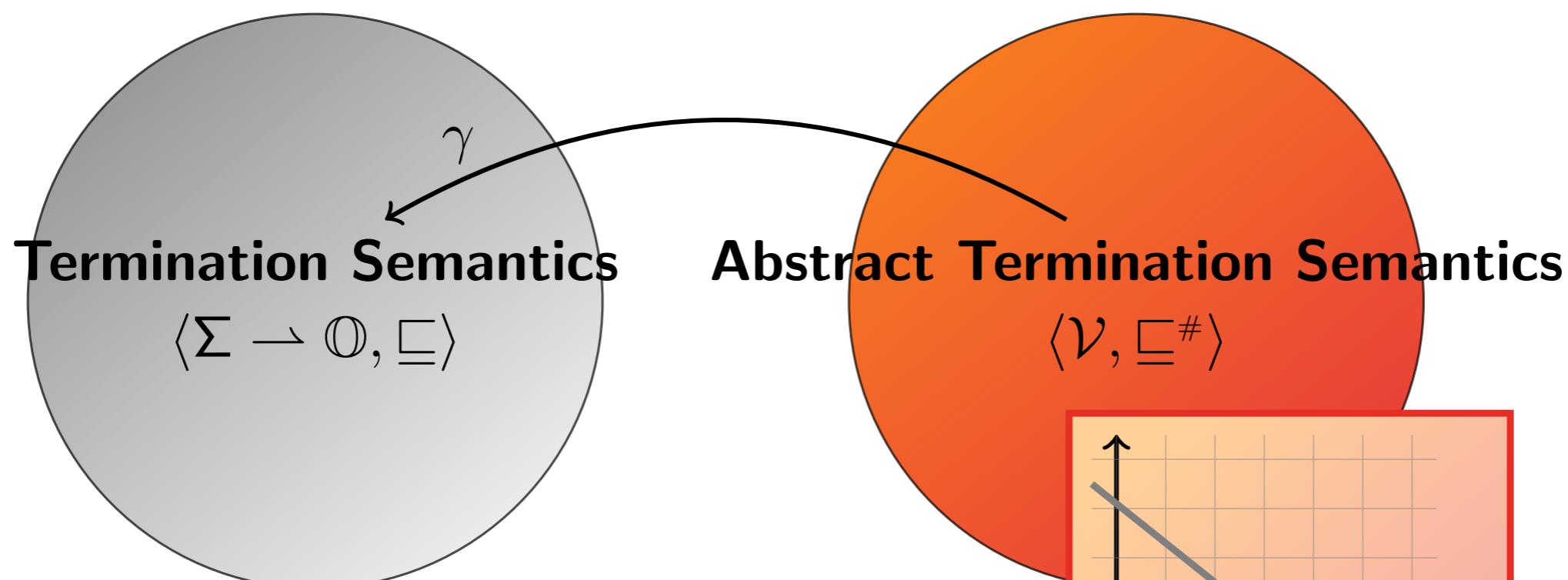
- addition
- multiplication

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

