

Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs

**Analyse Statique par Interprétation Abstraite de Propriétés
Temporelles Fonctionnelles des Programmes**

Caterina Urban



July 9th, 2015
PhD Defense
École Normale Supérieure, Paris

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 discusses 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 suggests letting the Zune run out of battery to fix it.

DISRUPT Register Now to Save £300 on Disrupt Europe: London AOL Privacy Policy and Terms of Service

Gadgets Headline Zune Feature

30GB Zunes all over the world fail en masse

Posted Dec 31, 2008 by [Matt Burns \(@mjburnsy\)](#)

Share 0 Share 0 Tweet 0

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 two visible tabs. The top tab is titled "Zune bug explained in detail" and the URL is "techcrunch.com/2008/12/31/zune-bug-explained-in-detail/". The bottom tab is titled "30GB Zunes" and the URL is "techcrunch.com/2008/12/31/30gb-zunes/".

Zune bug explained in detail
Posted Dec 31, 2008 by Devin Coldewey

Earlier today, the sound of thousands of Zune owners crying out in terror made ripples across the blogosphere. The response from Microsoft is to wait until tomorrow and all will be well. You're probably wondering, what kind of bug fixes itself?

Well, I've got the code here and it's very simple, really; if you've taken an introductory programming class, you'll see the error right away.

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

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

30GB Zunes
Posted Dec 31, 2008 by Matt

It seems that a random bug in a bunch of Zune 30s just started might have a gadget Y2K bug. The same mantra saying that a full reboot. We're sure Microsoft lines open up for the last time.

Update 2: The solution is ..
you wake up tomorrow and

● failure due to

<http://techcrunch>

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

- Microsoft **Zune** December 31st, 2008
[http://techcrunch.com/2008/12/31/
zune-bug-explained-in-detail/](http://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/)
- **Apache** HTTP Server Versions <2.3.3
[http://cve.mitre.org/cgi-bin/
cvename.cgi?name=CVE-2009-1890](http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1890)
- Microsoft **Azure Storage Service** November 19th, 2014
[http://azure.microsoft.com/blog/2014/11/19/
update-on-azure-storage-%20service-interruption](http://azure.microsoft.com/blog/2014/11/19/update-on-azure-storage-%20service-interruption)

- **safety properties:** “*something good always happens*”
(e.g., partial correctness, mutual exclusion)

$\Box \varphi$

- **guarantee properties:** “*something good happens at least once*” (e.g., total correctness, termination)

$\Diamond \varphi$

- **recurrence properties:** “*something good happens infinitely often*” (e.g., starvation freedom)

$\Box \Diamond \varphi$

- **persistence properties:** “*something good eventually happens continuously*” (e.g., stabilization)

$\Diamond \Box \varphi$

Ranking Functions

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



Primer, Part One.

Checking a Large Routine, by Dr. A. M. 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 program easily follows.

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

| | |
|-------|-------|
| 137 | |
| 596 | |
| 479 | |
| 437 | |
| 776 | <hr/> |
| 26104 | |

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

| | |
|-------|-------|
| 137 | |
| 396 | |
| 679 | |
| 437 | |
| 776 | <hr/> |
| 26104 | |

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

| | |
|-----|-------|
| 379 | |
| 213 | <hr/> |
| 592 | |

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 a without the use of a multiplier, multiplication being carried out by repeated addition.

At a typical moment of the process we have recorded r and $s \cdot r$ for some r, s . We can change $s \cdot r$ to $(s+1) \cdot r$ by addition of r . Now $s \cdot r$ is $s \cdot r$. We can change r to $r+1$ 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 below 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:

- a dashed letter indicates the value at the end of the process represented by the box;
- 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

67.

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).



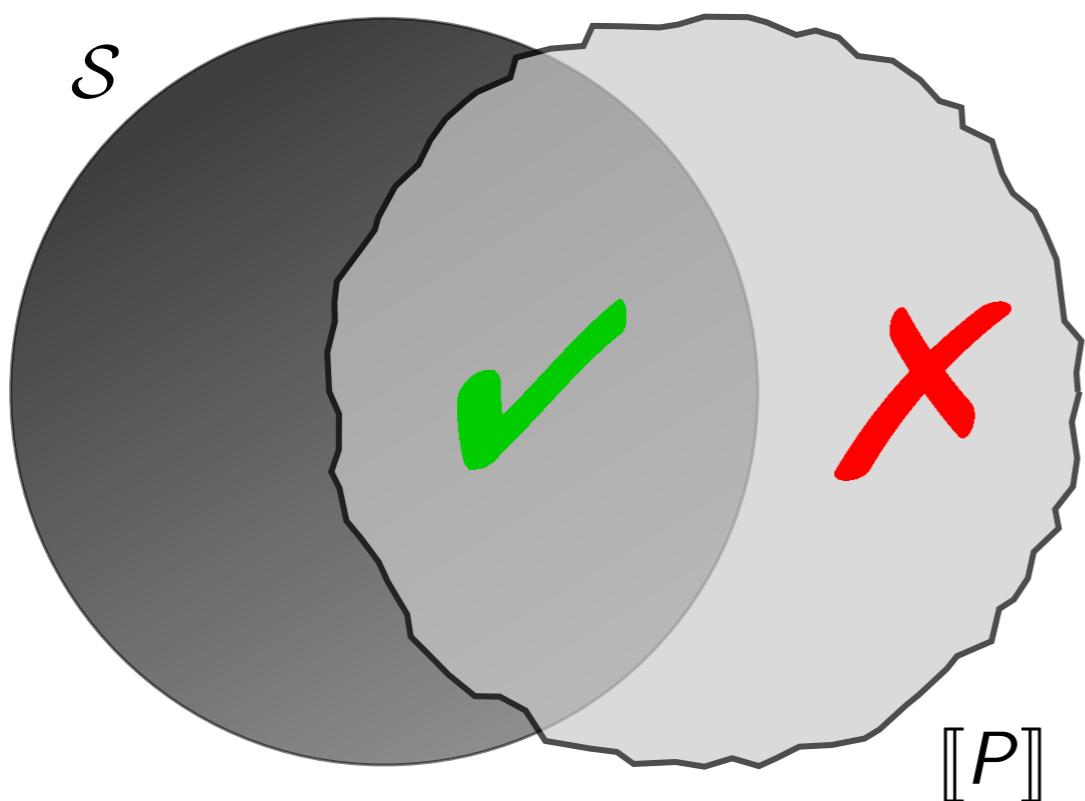
Turing - *Checking a Large Routine* (1949)

Floyd - *Assigning Meanings To Programs* (1967)

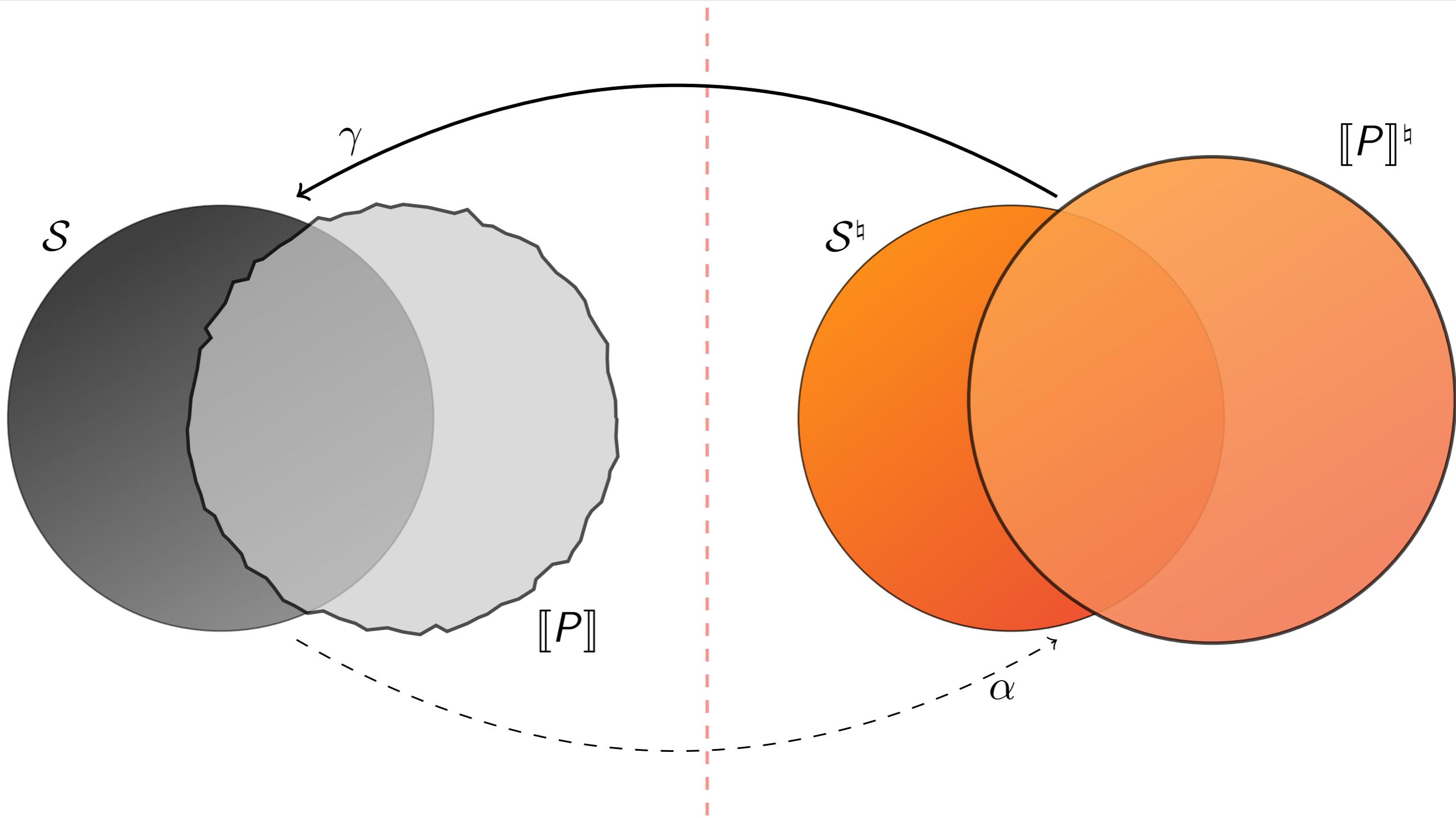
- **idea:** inference of ranking functions by **abstract interpretation**
 - termination semantics
 - **guarantee semantics**
 - **recurrence semantics**

- **idea:** inference of ranking functions by **abstract interpretation**
 - termination semantics
 - **guarantee semantics**
 - **recurrence semantics**
- family of **abstract domains** for liveness properties
 - **piecewise-defined ranking functions**
 - backward analysis
 - sufficient preconditions

Abstract Interpretation

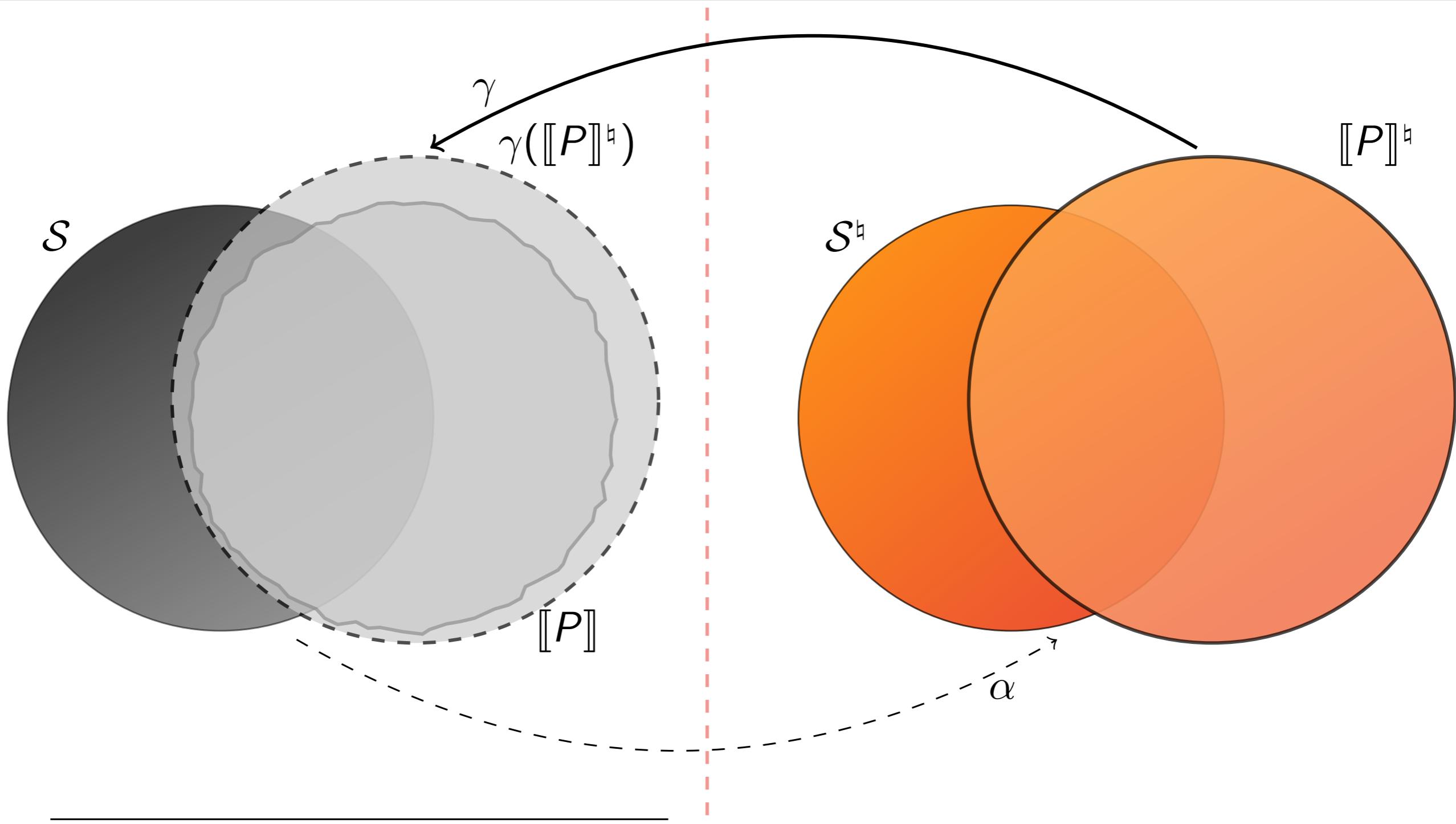


Abstract Interpretation

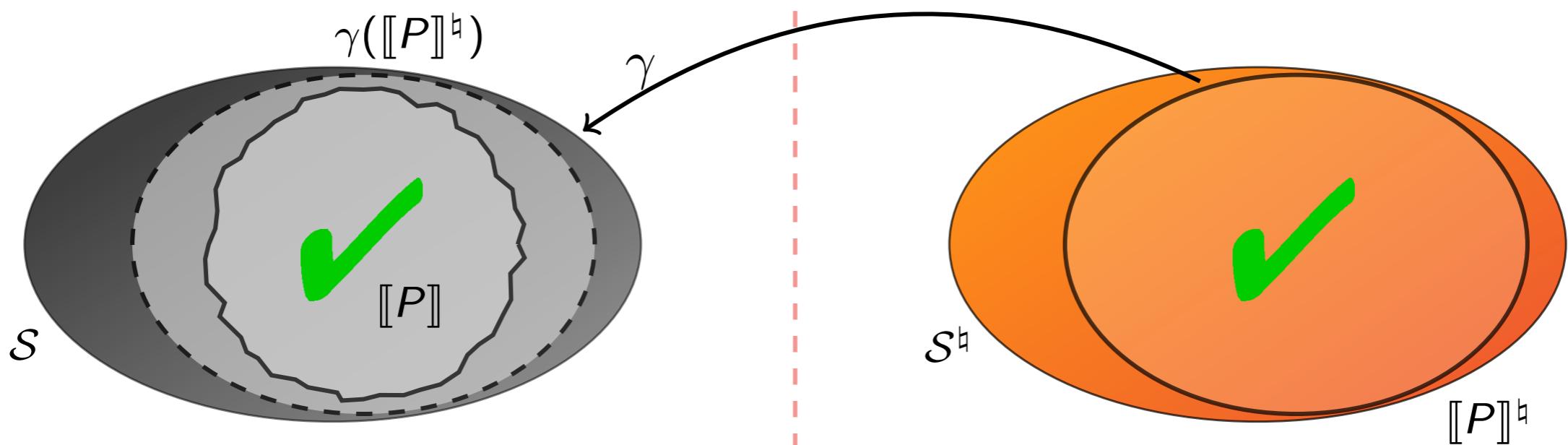


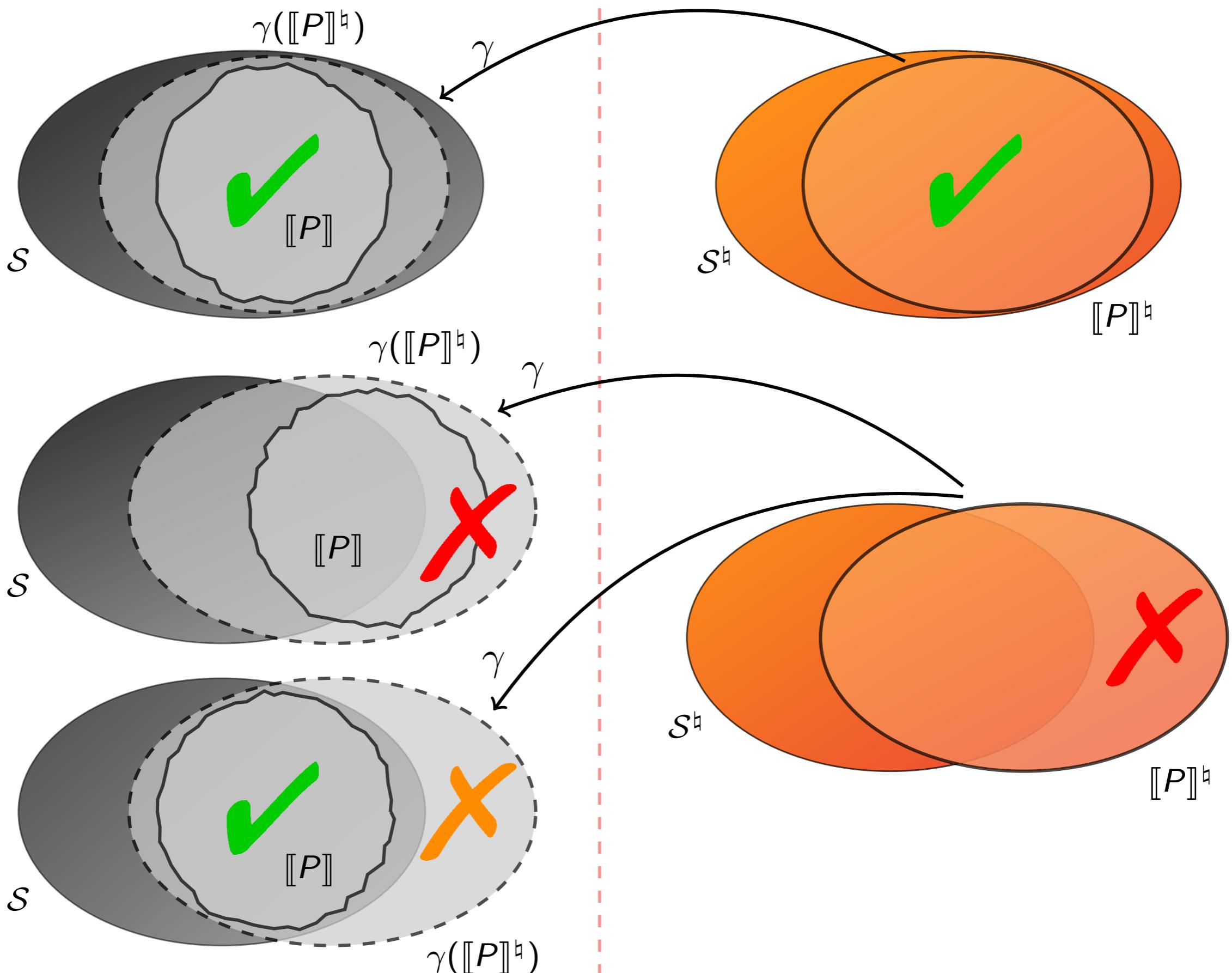
Cousot&Cousot - *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints* (POPL 1977)

Abstract Interpretation



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





Termination

An Abstract Interpretation Framework for Termination

Patrick Cousot¹
CNRS, Ecole Normale Supérieure, and INRIA, France
Cousot@csail.mit.edu, USA
cousot@inria.fr, pcousot@cnrs.fr

Radhia Cousot²
CNRS, Ecole Normale Supérieure, and INRIA, France
cousot@inria.fr

Abstract
Proof, verification, and analysis methods for termination often rely on two main approaches: (1) static analysis methods for induction on data structures or programs towards the end; and (2) some form of induction on the program iteration.