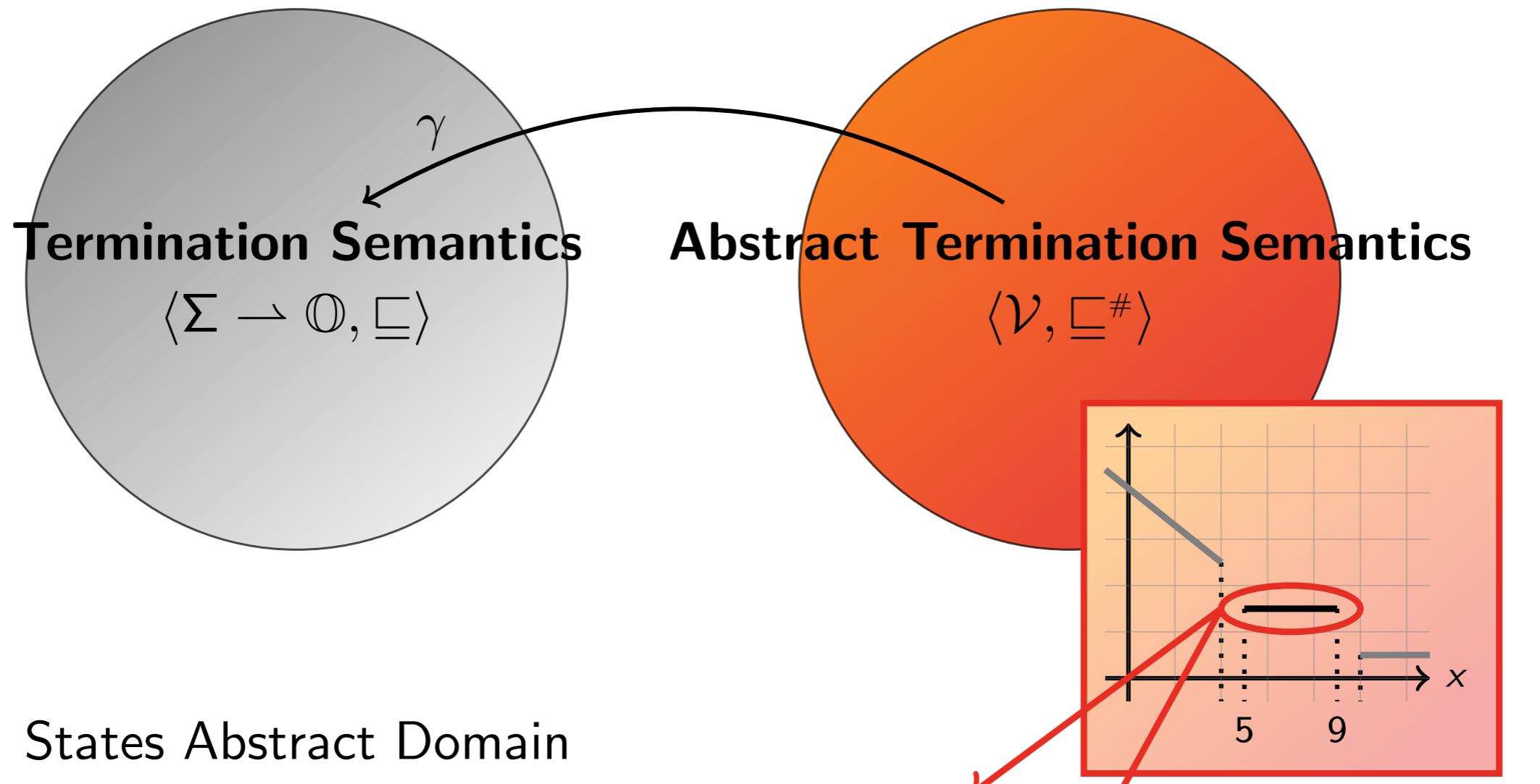
$$\alpha \cdot \text{succ}(\beta) = (\alpha \cdot \beta) + \alpha \quad (\text{successor case})$$