The abstract interpretation design principle is best illustrated by the following diagram showing forward and backward proof, verification and analysis methods for safety. The safety-collecting semantics defining the strongest safety property of programs is then expressed in a constructive fashion (left), while proof and verification methods are based on abstract interpretation (right). By the way, these two approaches are complementary: safety properties such as termination are constructively designed by abstract interpretation (or approximation) to automatically solve safety problems. As far as termination is concerned, the two different approaches are dual: (1) the forward analysis methods are saturated and largely not compatible with each other;

For (2), we show that the forward approach applies mostly well to parallel and infinite termination. The backward verification approach is more compatible with (1) and provides a formal proof for termination of various programs. Its abstraction allows for generalization: reuse, proof, verification and static analysis methods by induction on both program iteration, as well as termination. Finally, we show how the abstract interpretation (or Floyd's) backward analysis of termination of programs can be used to prove termination of loop variants as well as nested loops, Russell's theorem on iterative fixed point method, and Peleg-Tochino-Kozen's termination theorems.

Categories and Subject Descriptors D.2.4 [Symbolic/Program Semantics]; F.3.1 [Formal Definitions]; F.3.1 [Open Problems and Definitions and Requirements about Programs].
General Terms Languages, Reliability, Security, Theory, Verification.
Keywords Abstract Interpretation, Induction, Proof, Safety, Static Analysis, Termination Functions, Verification, Termination.

1. Introduction
Proof/Verification/program-proof methods for insurance and verification [24, 25] and termination analysis [26, 27] are now well developed and widely used.

For static insurance analysis by abstract interpretation [28, 29], a key step is to capture the strongest invariant as a fixpoint and then to approximate this strongest invariant to automatically infer an abstract inductive invariant using the constructive fixpoint approximation methods.

For termination analysis, the discovery of certain functions is either done directly as limited cover [29] or else is based on the Floyd/Turing idea of various functions being well-founded semi-computable functions.

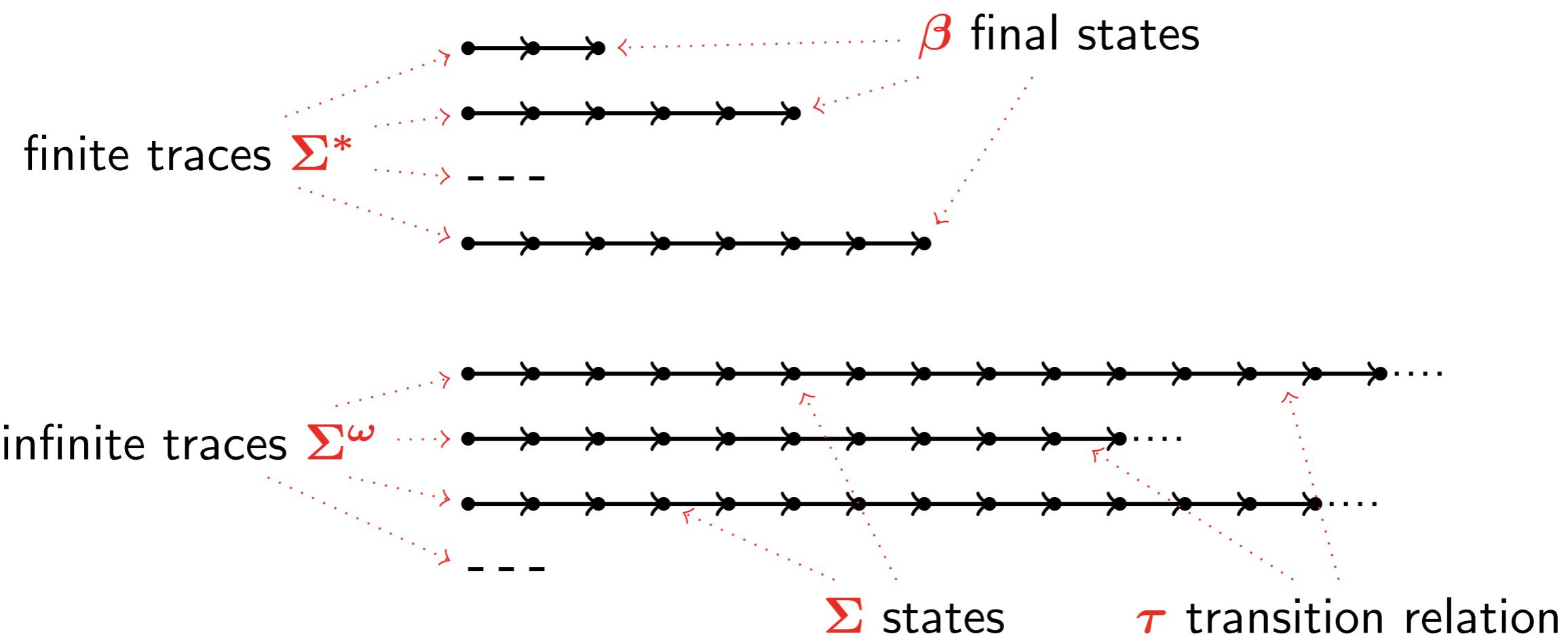
¹Work supported in part by the CNRS/SFRD Collaboration in Computing award 2009-2010.

²Previous to 2008, Radhia Cousot was a member of the team of the University of Illinois at Urbana-Champaign. She is grateful to her former colleagues for their support and contributions on a project without the provided tools are not made in this document. Her current address is the Institut Élie Cartan de Lorraine, Université de Lorraine, Nancy, France. This paper is an extended version of the one published in the first page of the conference, in which the authors are grateful to the editor and to the referees for their important contributions and help.

ACM SIGART Newsletter, Volume 1, Number 1, March 2012, Article 10, ISSN 2153-235X, DOI 10.1145/2132222.2132223, 10 pages.



program \mapsto maximal trace semantics



Termination Semantics

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

$$\mathcal{T}_t \in \Sigma \rightarrow \emptyset$$

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

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

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

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

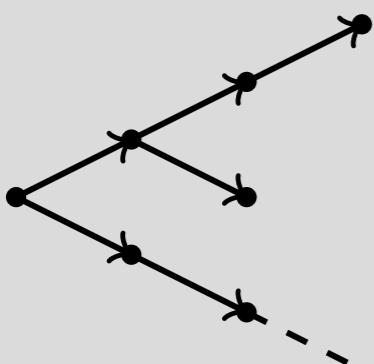
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

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

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

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

Example



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

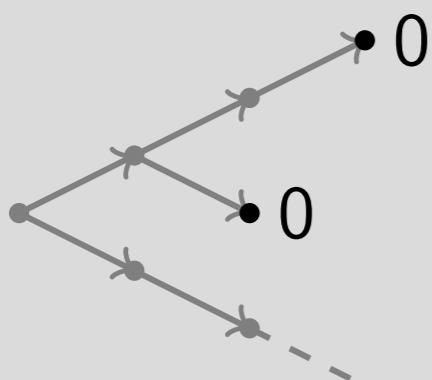
$$\mathcal{T}_t \in \Sigma \rightarrow \emptyset$$

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

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

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

Example



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

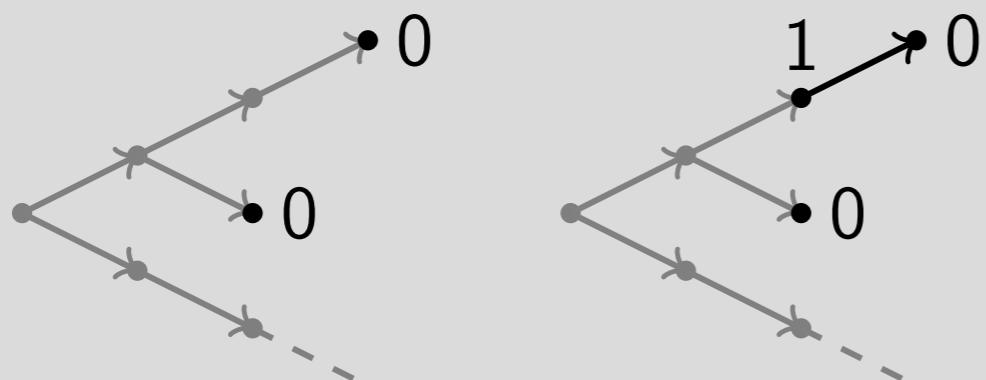
$$\mathcal{T}_t \in \Sigma \rightarrow \emptyset$$

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

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

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

Example



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

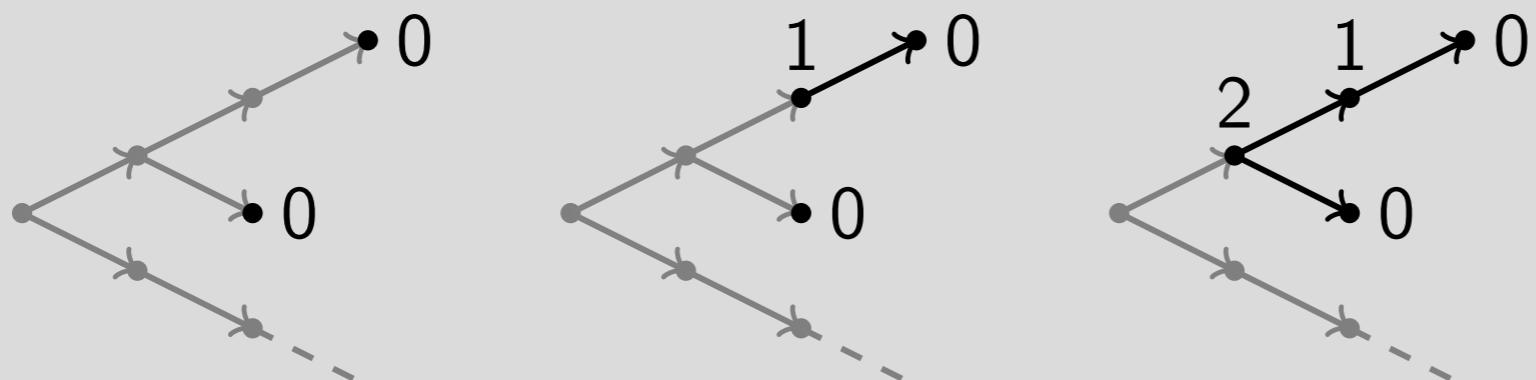
$$\mathcal{T}_t \in \Sigma \rightarrow \mathbb{O}$$

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

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

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

Example



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

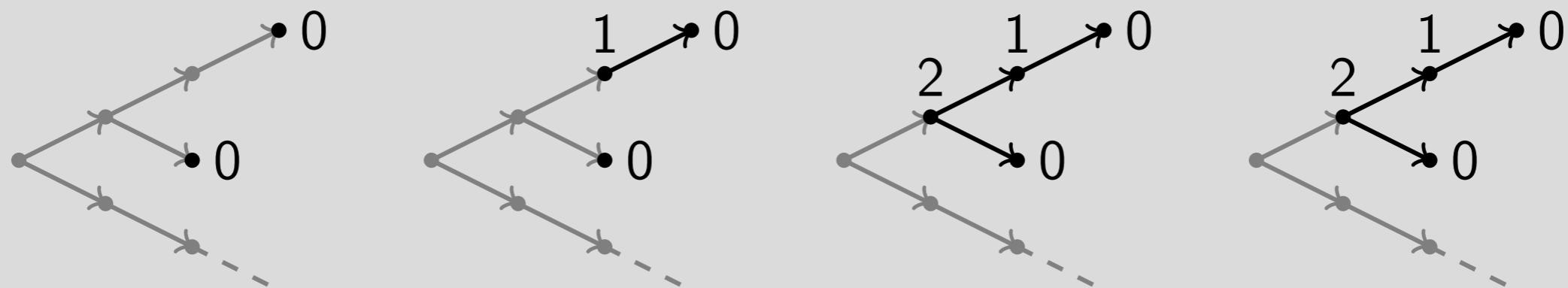
$$\mathcal{T}_t \in \Sigma \rightarrow \emptyset$$

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

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

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

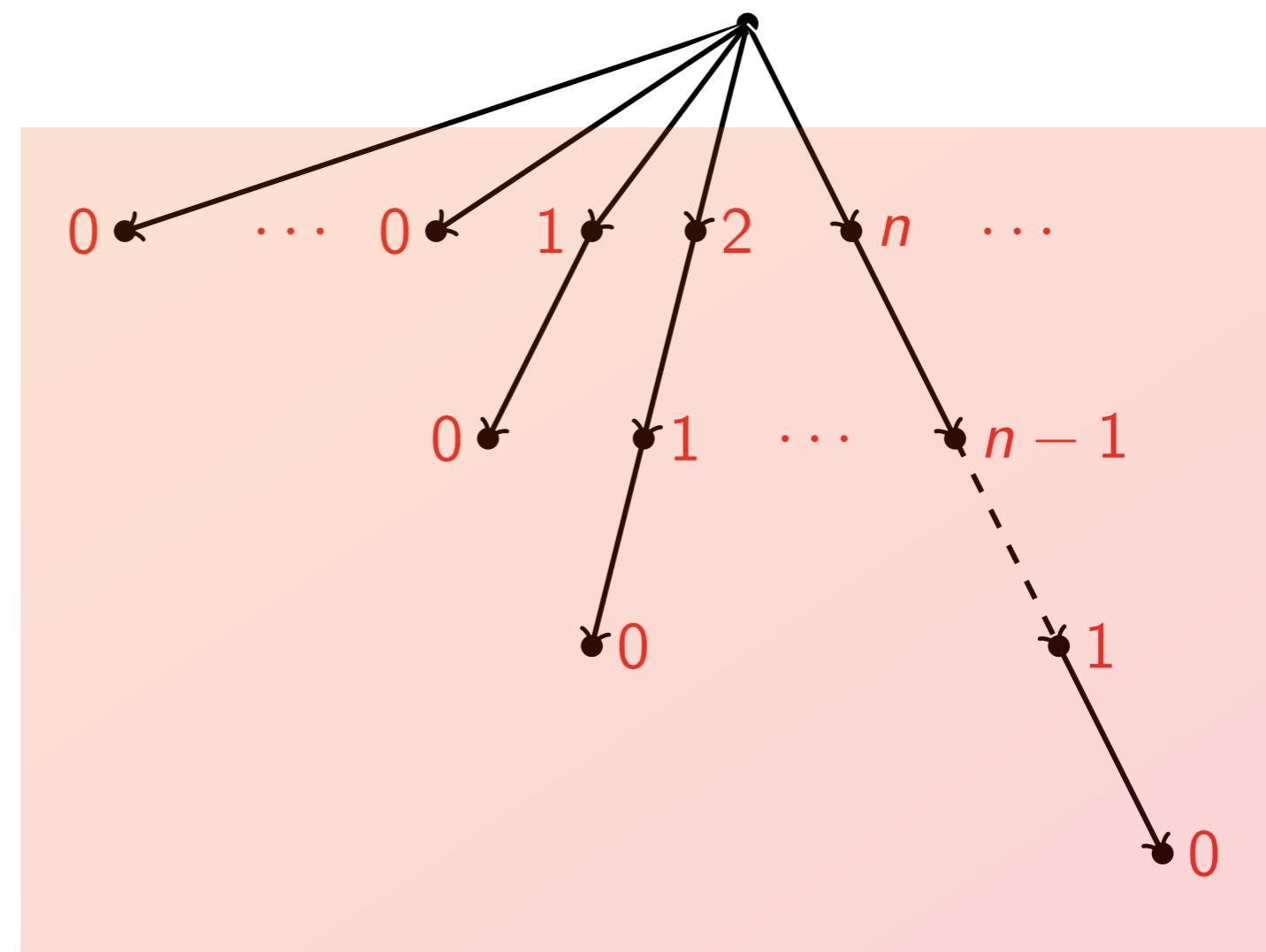
Example



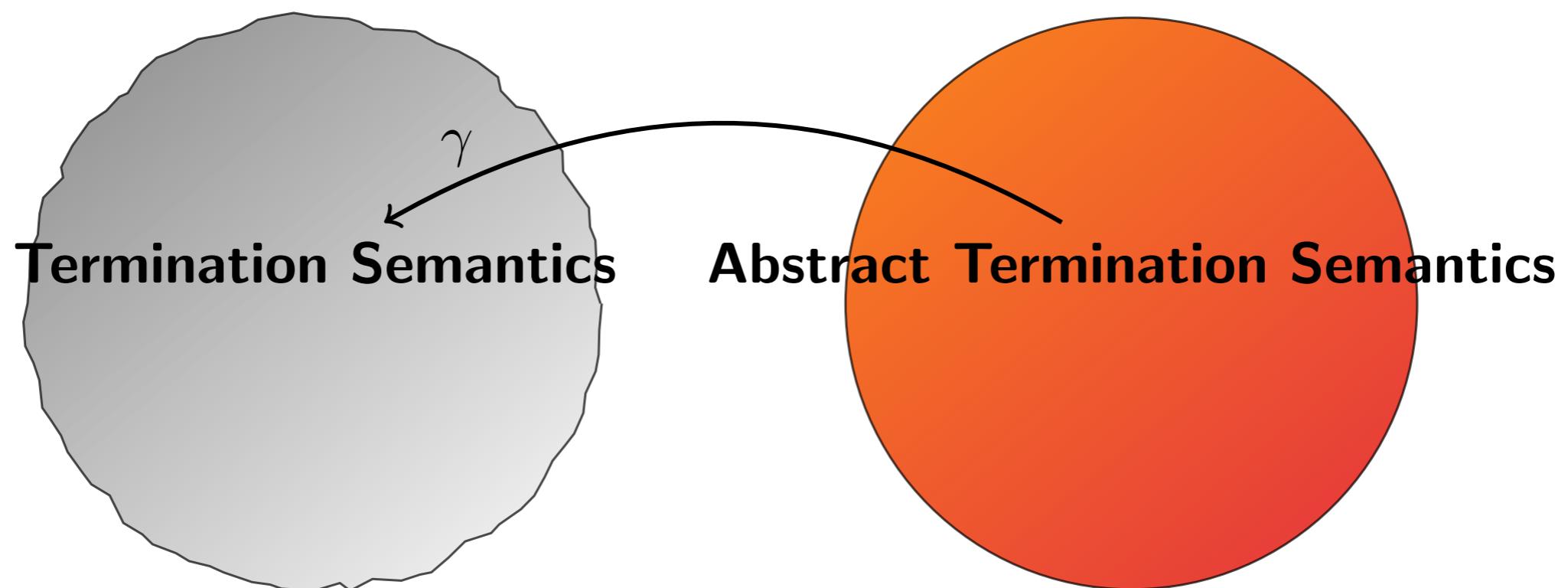
- **remark:** the termination semantics is **not computable!**

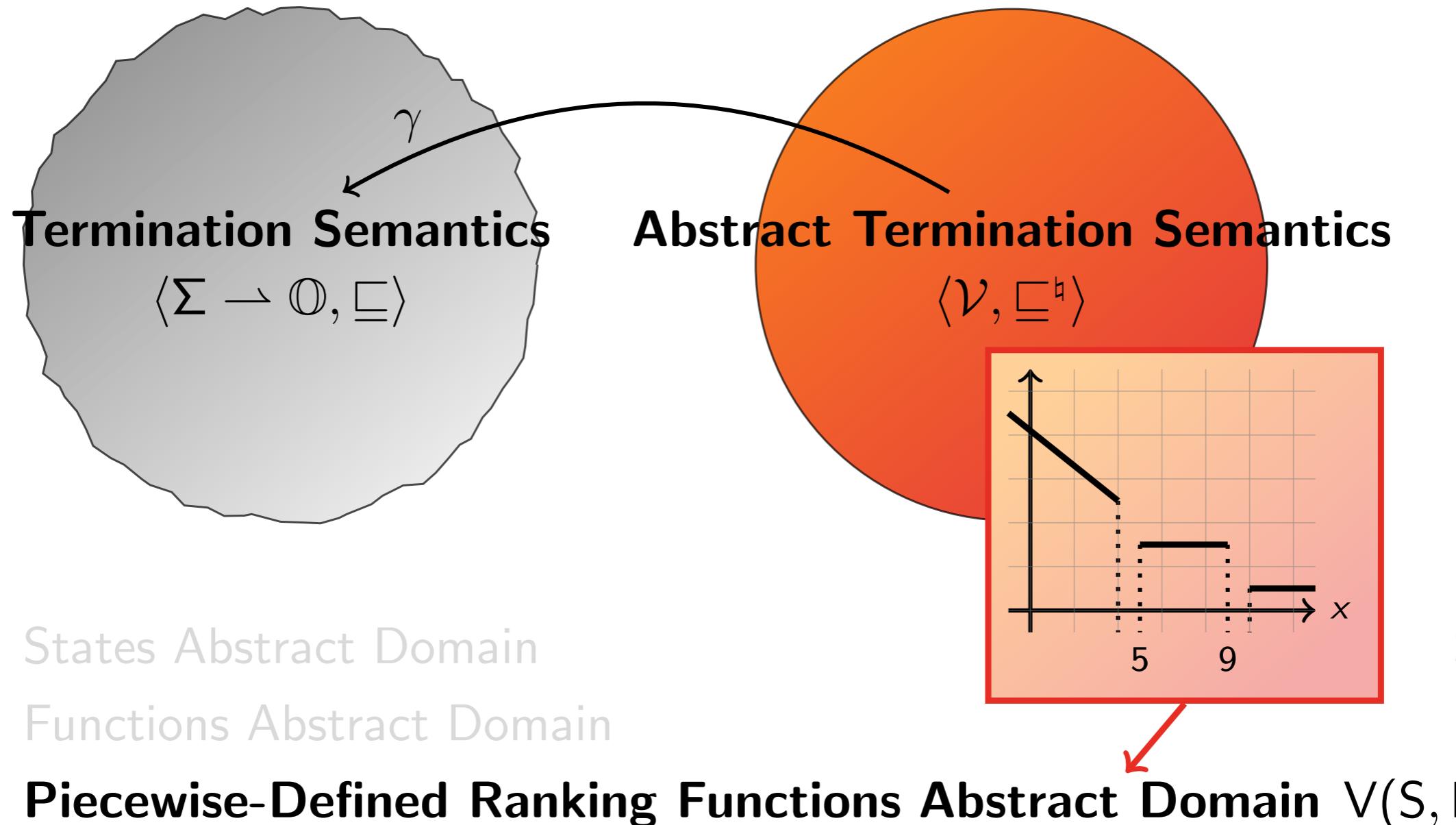
Example

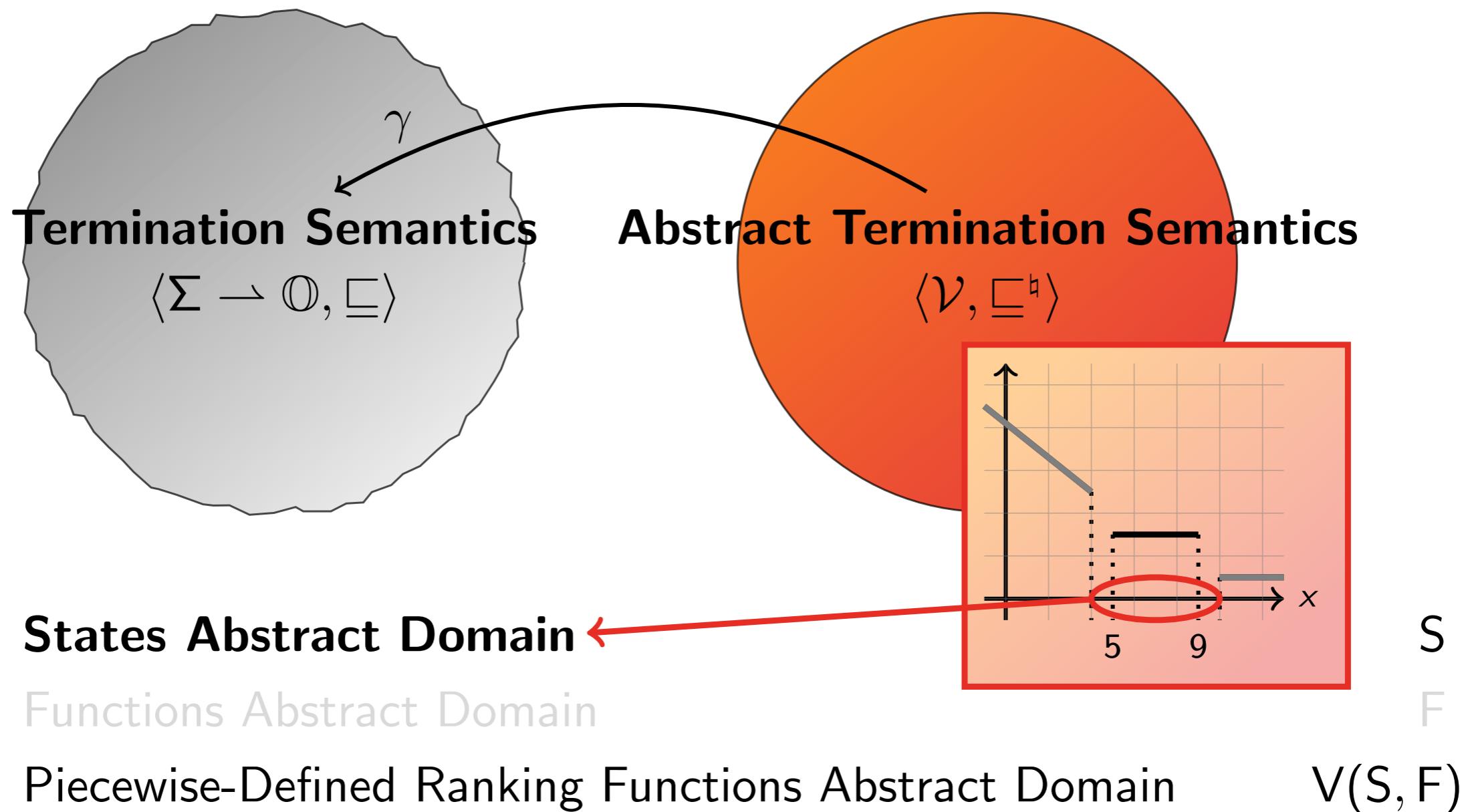
```
int : x
x := ?
while (x > 0) do
    x := x - 1
od
```

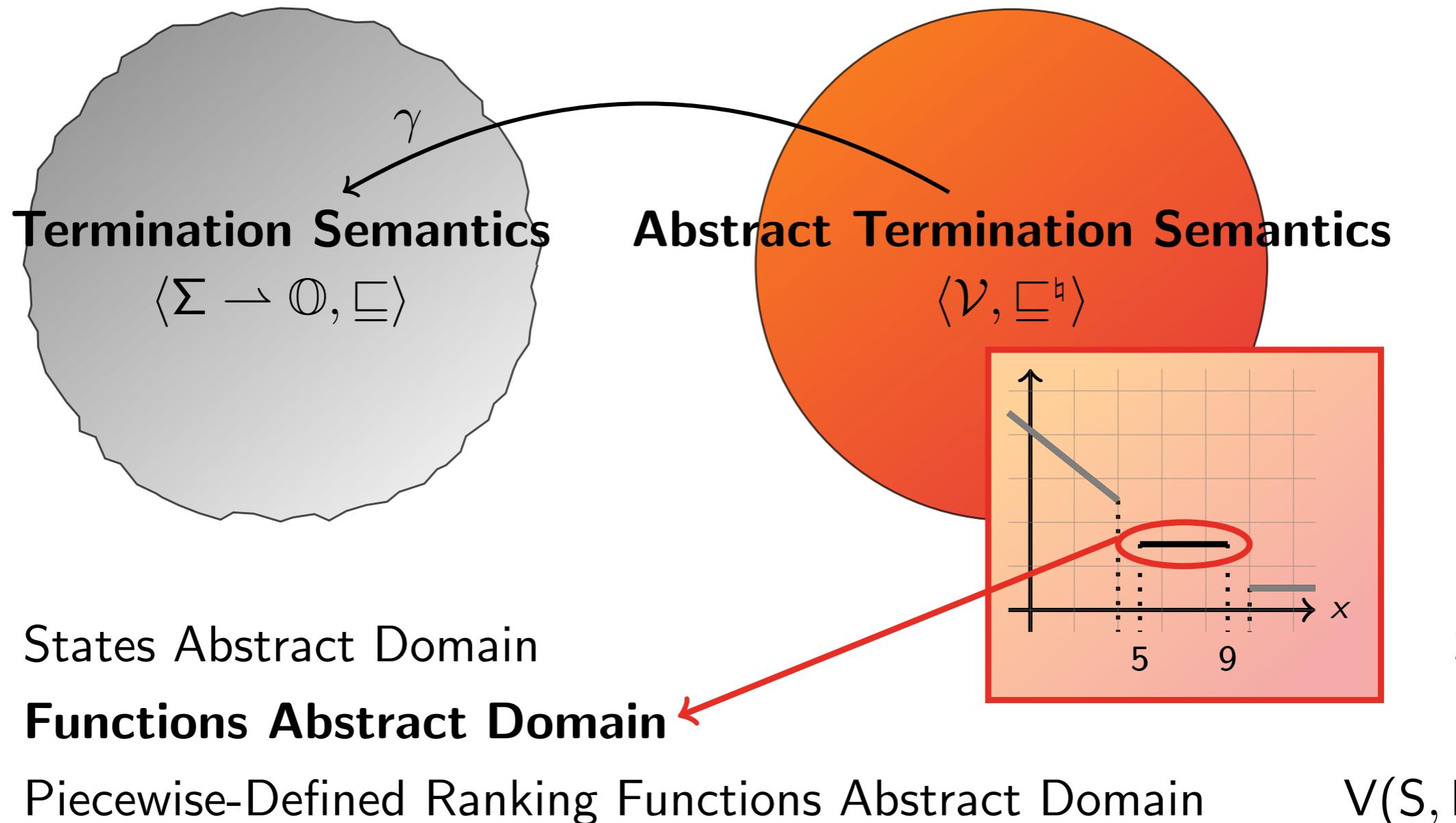


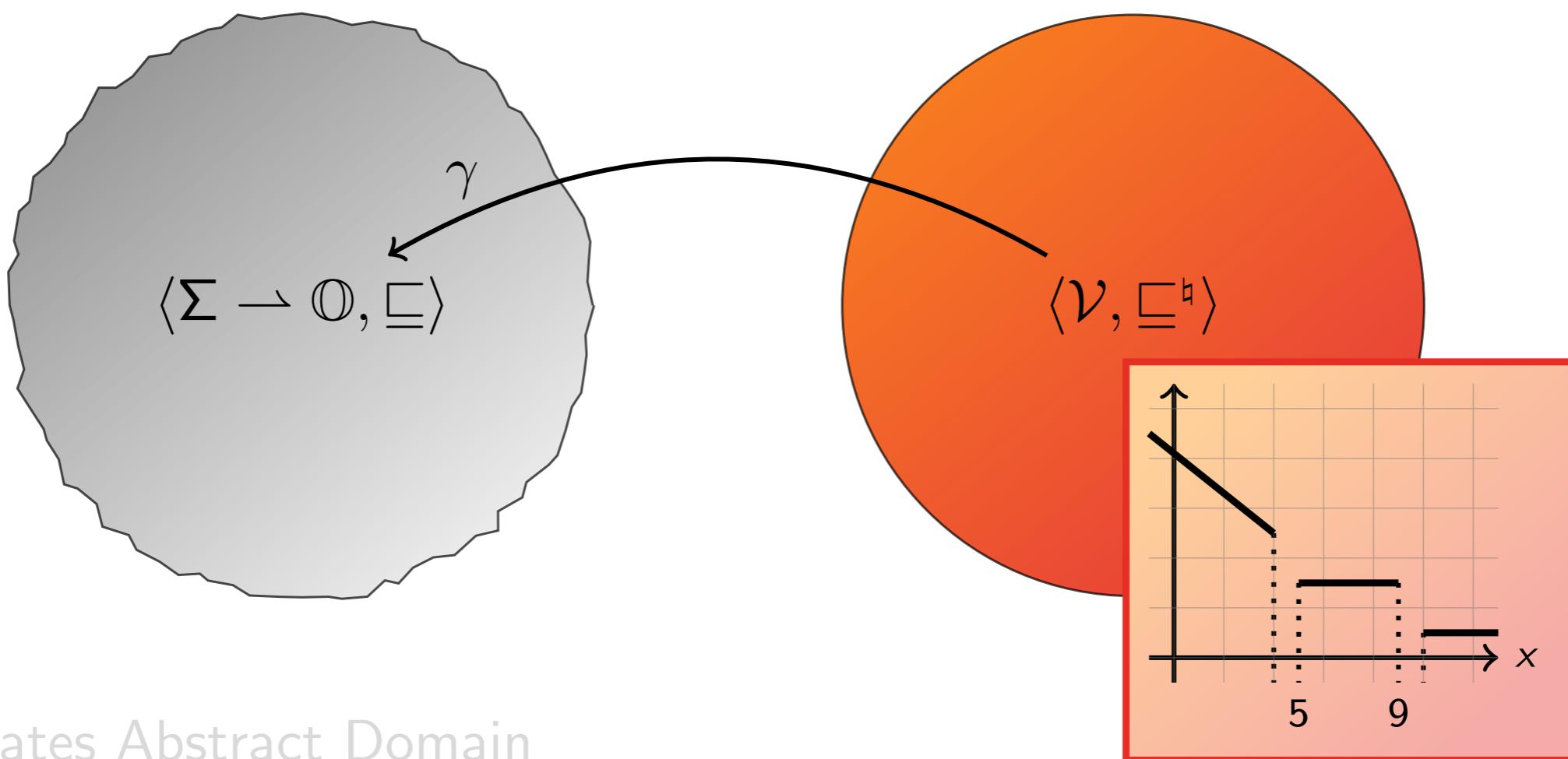
Piecewise-Defined Ranking Functions



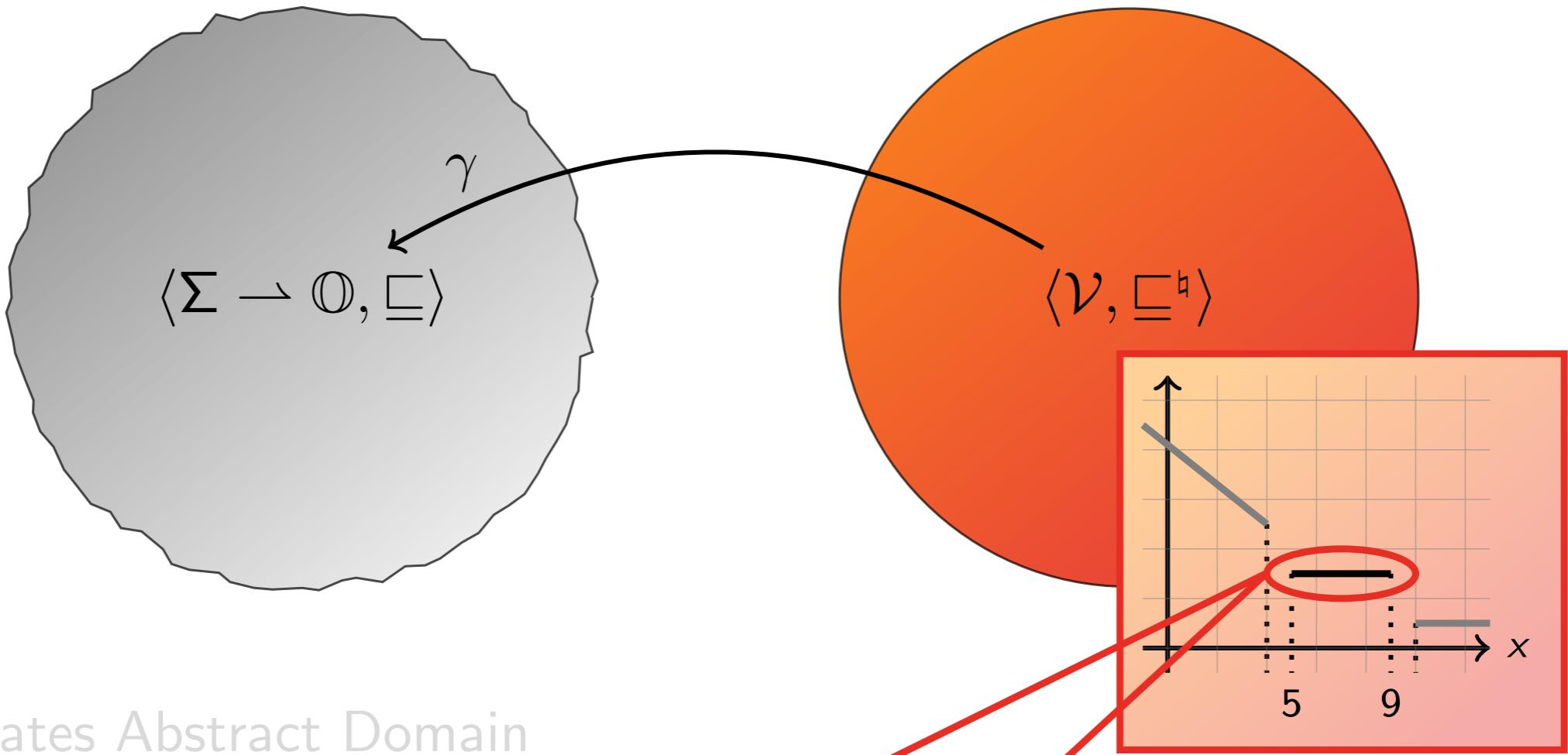




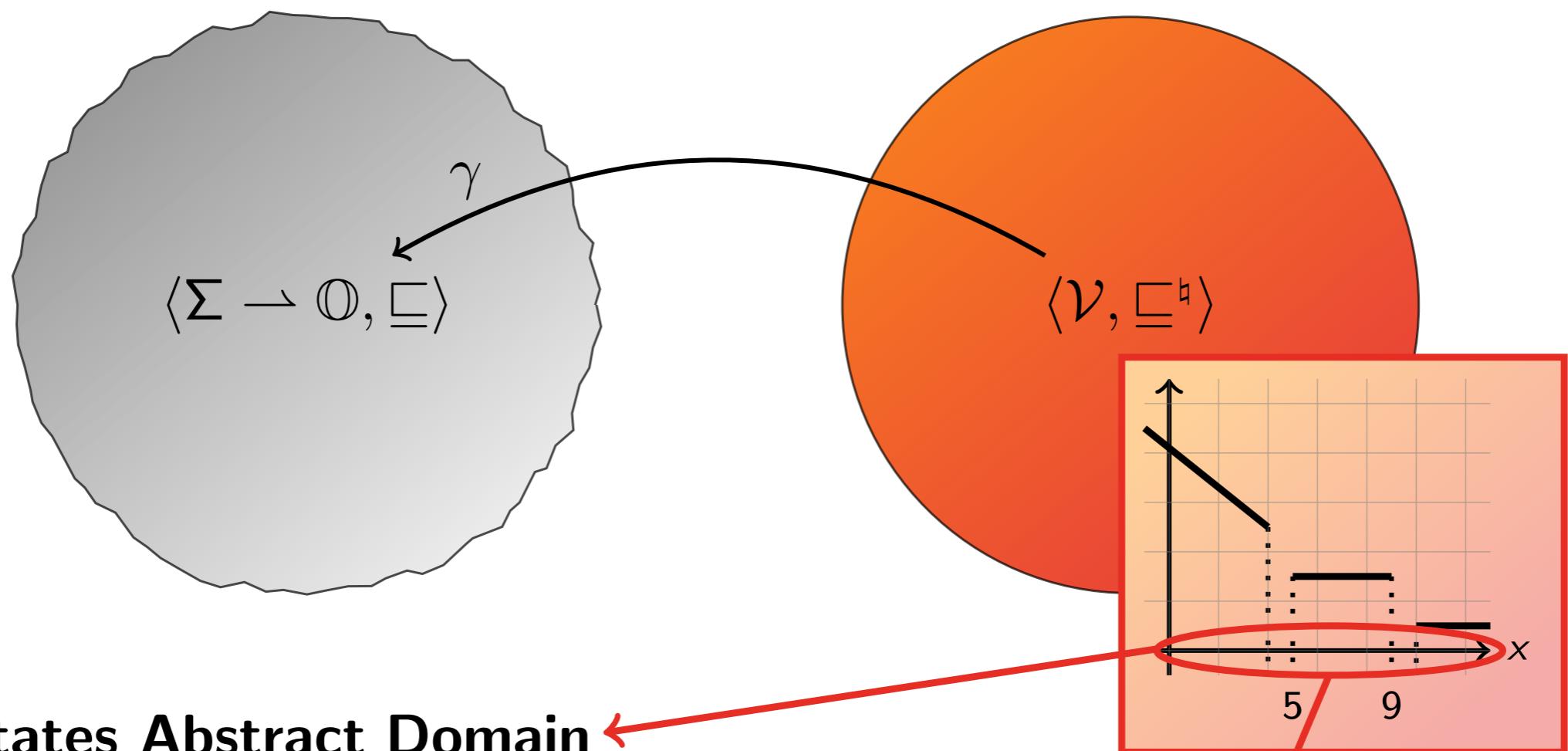




- States Abstract Domain
 - $\mathcal{L} \stackrel{\text{def}}{=} \text{Interval/Octagonal/Polyhedral Linear Constraints}$
- Functions Abstract Domain
 - $\mathcal{F} \stackrel{\text{def}}{=} \{\perp\} \cup \{f \mid f \in \mathbb{Z}^n \rightarrow \mathbb{N}\} \cup \{\top\}$
where $f \equiv f(x_1, \dots,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