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

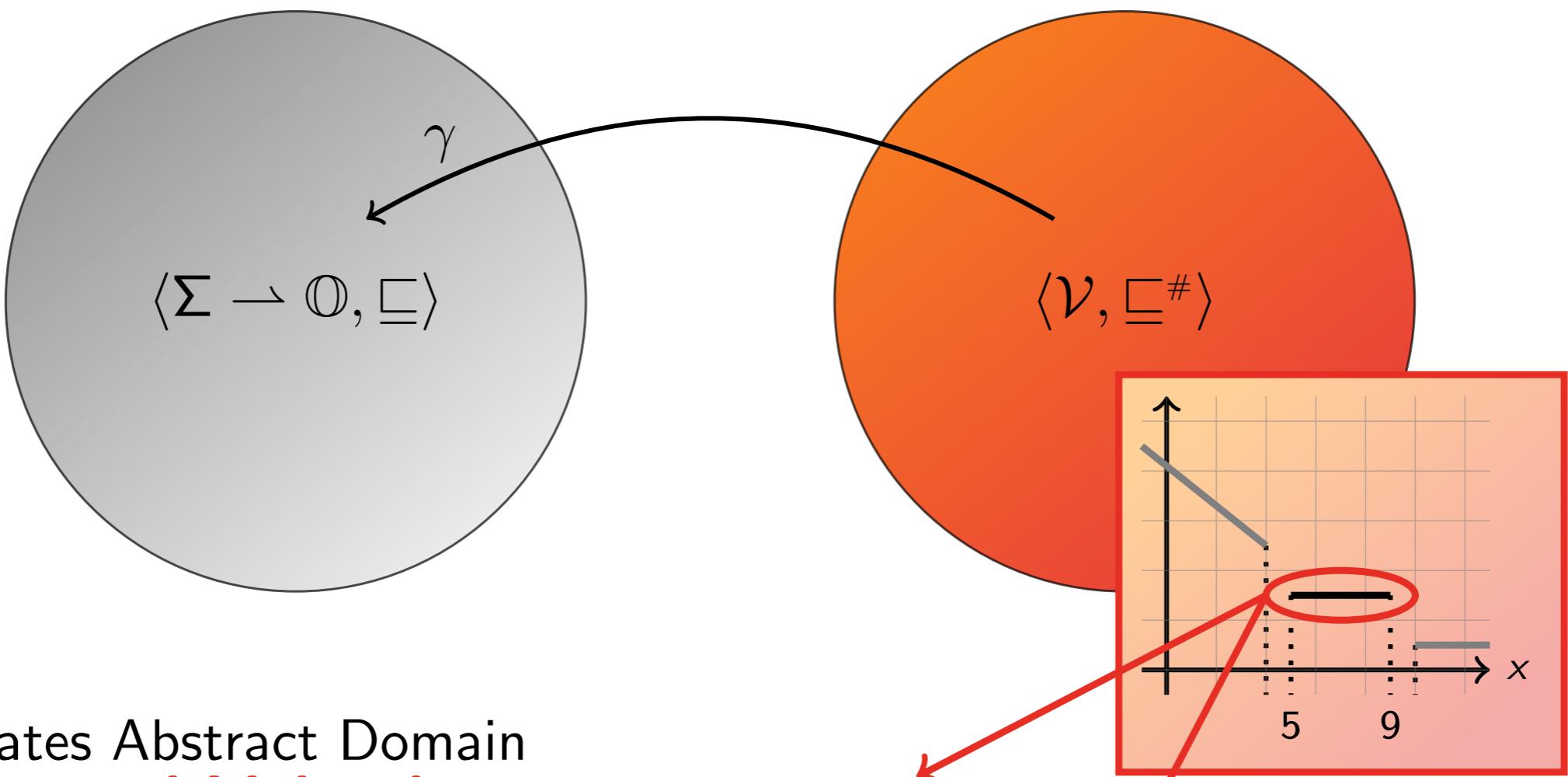
- associative: $(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$
- left distributive: $\alpha \cdot (\beta + \gamma) = (\alpha \cdot \beta) + (\alpha \cdot \gamma)$
- not commutative: $2 \cdot \omega = \omega \neq \omega \cdot 2$
- not right distributive: $(\omega + 1) \cdot \omega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



- States Abstract Domain
- **Functions Abstract Domain**
- Piecewise-Defined Ranking Functions Abstract Domain



- States Abstract Domain S
- **Natural-Valued Functions Abstract Domain** F
- **Ordinal-Valued Functions Abstract Domain** O(F)
- Piecewise-Defined Ranking Functions Abstract Domain V(S, 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