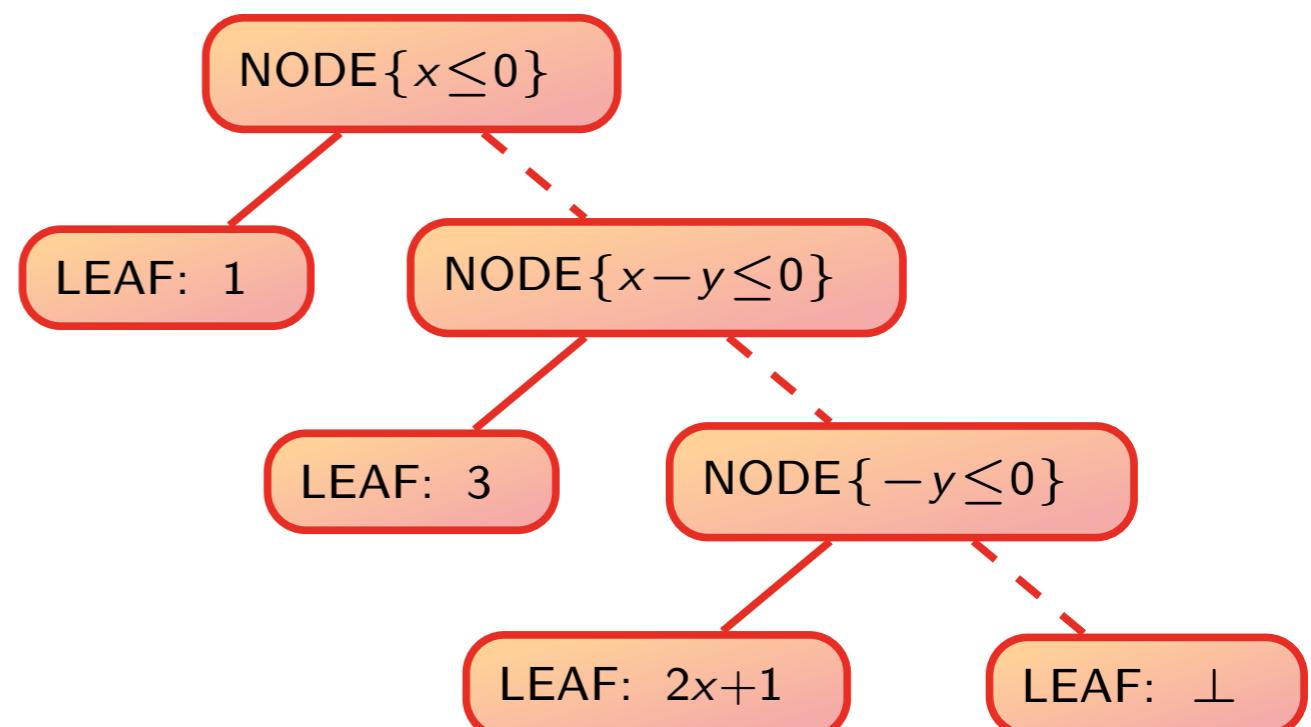
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

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

Example

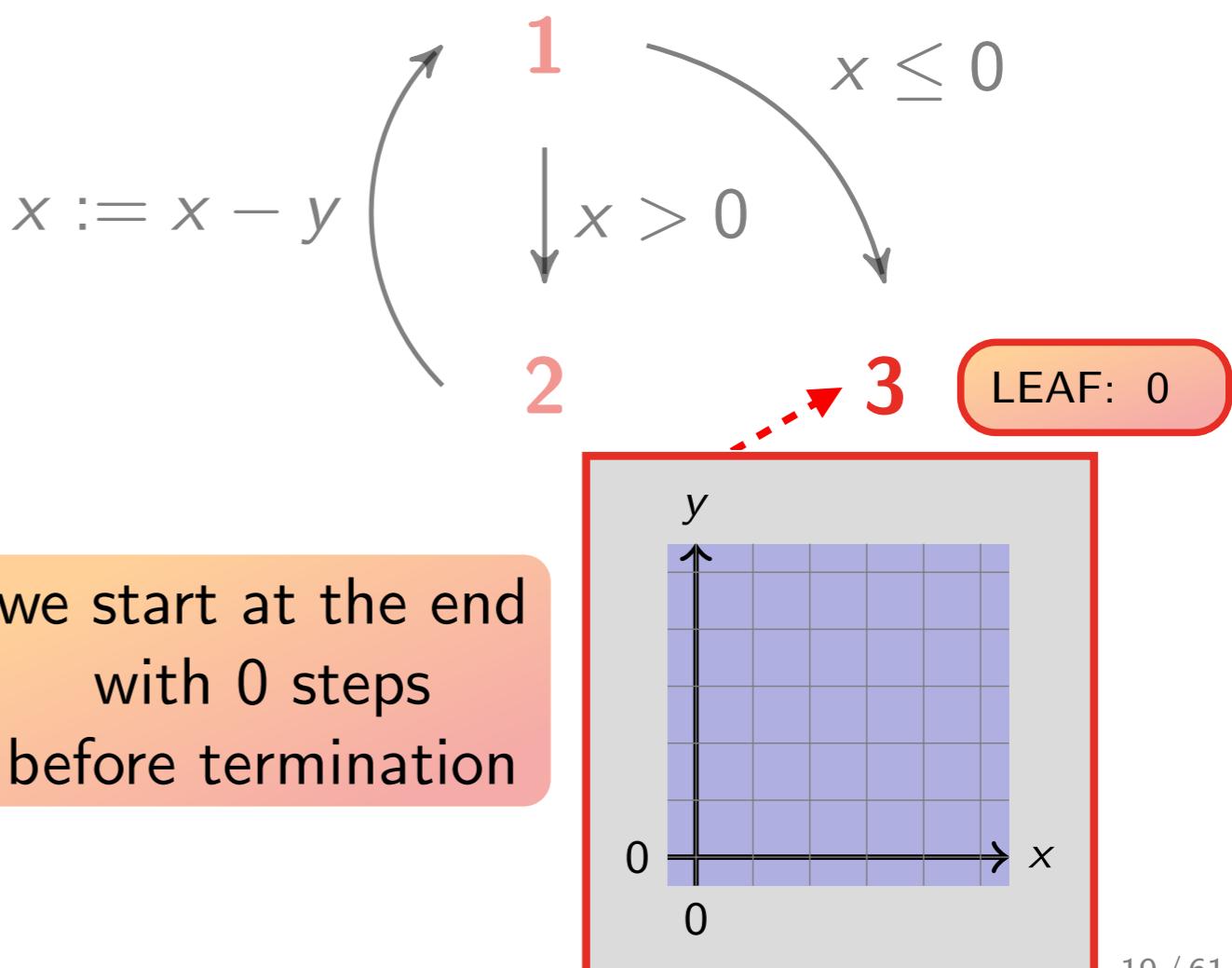
```
int : x, y
while 1(x > 0) do
    2x := x - y
od3
```

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



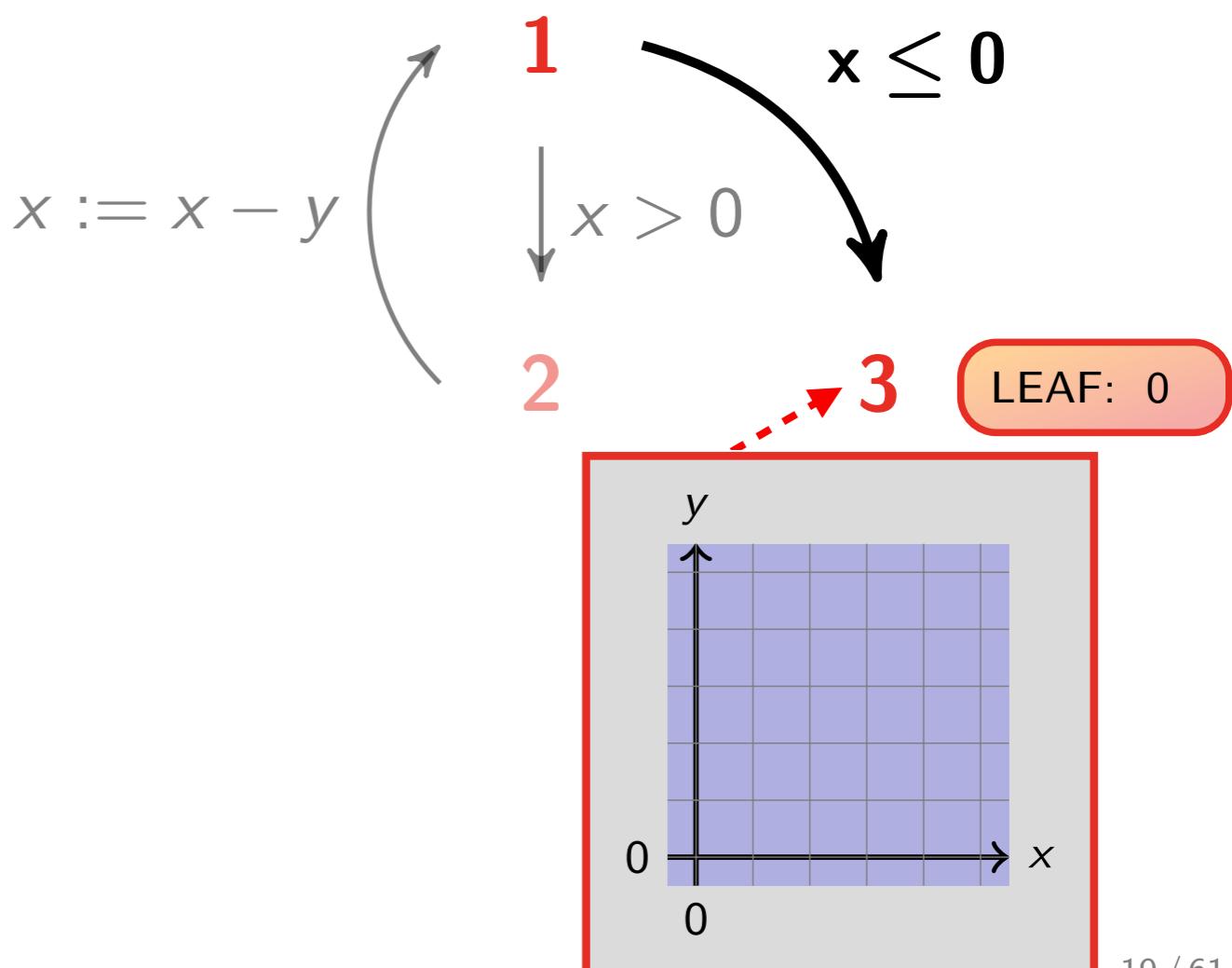
Example

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

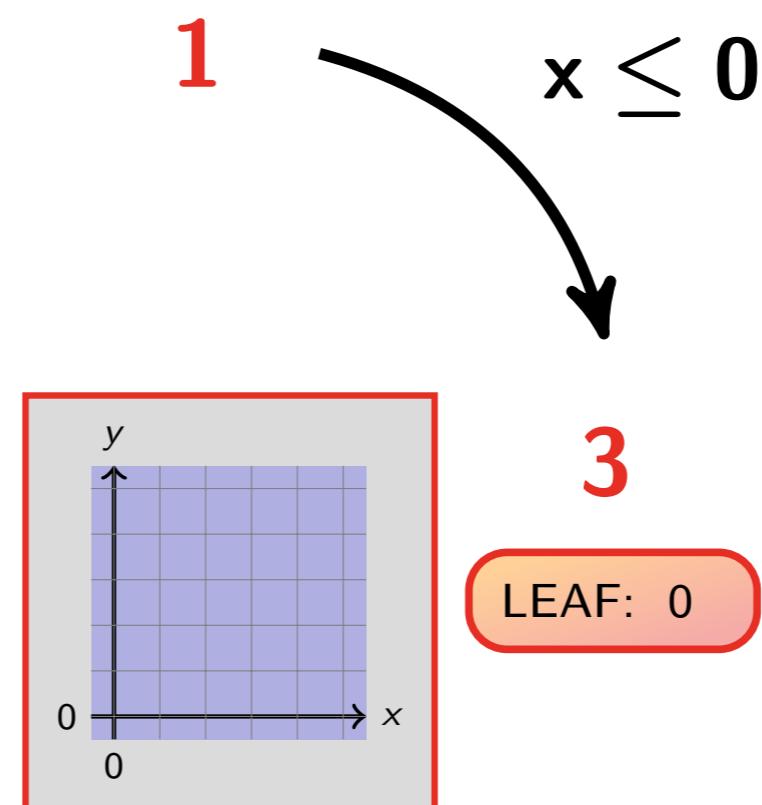


Example

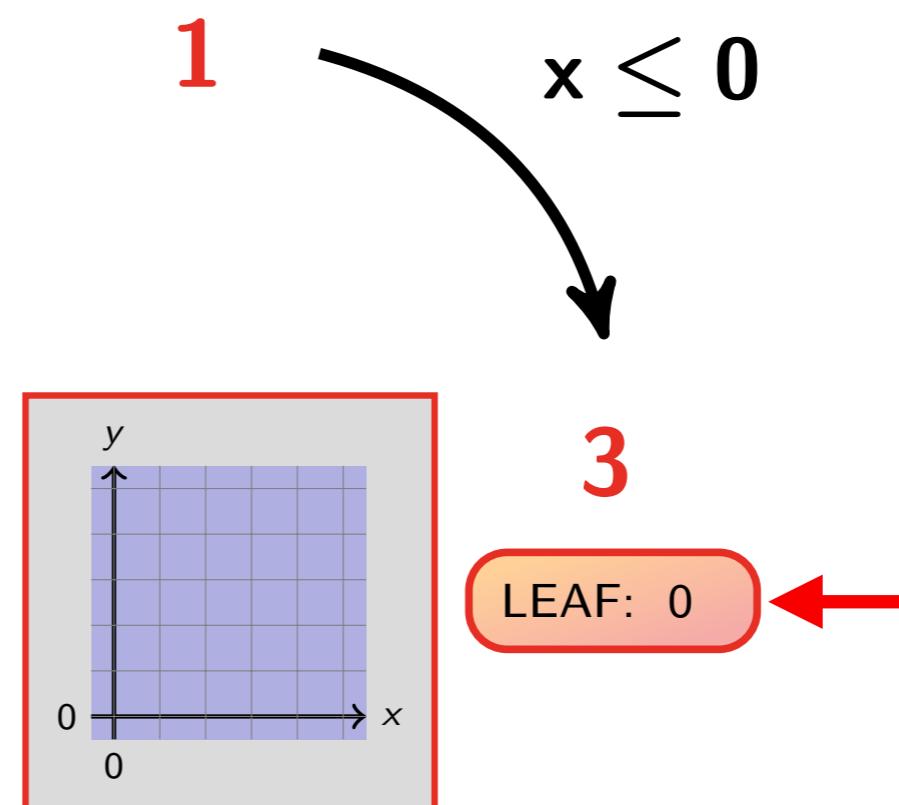
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



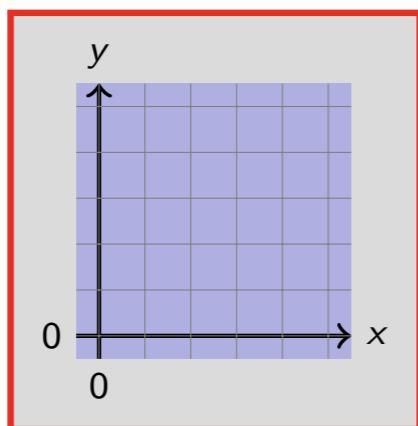
Tests



Tests

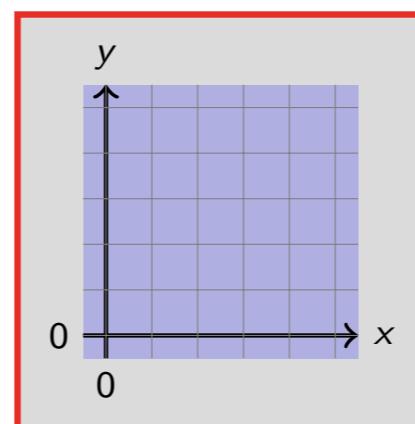


Tests



LEAF: 1

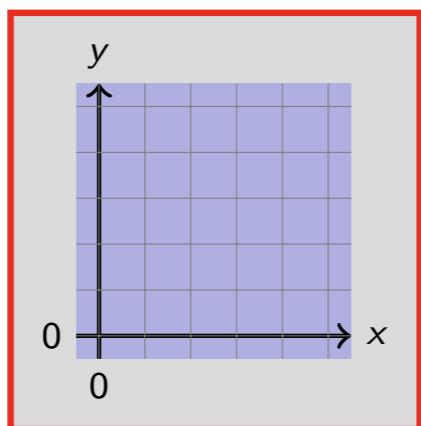
1 $x \leq 0$



3

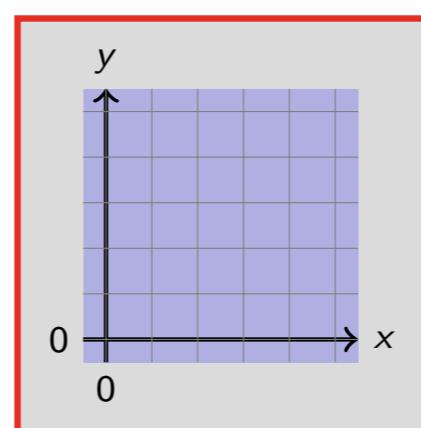
LEAF: 0

Tests



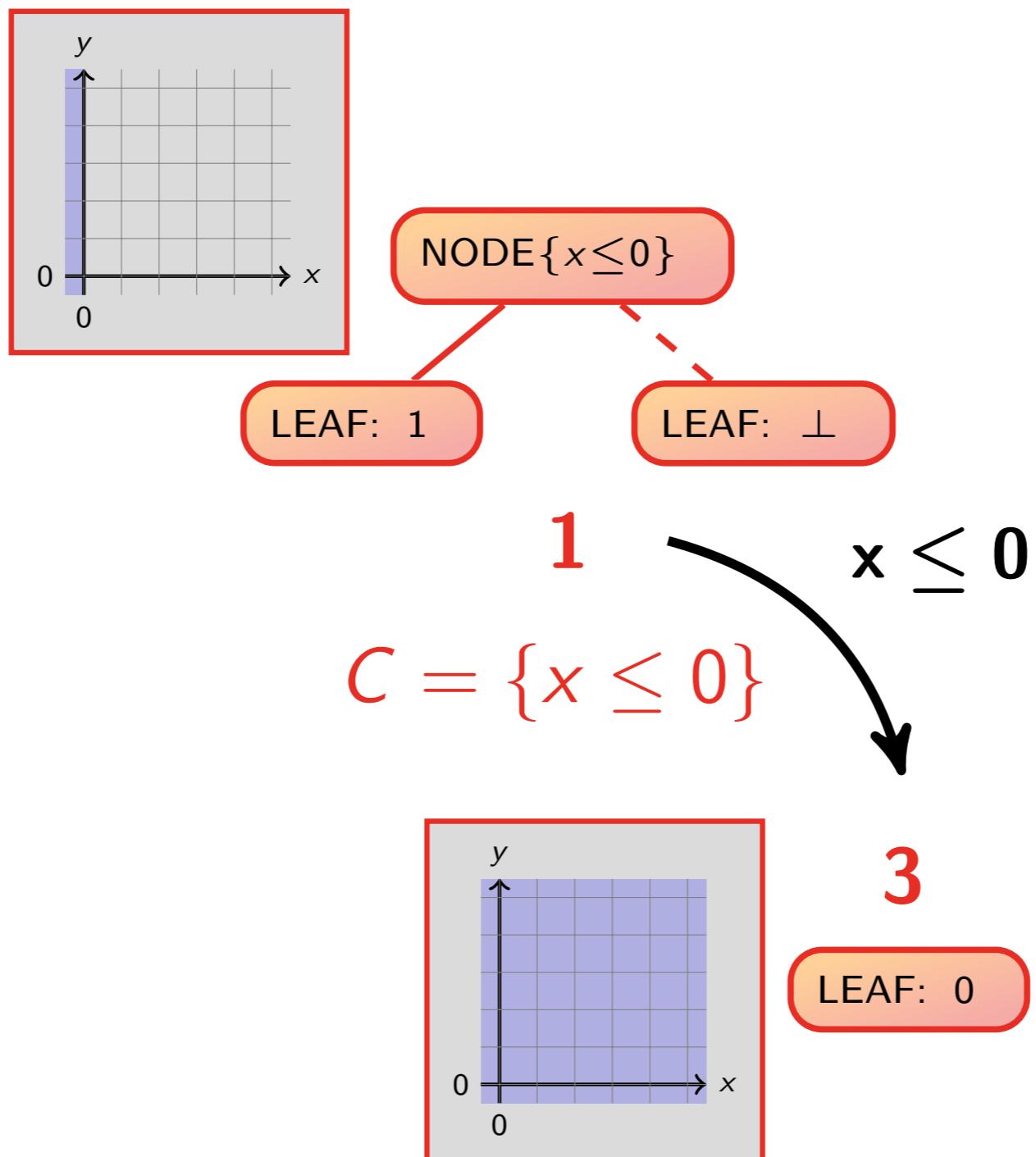
LEAF: 1

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

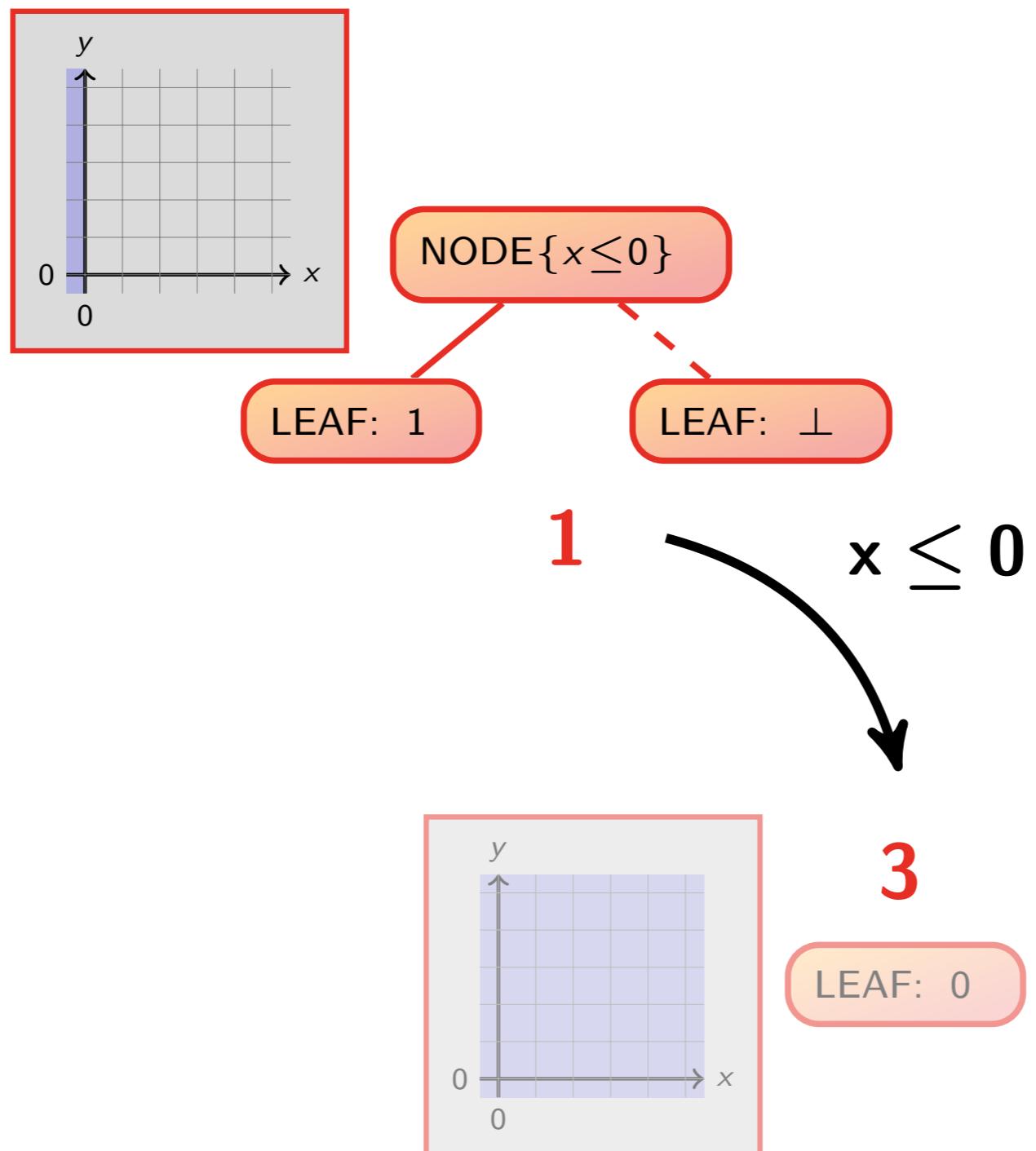


3
LEAF: 0

Tests

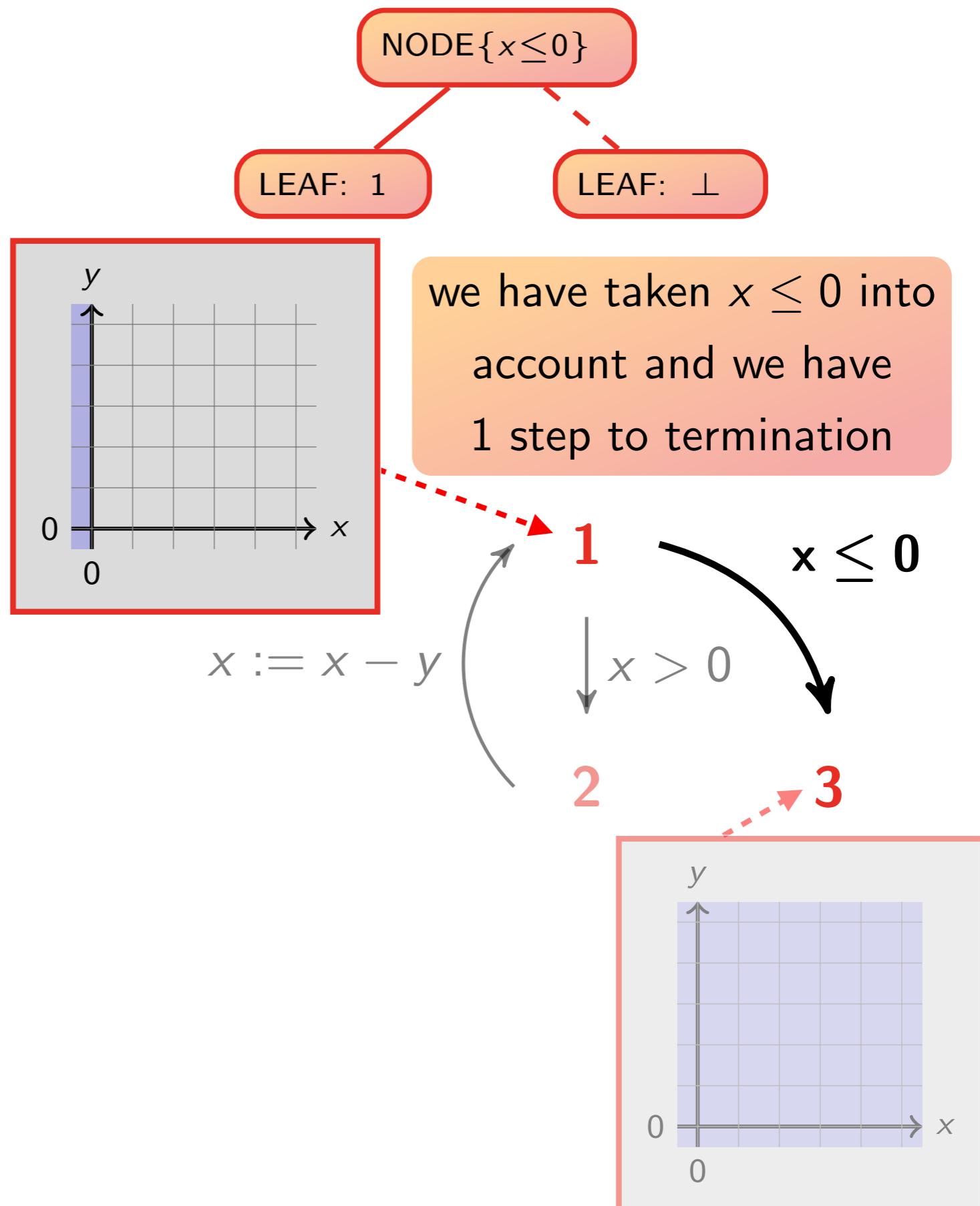


Tests



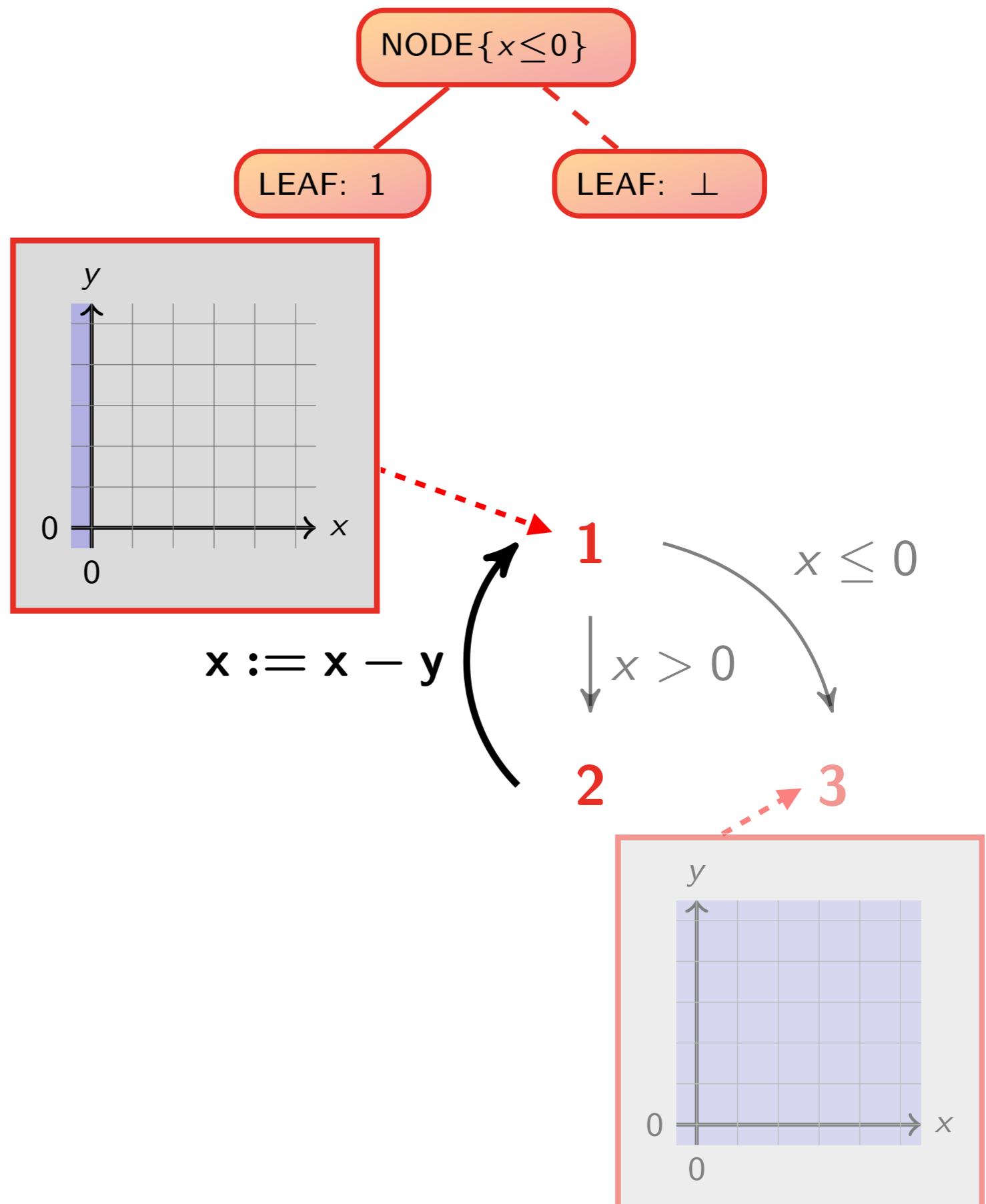
Example

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

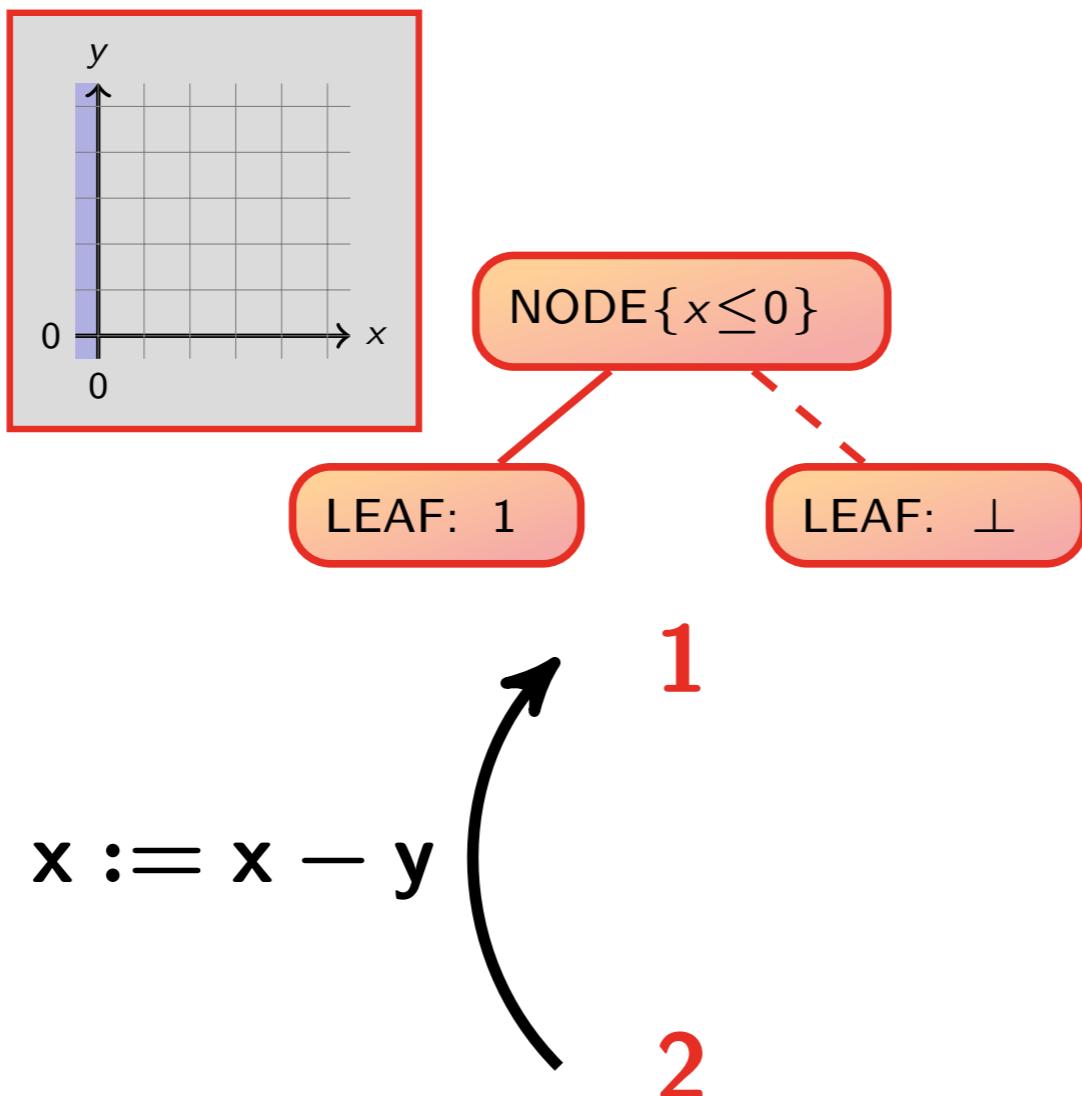


Example

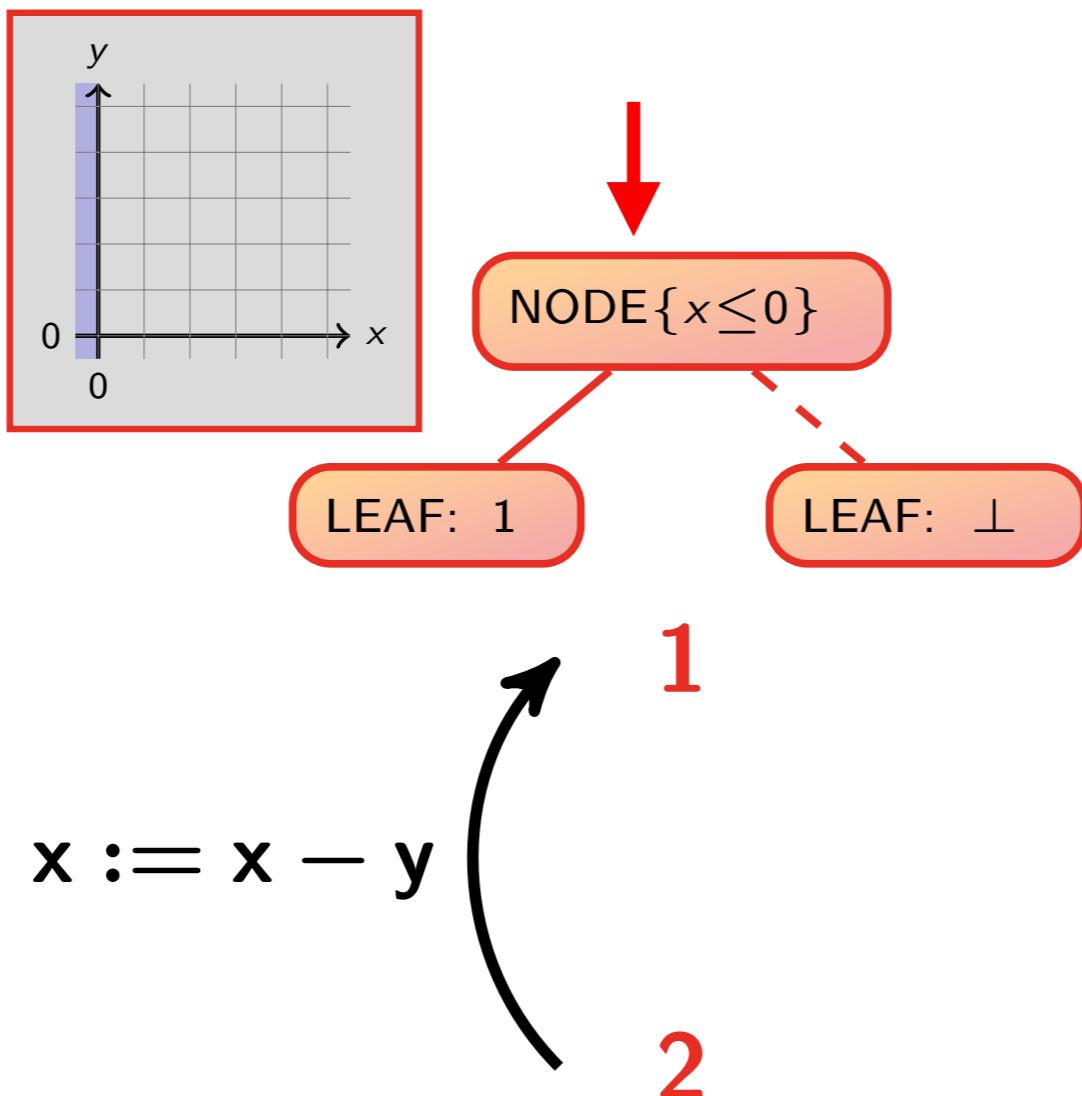
```
int : x, y
while 1(x > 0) do
    2x := x - y
od3
```