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

$\Downarrow \quad x_1 := ?$

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

Backward Non-Deterministic Assignments

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

Example

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

↓ $x_1 := ?$

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

Backward Non-Deterministic Assignments

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

Example

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

Backward Non-Deterministic Assignments

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

Example

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

$\Downarrow x_1 := ?$

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

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

Backward Non-Deterministic Assignments

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

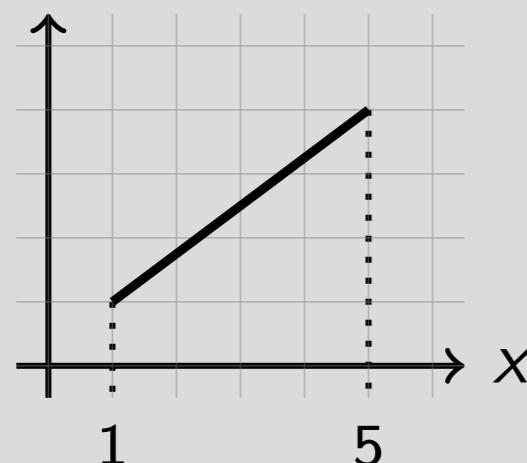
Example

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

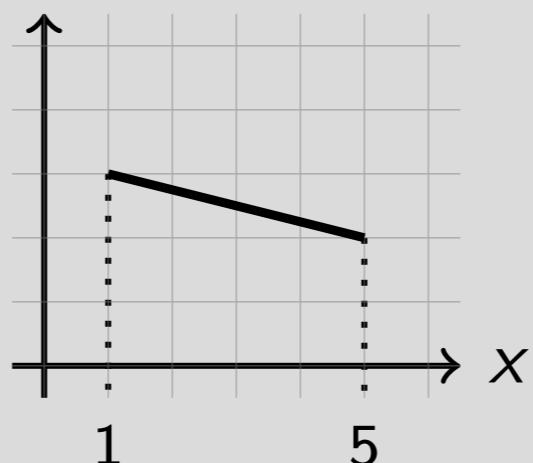
Join

- join of natural-valued functions:

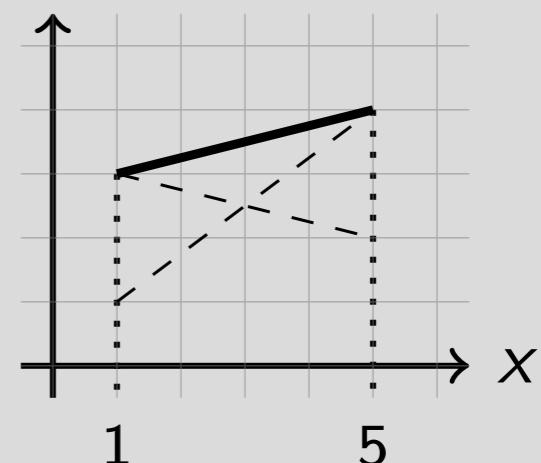
Example



$\sqcup^{\#}$



=



- 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

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

Join

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

Example

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

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

$$[-\infty, +\infty] \mapsto o_1 \sqcup^{\#} o_2 \equiv + 4$$

Join

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

Example

$$\begin{array}{rcl}
 [-\infty, +\infty] \mapsto & o_1 & \equiv \omega^2 \cdot x_1 + \omega \cdot x_2 + 3 \\
 [-\infty, +\infty] \mapsto & o_2 & \equiv \omega^2 \cdot x_1 + \omega \cdot (-x_2) + 4 \\
 \hline
 [-\infty, +\infty] \mapsto & o_1 \sqcup^\# o_2 & \equiv \omega^2 \cdot 1 + \omega \cdot 0 + 4
 \end{array}$$

The result $\omega^2 \cdot 1 + \omega \cdot 0 + 4$ is highlighted with a red dashed box. A red arrow points from the term $\omega^2 \cdot 1$ to this box.

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

Join

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

Example

$$\begin{array}{rcl}
 [-\infty, +\infty] \rightarrow o_1 & \equiv & \omega^2 \cdot \textcolor{red}{x}_1 + \omega \cdot x_2 + 3 \\
 [-\infty, +\infty] \rightarrow o_2 & \equiv & \omega^2 \cdot \textcolor{red}{x}_1 + \omega \cdot (-x_2) + 4 \\
 \hline
 [-\infty, +\infty] \rightarrow o_1 \sqcup^{\#} o_2 & \equiv & \omega^2 \cdot \textcolor{red}{x}_1 \textcolor{red}{\omega^2 \cdot 1} + 4
 \end{array}$$

Join

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

Example

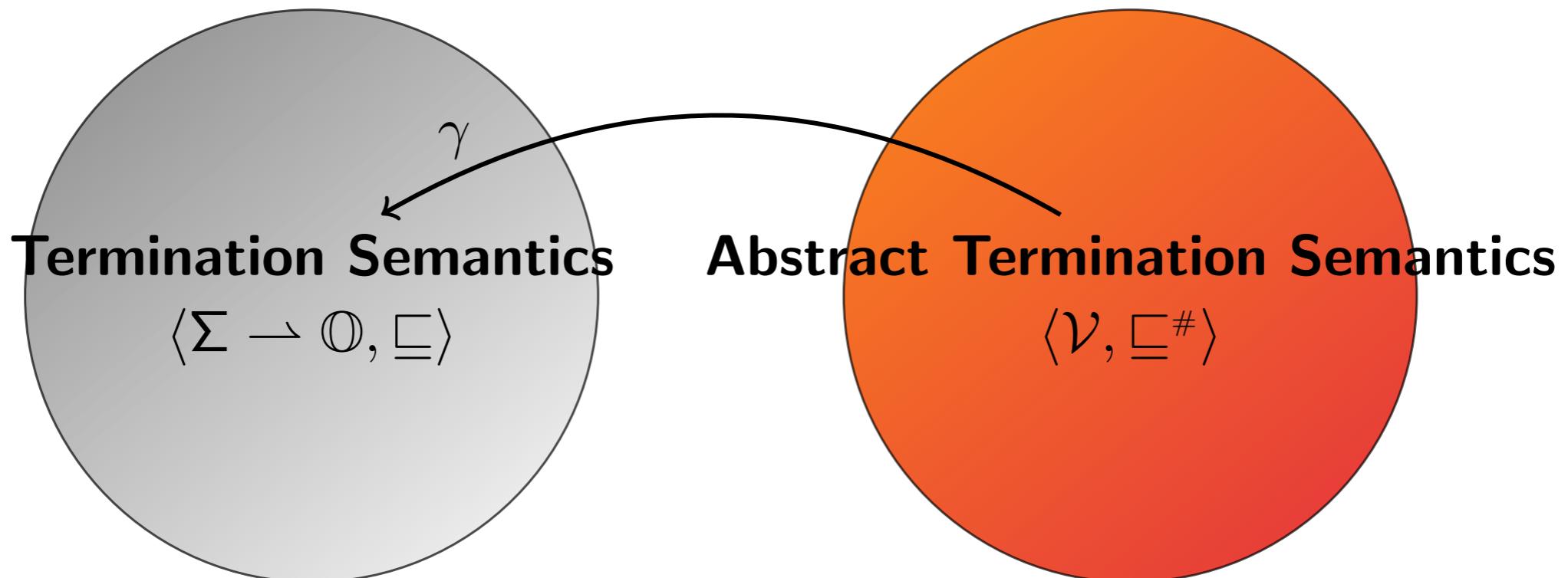
$$\begin{array}{rcl}
 [-\infty, +\infty] \mapsto & o_1 & \equiv \quad \omega^2 \cdot \textcolor{red}{x}_1 \quad + \quad \omega \cdot x_2 \quad + \quad 3 \\
 [-\infty, +\infty] \mapsto & o_2 & \equiv \quad \omega^2 \cdot \textcolor{red}{x}_1 \quad + \quad \omega \cdot (-x_2) \quad + \quad 4 \\
 \hline
 [-\infty, +\infty] \mapsto & o_1 \sqcup^\# o_2 & \equiv \quad \omega^2 \cdot (\textcolor{red}{x}_1 + 1) \quad + \quad 4
 \end{array}$$

Join

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

Example

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



Theorem (Soundness)