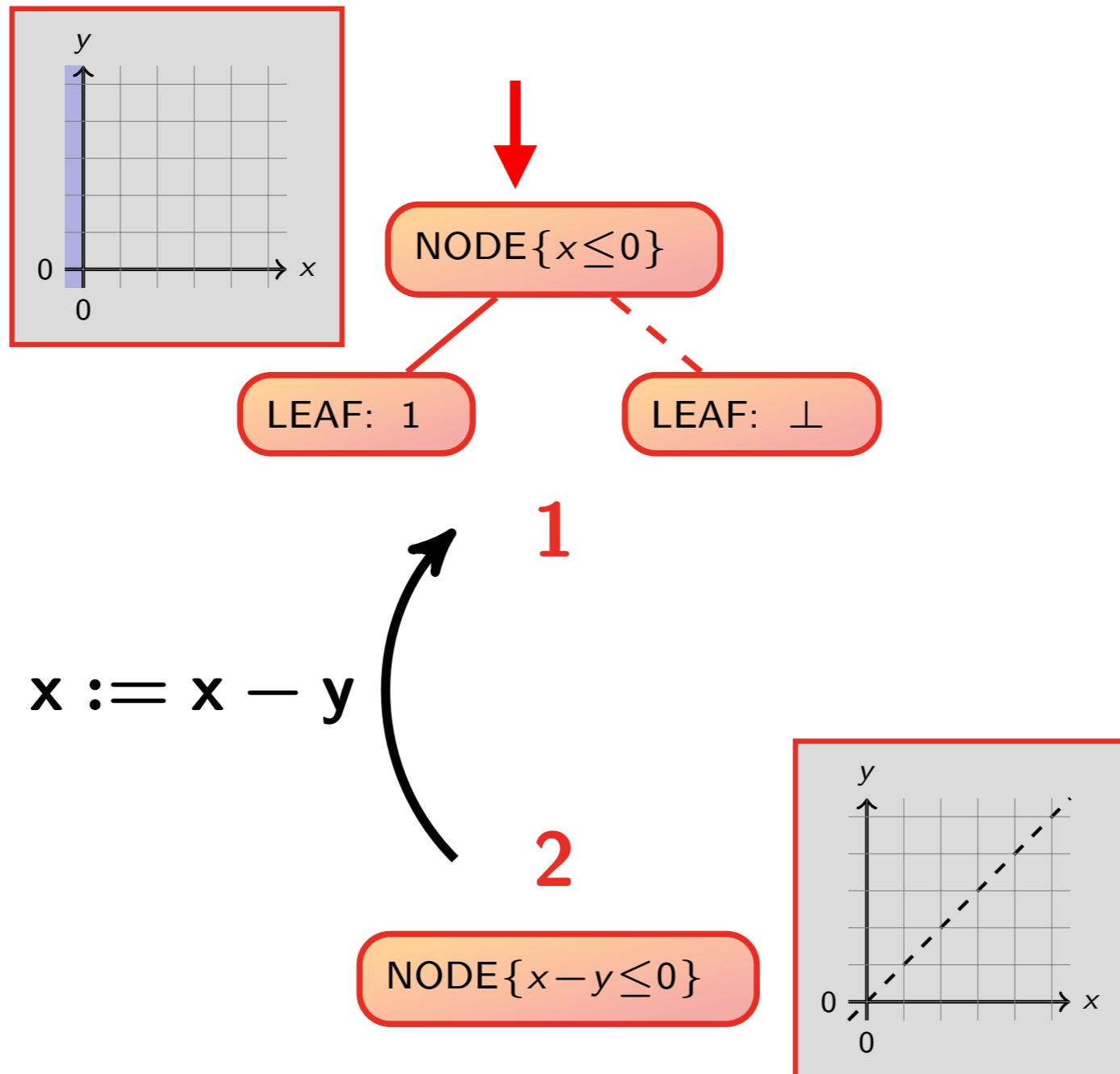
Assignments



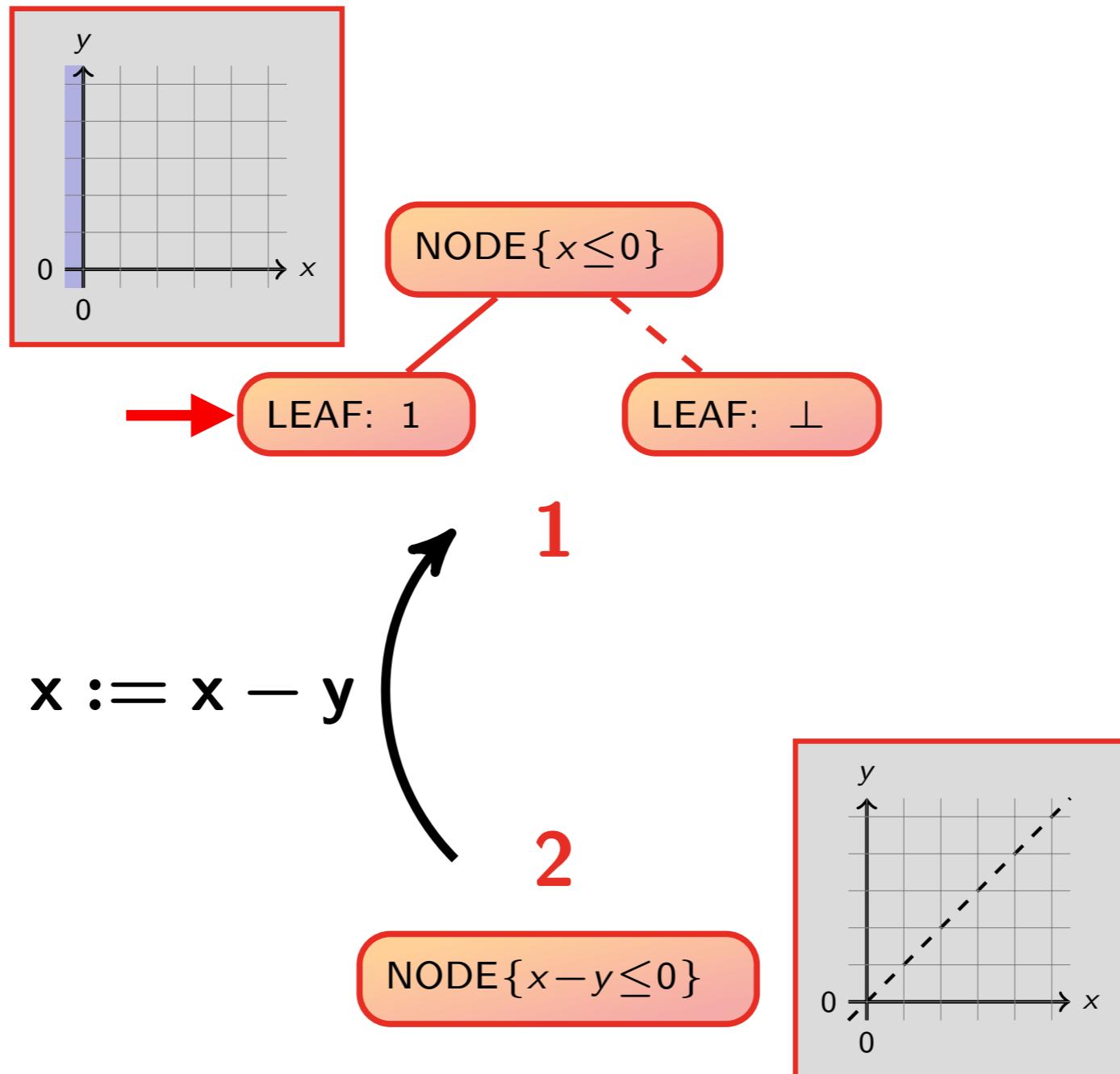
Assignments



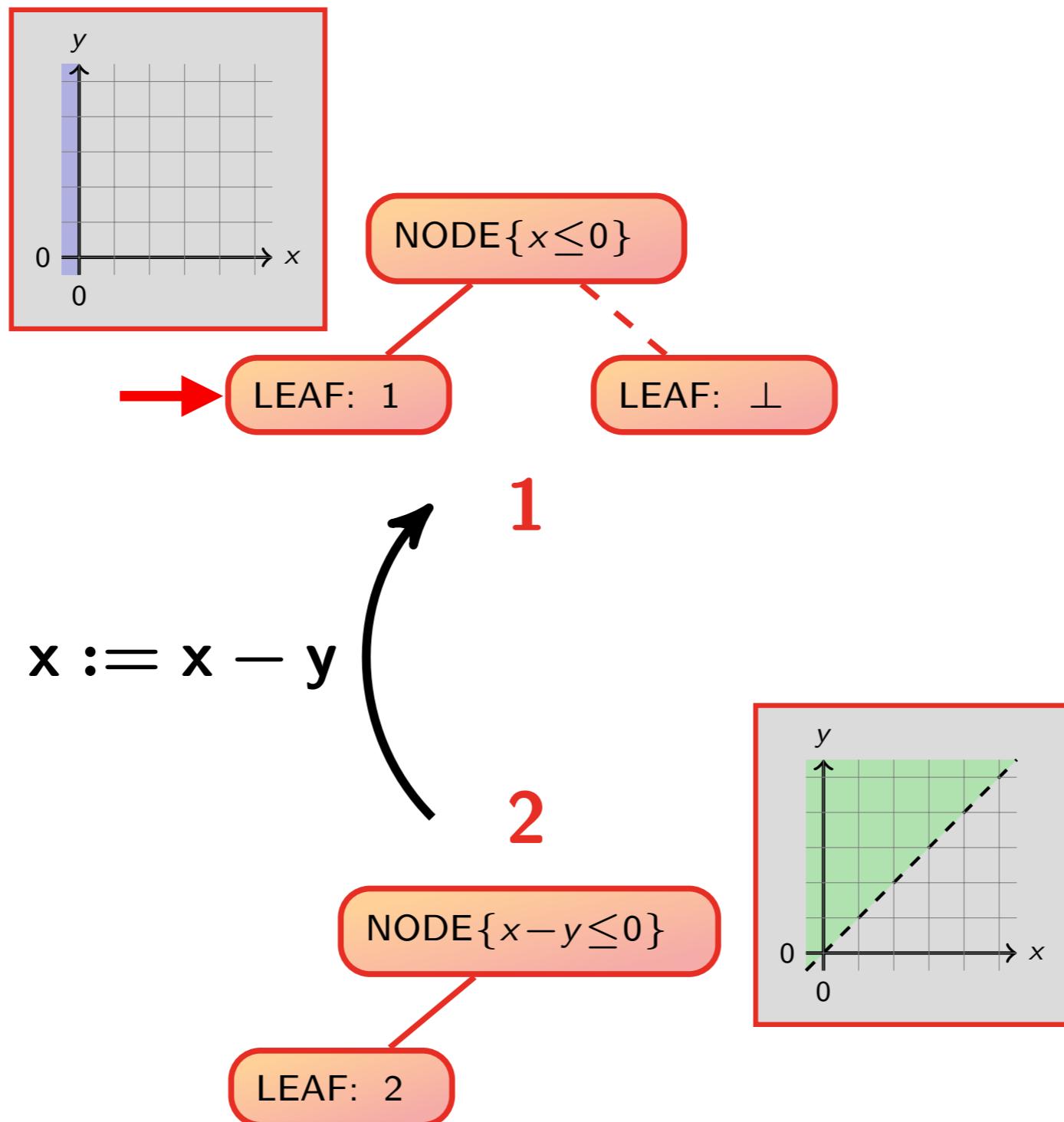
Assignments



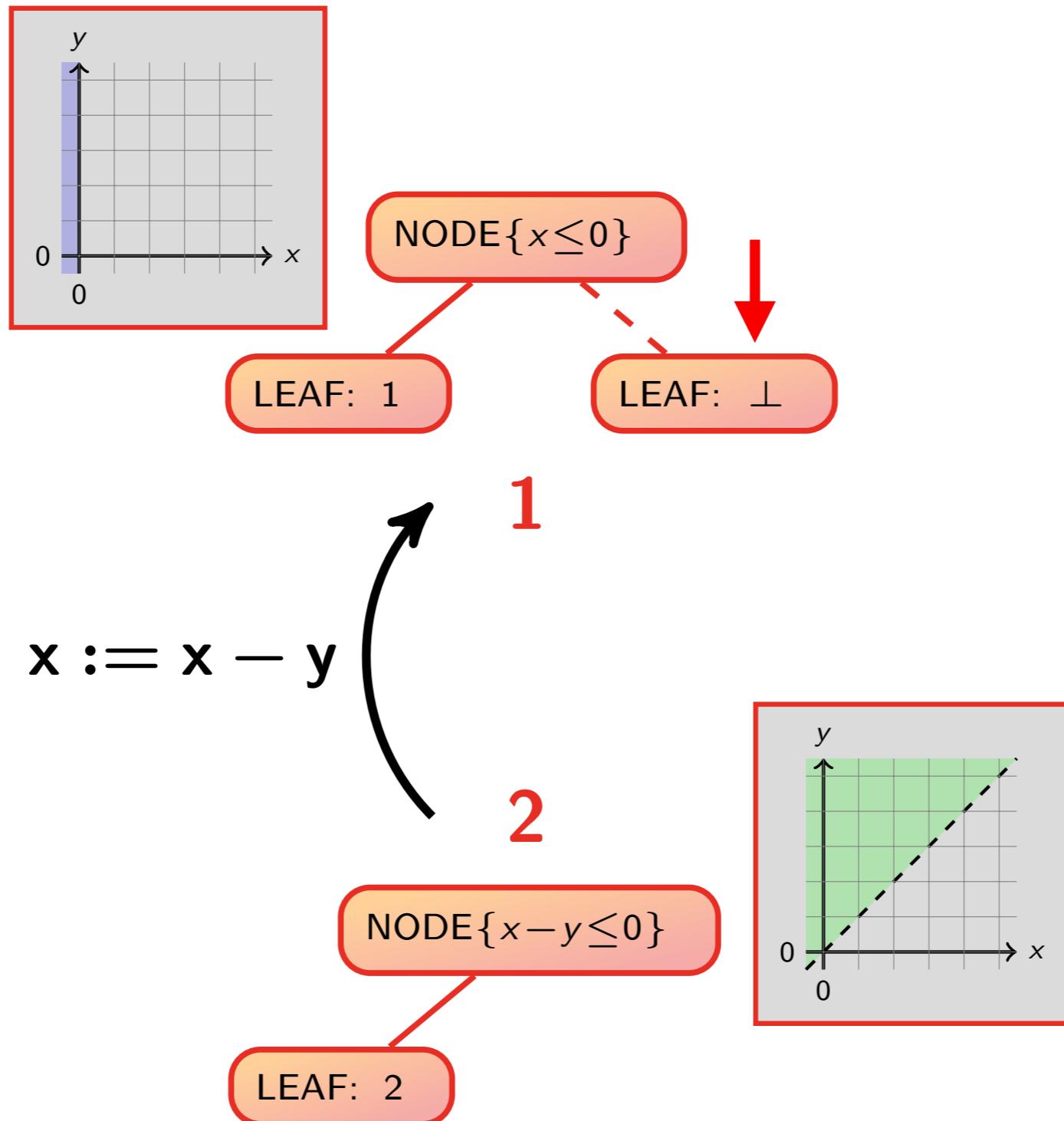
Assignments



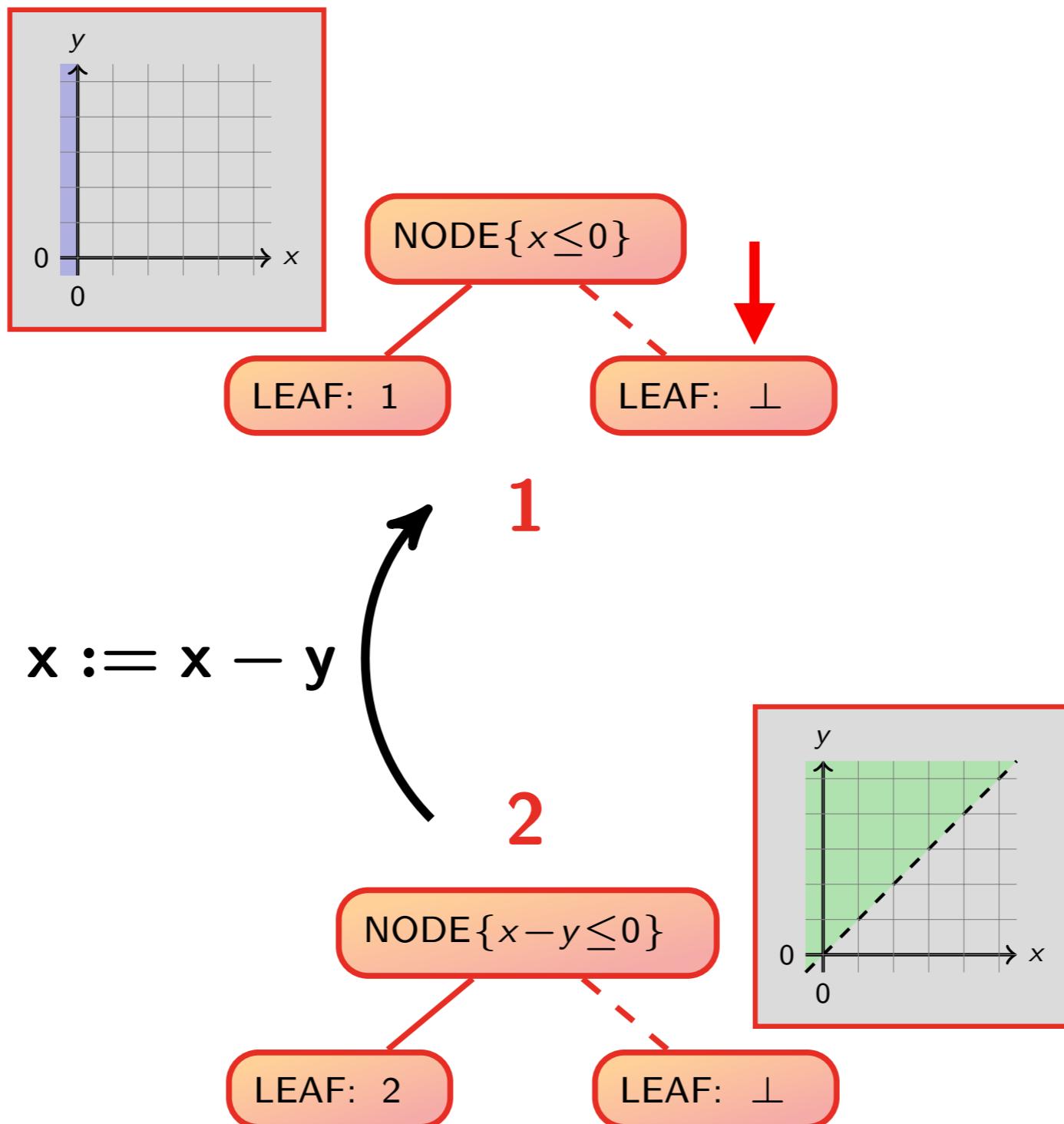
Assignments



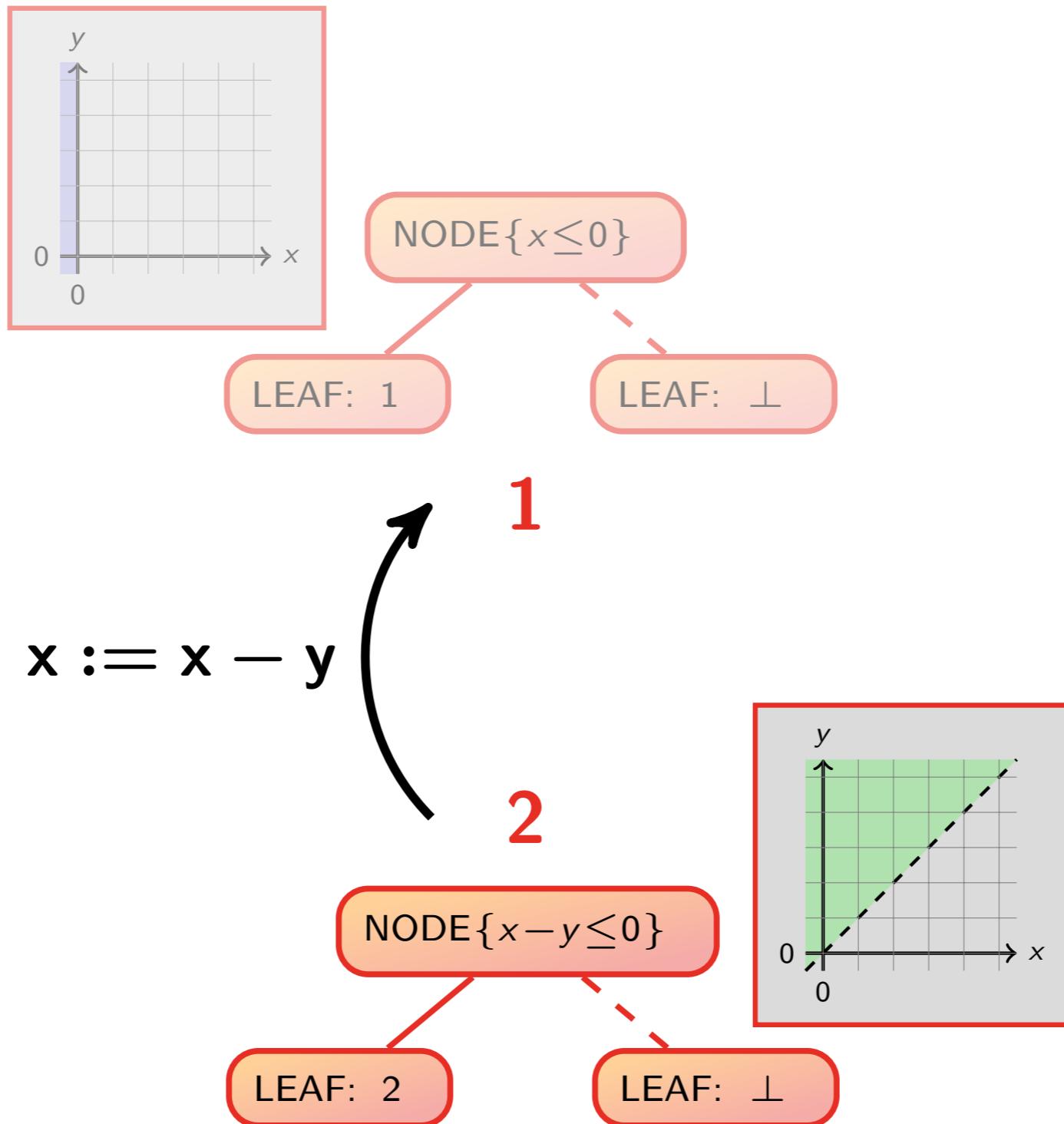
Assignments



Assignments



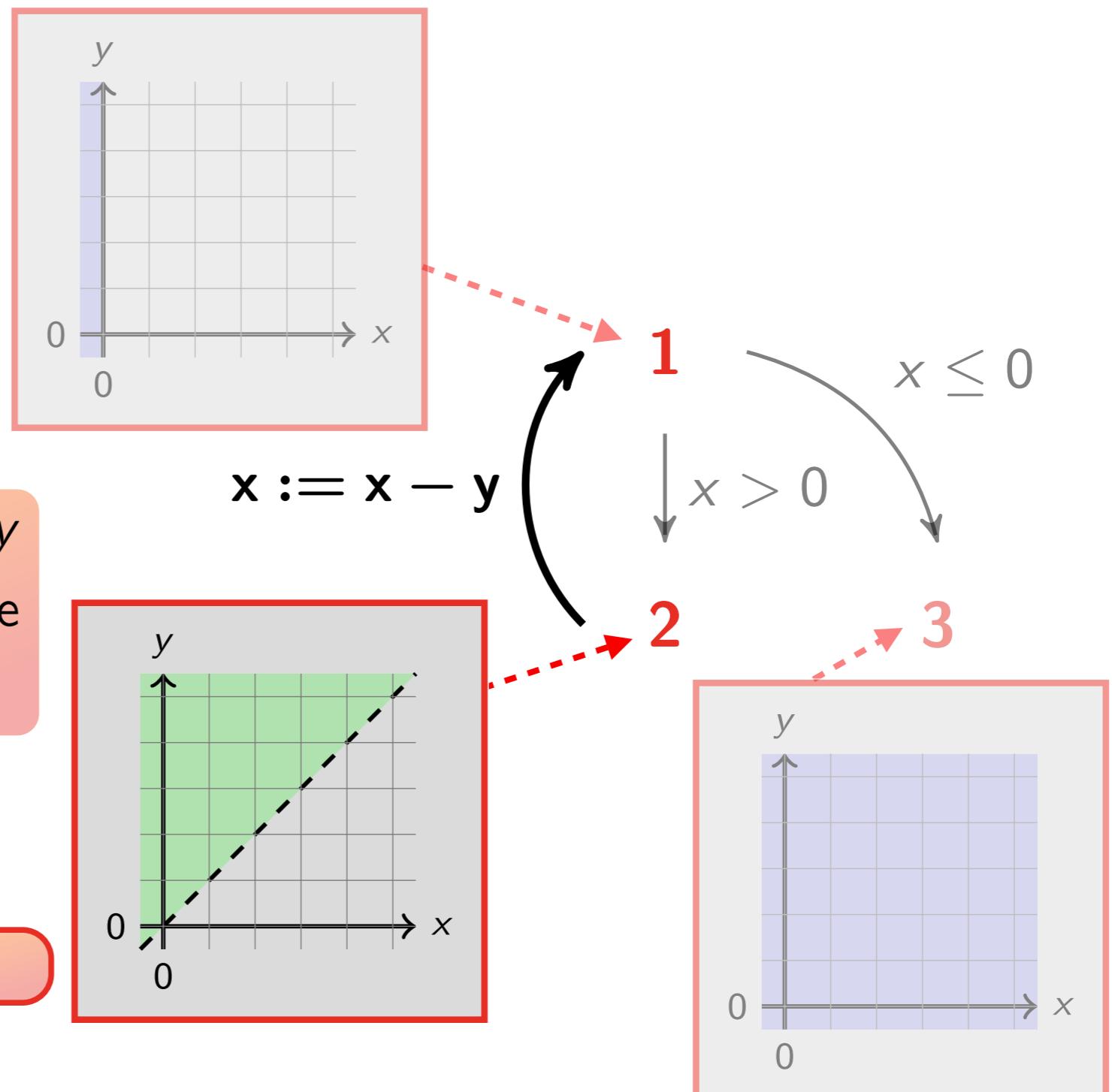
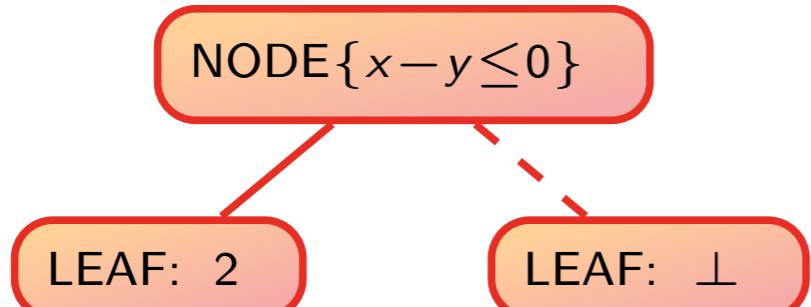
Assignments