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

Example

```
int : x1, x2
while 1(x1 > 0 ∧ x2 > 0) do
    if 2( ? ) then
        3x1 := x1 - 1
        4x2 := ?
    else
        5x2 := x2 - 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
        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, x1, x2
1 x1 := N
while 2(x1 ≥ 0) do
    3 x2 := N
    while 4(x2 ≥ 0) do
        5 x2 := x2 − 1
    od6
    7 x1 := x1 − 1
od8
```

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

Non-Linear Ranking Functions

Example

```

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

The screenshot shows a web browser window titled "FuncTion" with the URL "www.di.ens.fr/~urban/FuncTion.html". The page content is as follows:

Welcome to FuncTion's web interface!

Type your program:

or choose a predefined example:

and choose an entry point:

Forward option(s):

- Widening delay:

Backward option(s):

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

Guarantee Semantics

**Proving Guarantee and Recurrence
Temporal Properties by Abstract Interpretation^{*}**

Caterina Urban and Antoine Miné
INRIA & CTOOL & INRIA, France
curban,amine@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 Mastra and Pnueli, we focus on guarantees (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 termination presented by Cousot and Cousot. Specifically, static analyses of guarantee and recurrence temporal properties are automatically derived by abstraction of their corresponding concrete properties. These methods automatically infer sufficient preconditions for the temporal properties by solving existing numerical abstract domains based on pointer-defined ranking functions. We augment these abstract domains with new abstract operations, 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 Mastra and Pnueli [24] distinguishes four basic classes:

- safety properties: “something good always happens”, i.e., the program never reaches an unacceptable state (e.g., mutual non-exclusion, 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 [6], a general theory of semantic

* The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 269916 (ARTEMIS project MIDAT) (see Article II.9 of the EU Grant Agreement).



program \mapsto maximal trace semantics $\rightarrow \varphi$ -guarantee semantics

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

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

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

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

program \mapsto maximal trace semantics $\rightarrow \varphi$ -guarantee semantics

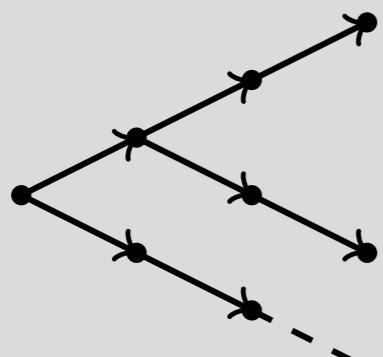
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

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

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

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

Example



: program \mapsto maximal trace semantics $\rightarrow \varphi$ -guarantee semantics :

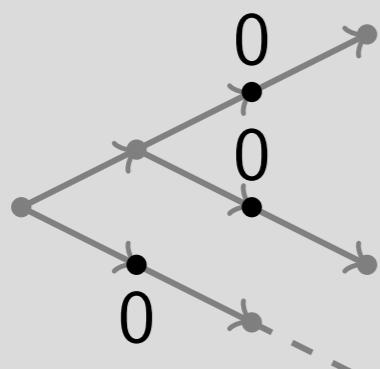
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

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

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