Example

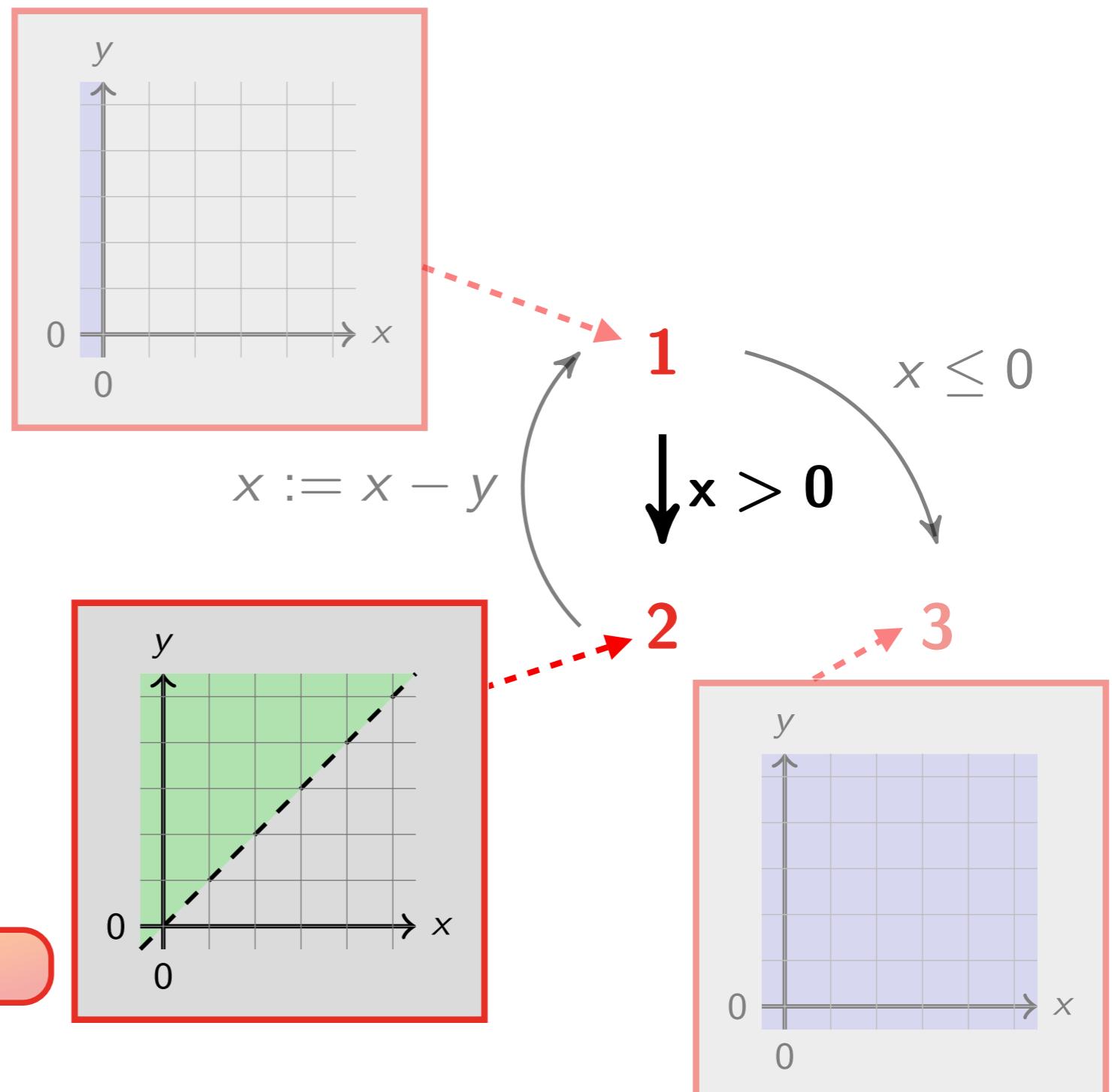
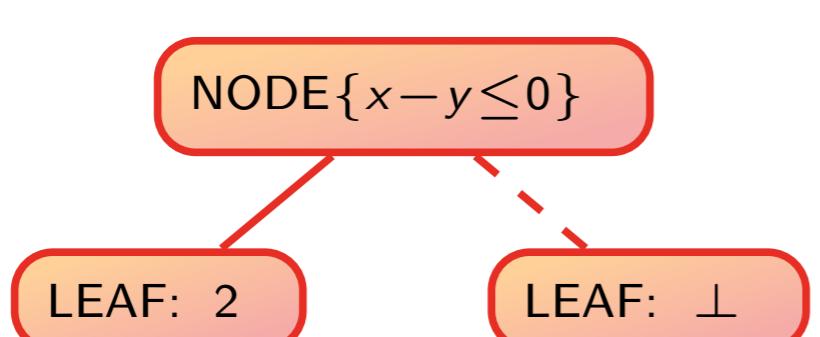
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

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

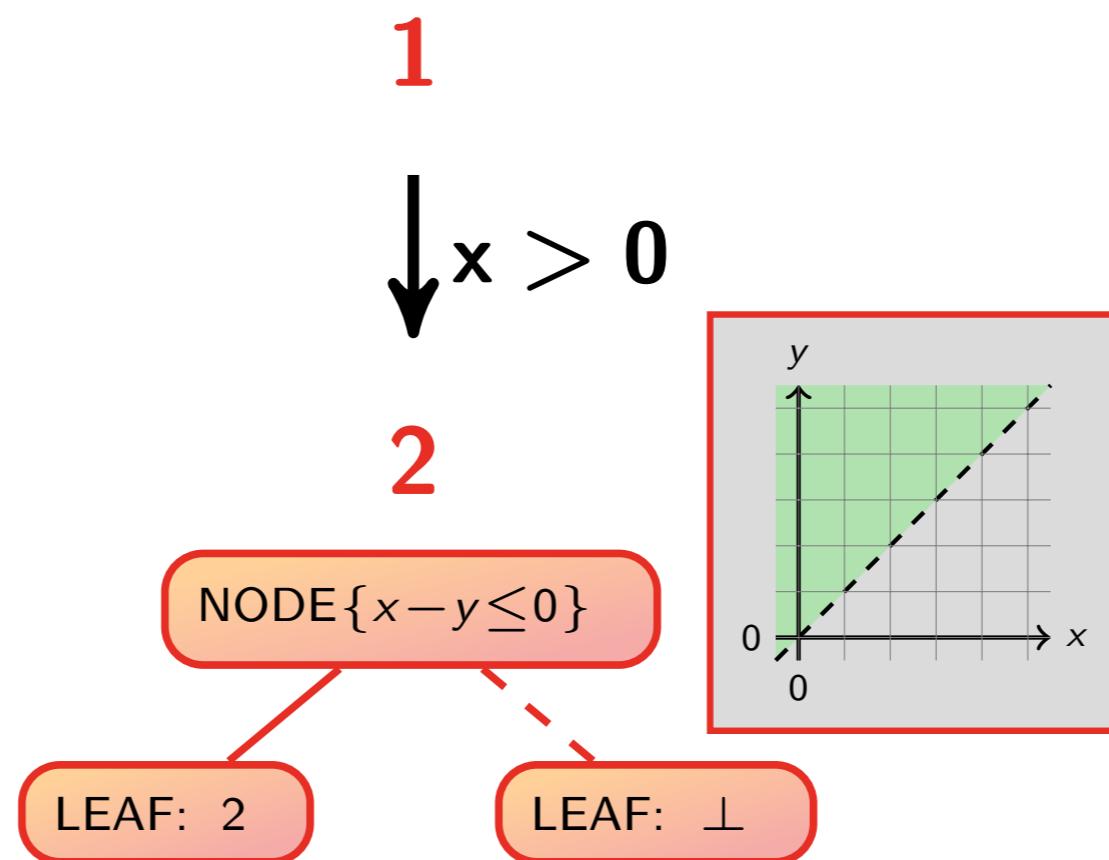


Example

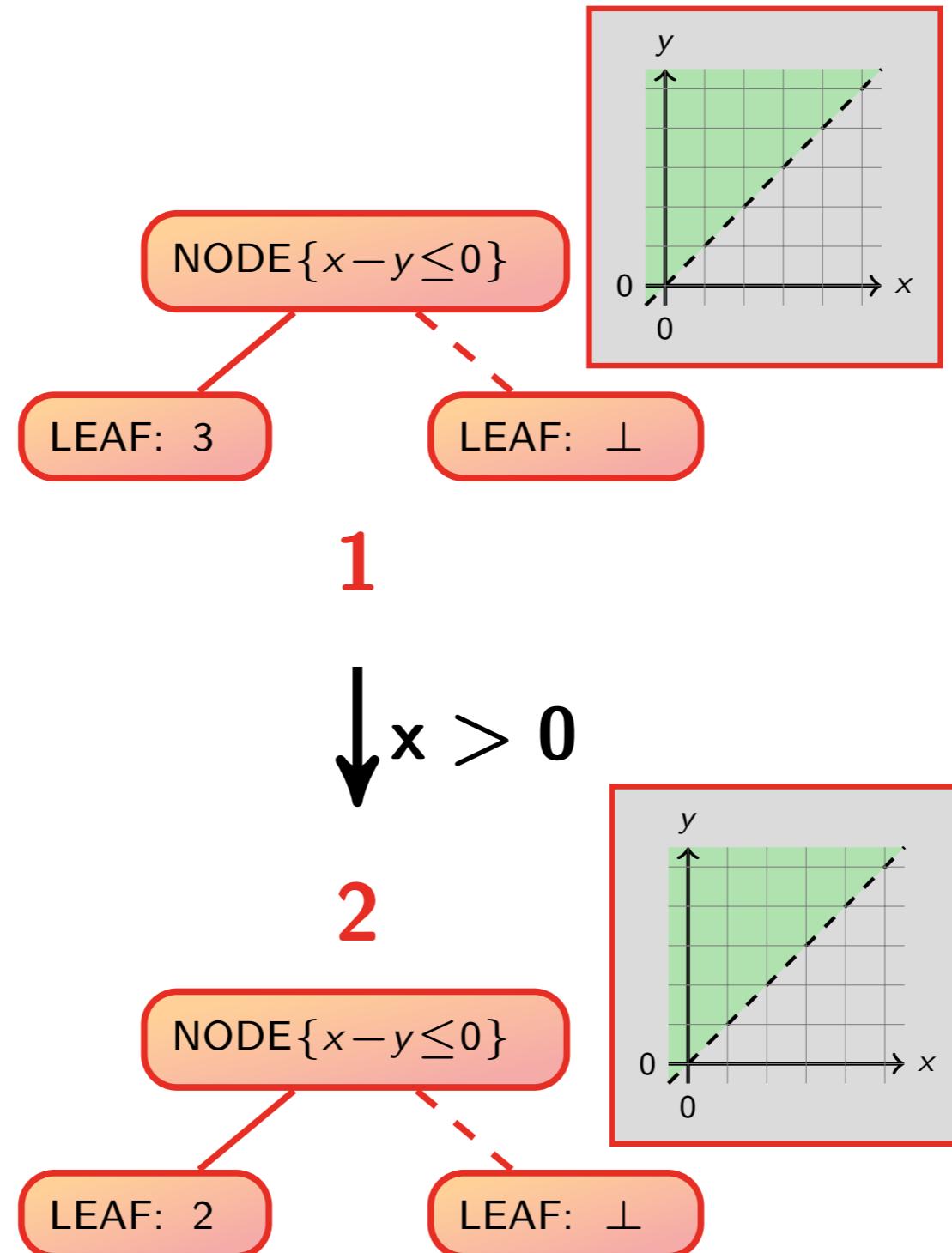
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