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

Example



; program \mapsto maximal trace semantics $\rightarrow \varphi$ -guarantee semantics ;

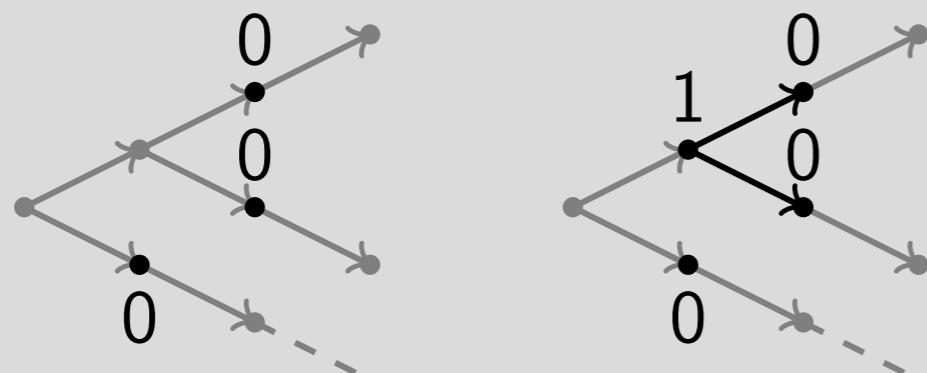
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

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

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

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

Example



program \mapsto maximal trace semantics $\rightarrow \varphi$ -guarantee semantics

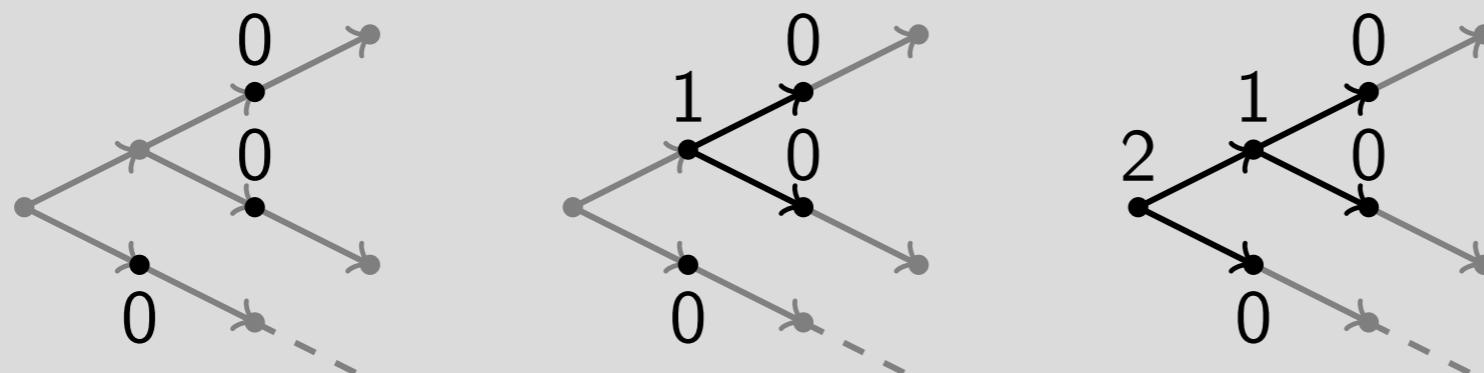
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

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

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

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

Example



program \mapsto maximal trace semantics $\rightarrow \varphi$ -guarantee semantics

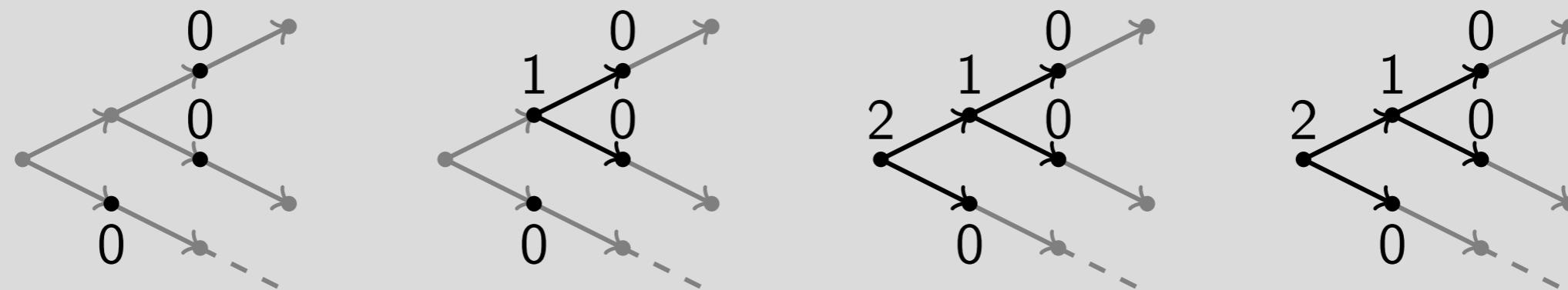
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

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

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

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

Example



program \mapsto maximal trace semantics $\rightarrow \varphi$ -guarantee semantics

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

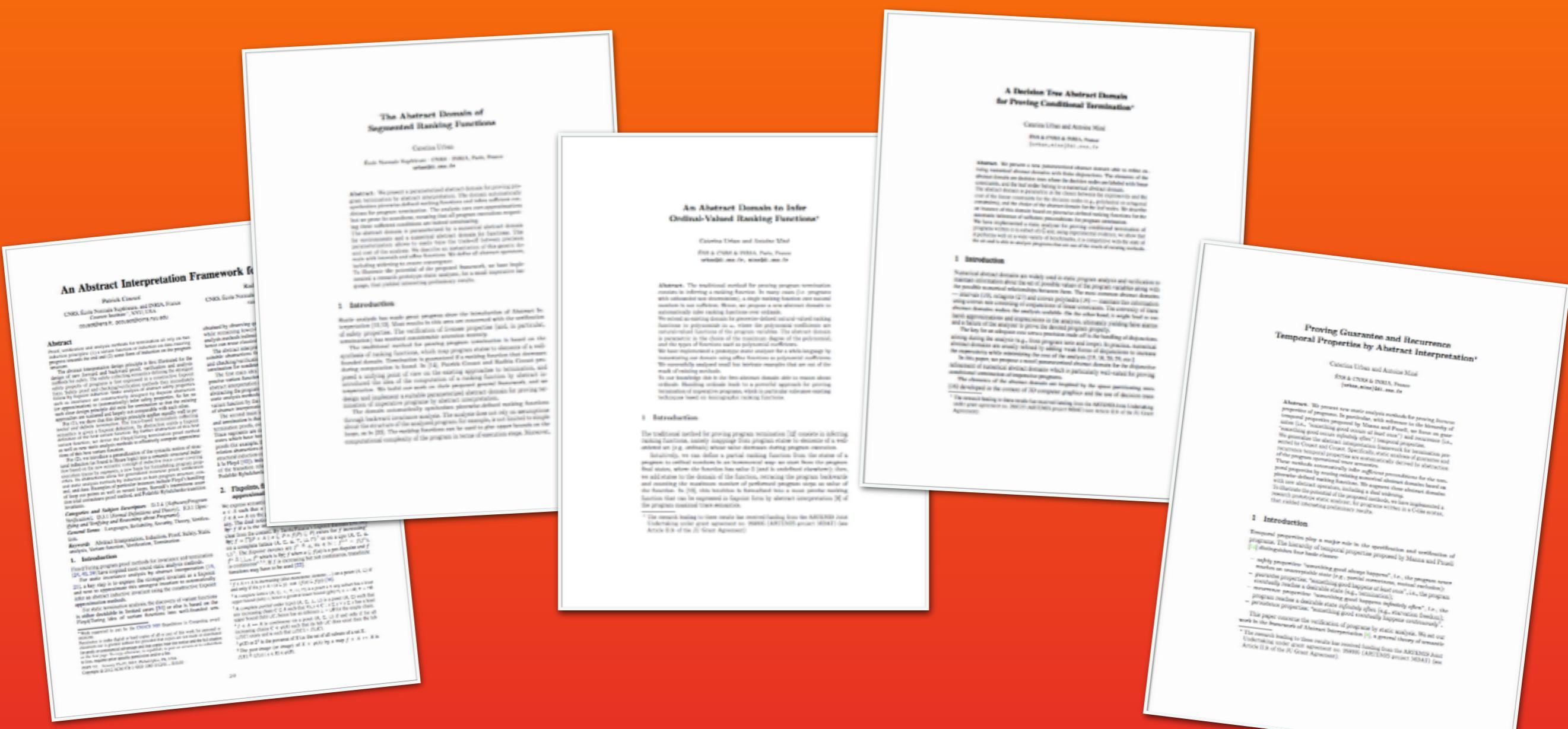
$$\mathcal{T}_g^\varphi \stackrel{\text{def}}{=} \text{lfp } F_g^\varphi$$

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

Theorem (Soundness and Completeness)

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

Bibliography



- **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)