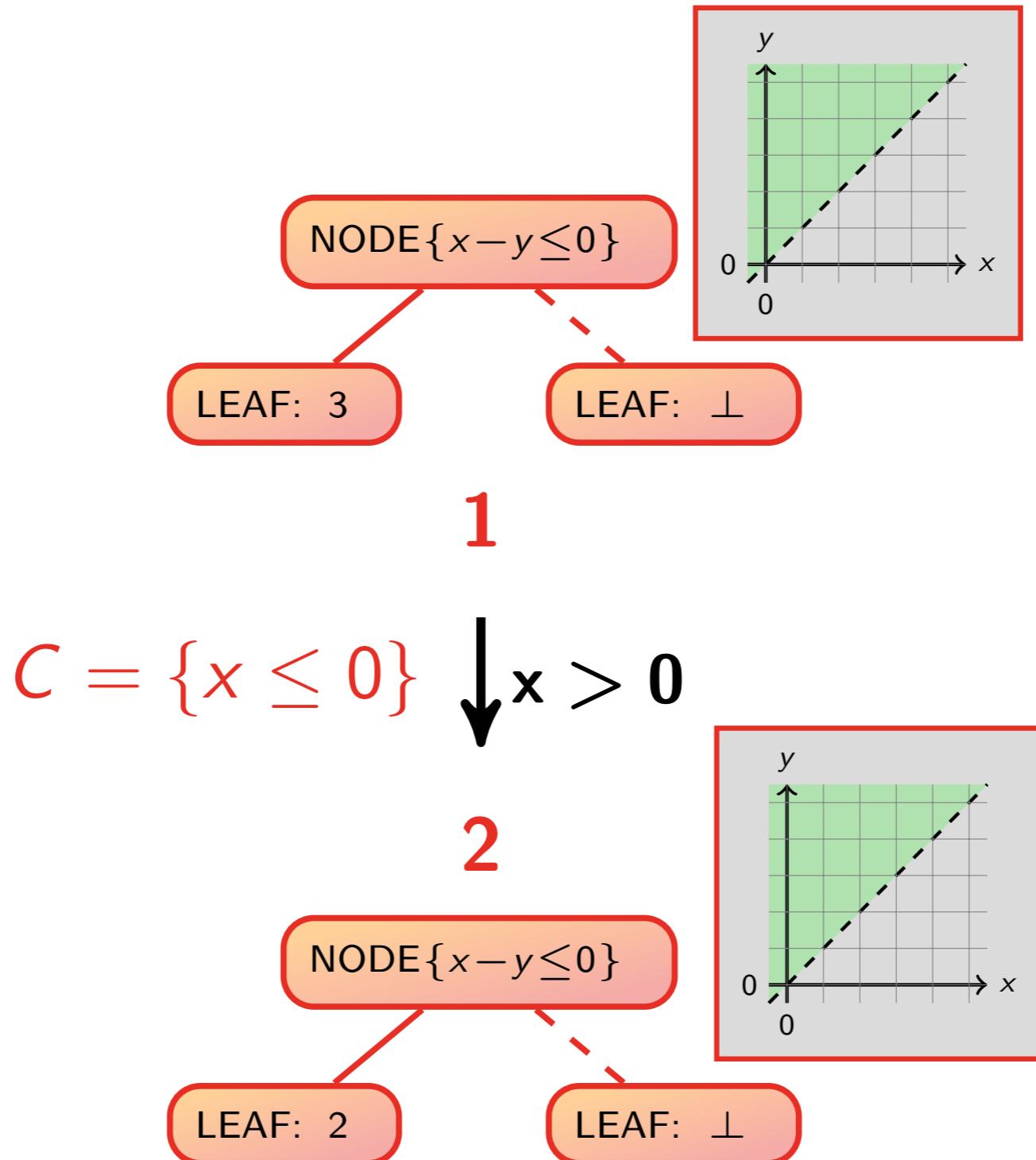
Tests



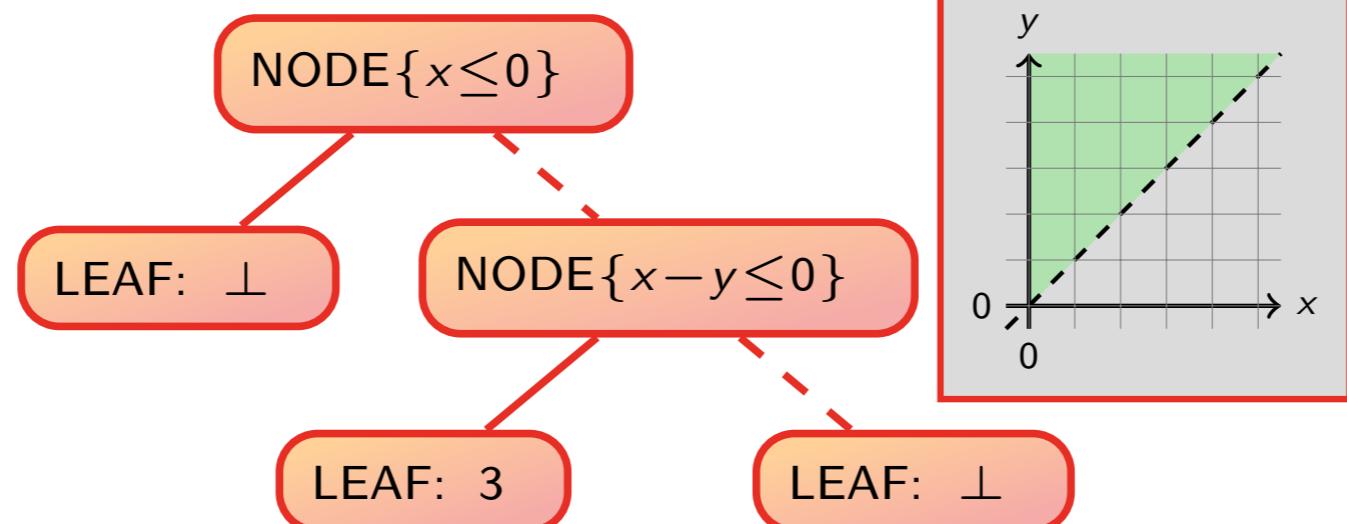
Tests



Tests



Tests

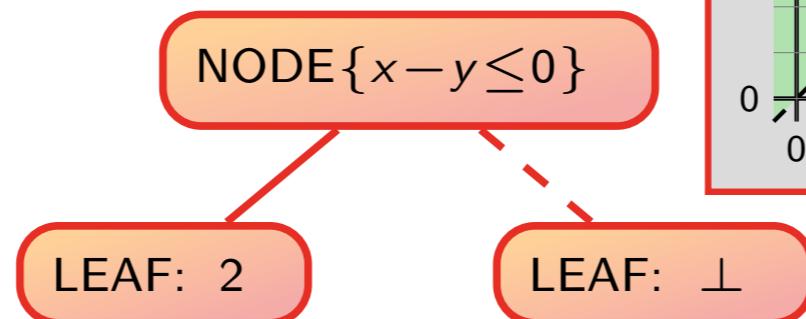


1

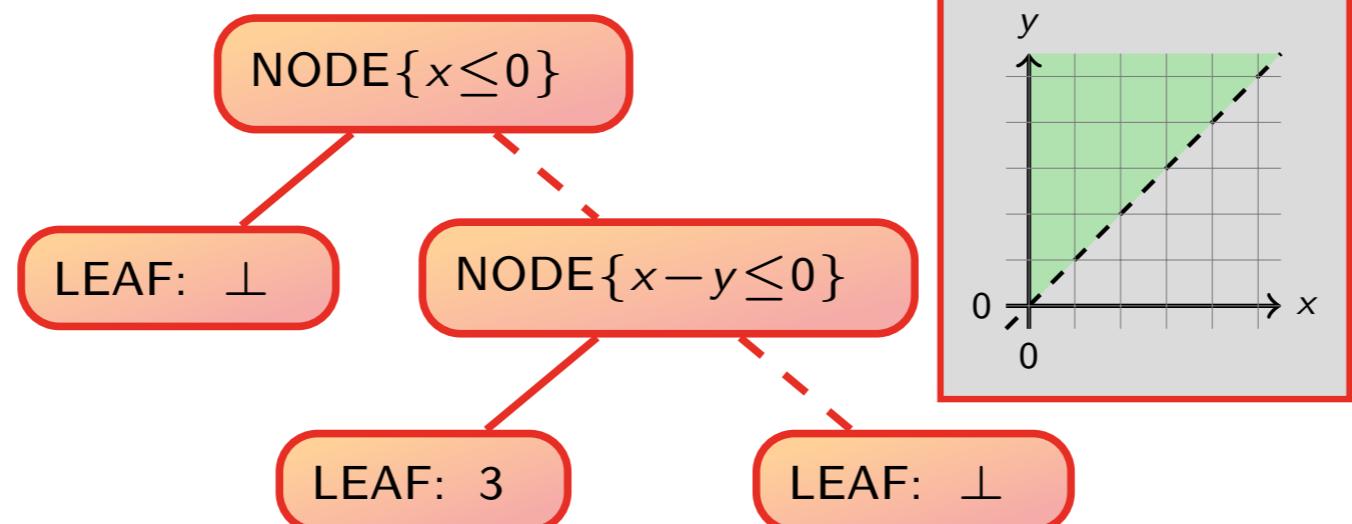
$$C = \{x \leq 0\}$$



2



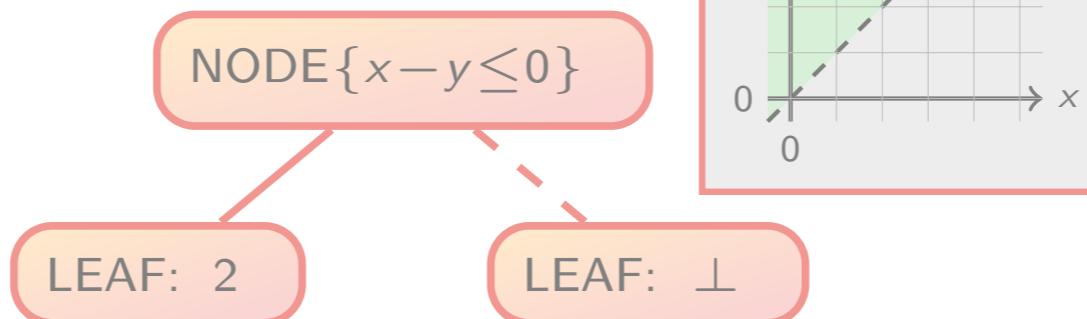
Tests



1

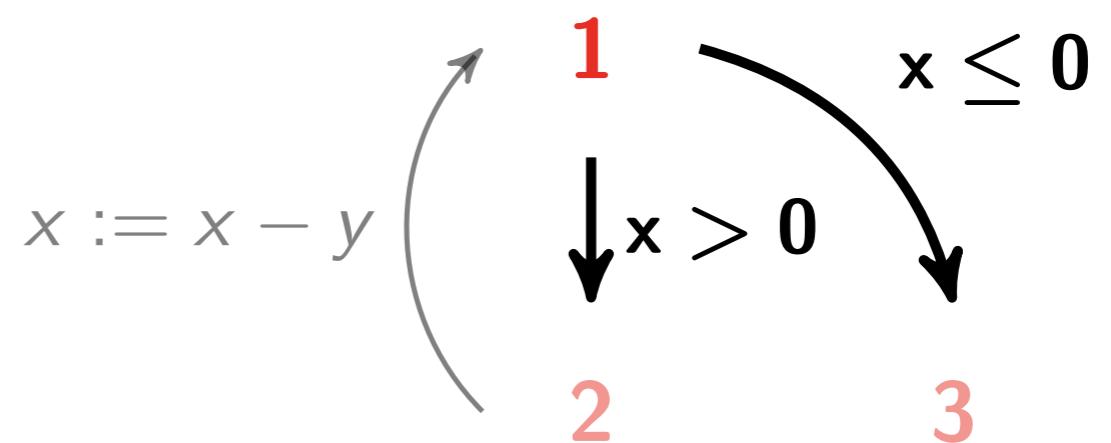
↓
 $x > 0$

2

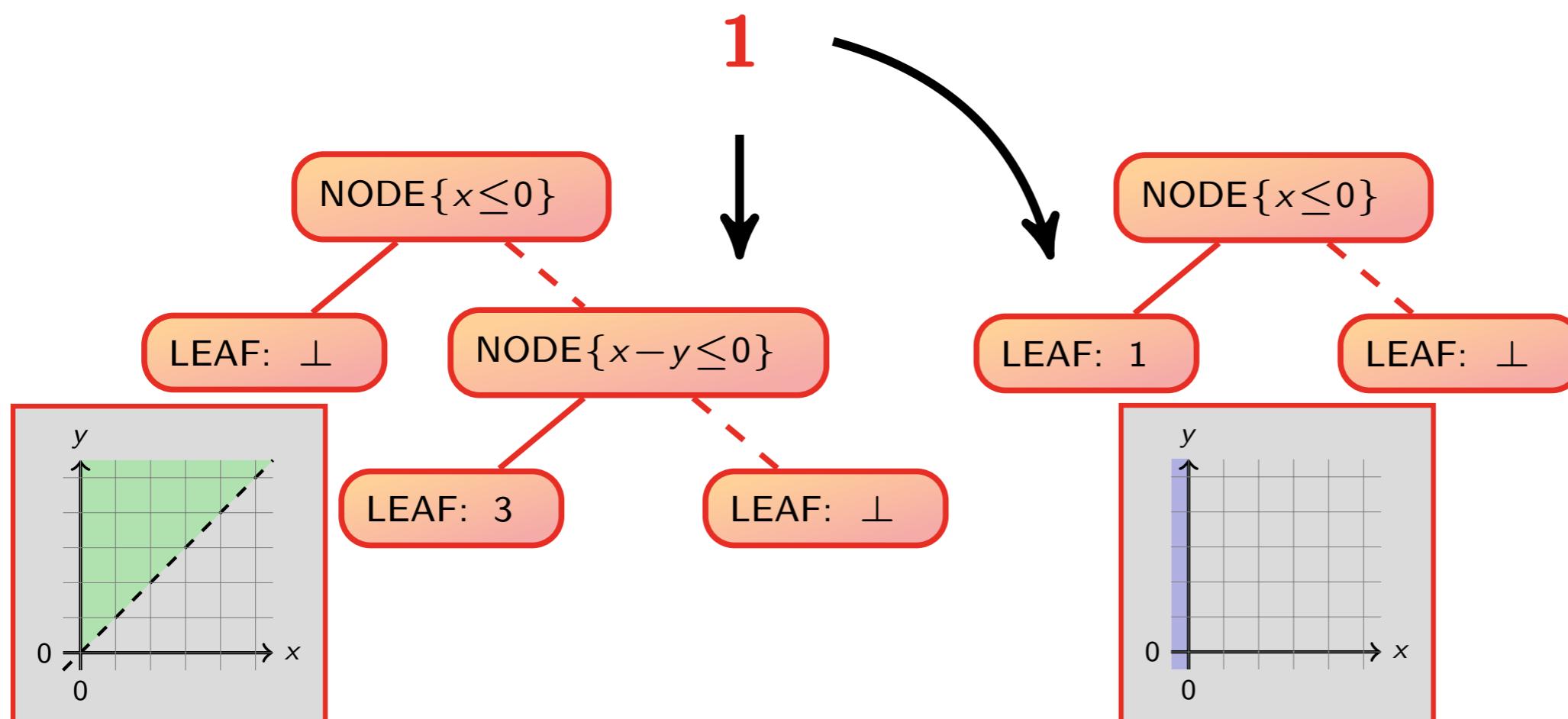


Example

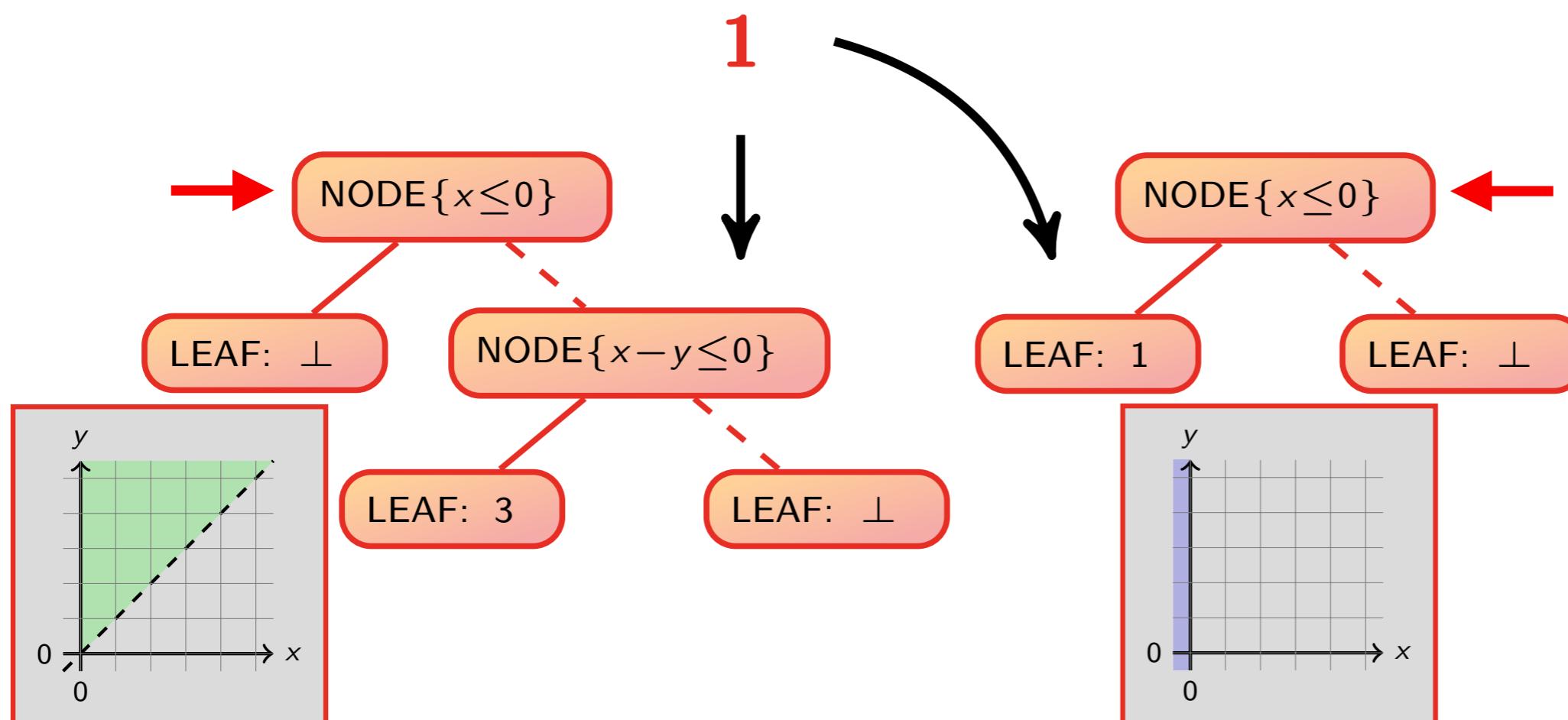
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



Join

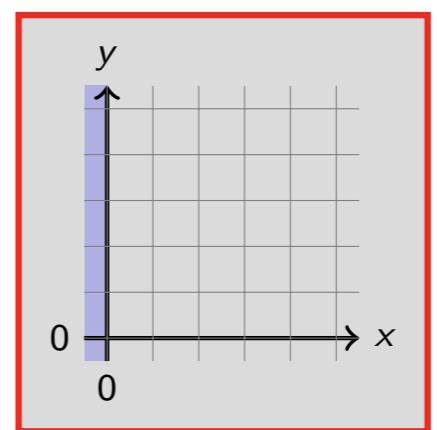
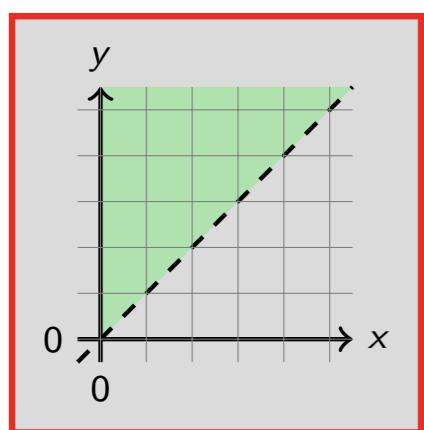
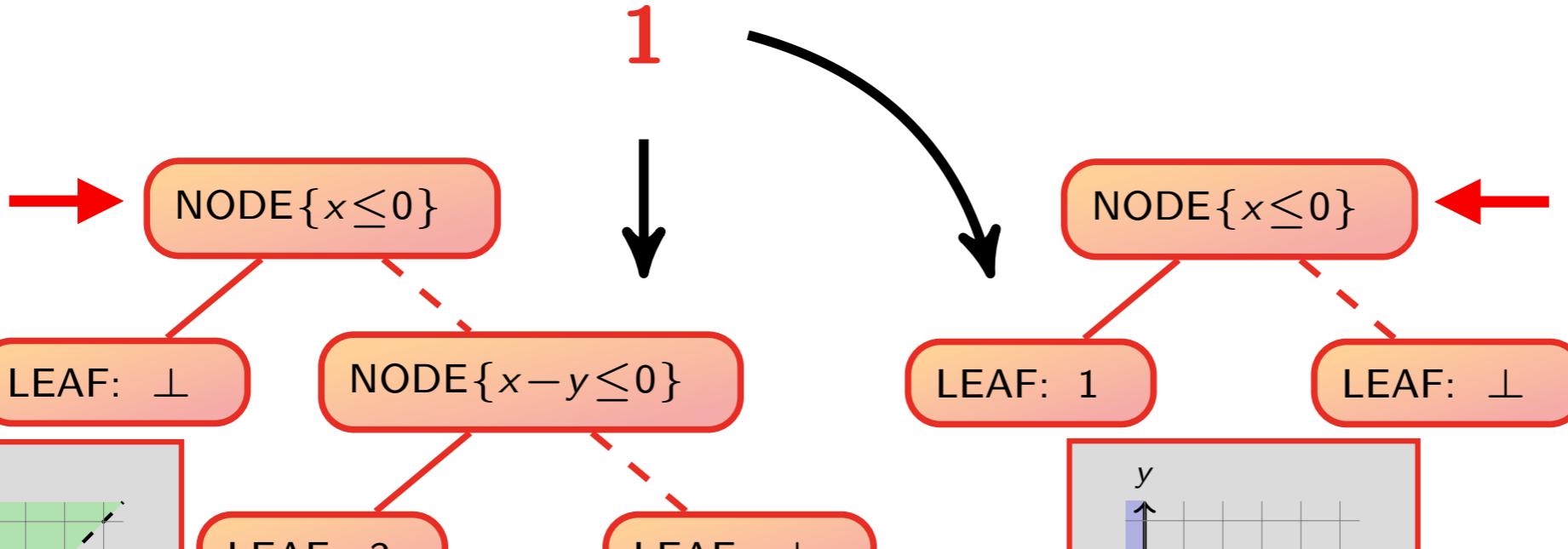
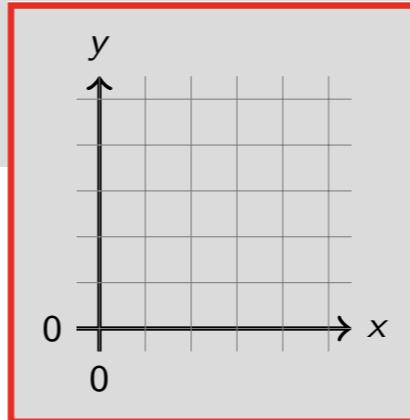


Join



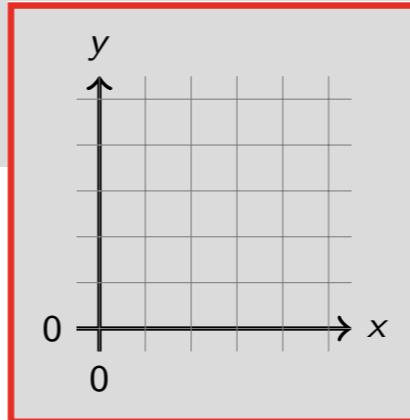
Join

NODE $\{x \leq 0\}$

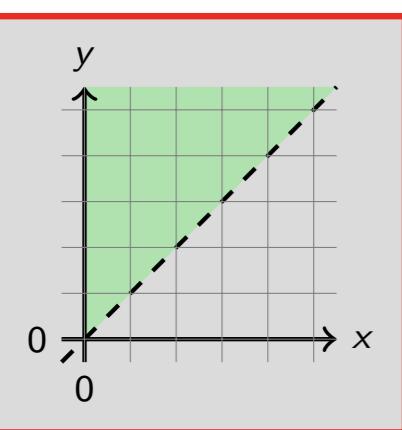


Join

NODE $\{x \leq 0\}$



NODE $\{x \leq 0\}$



LEAF: \perp

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

LEAF: 3

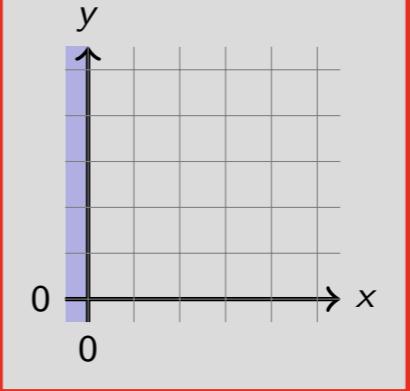
1



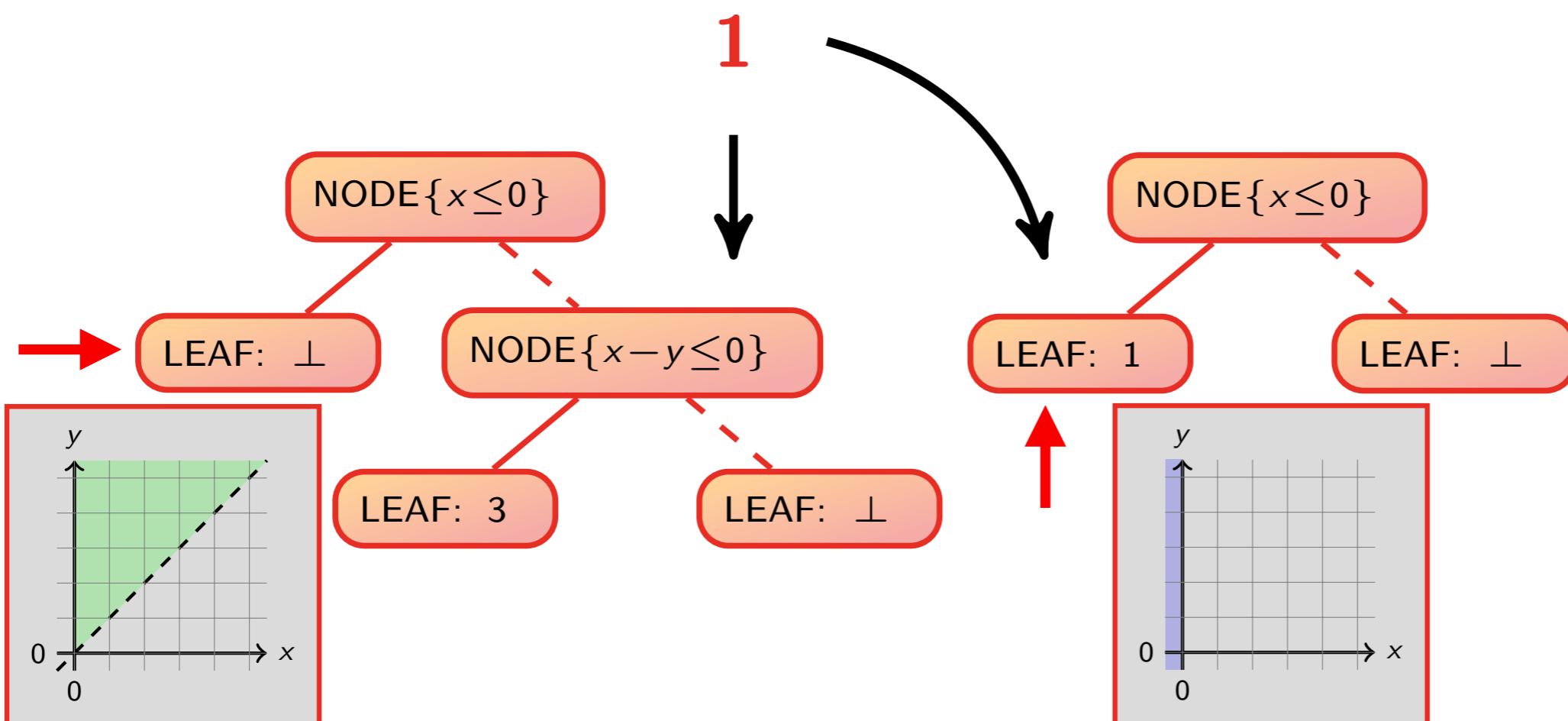
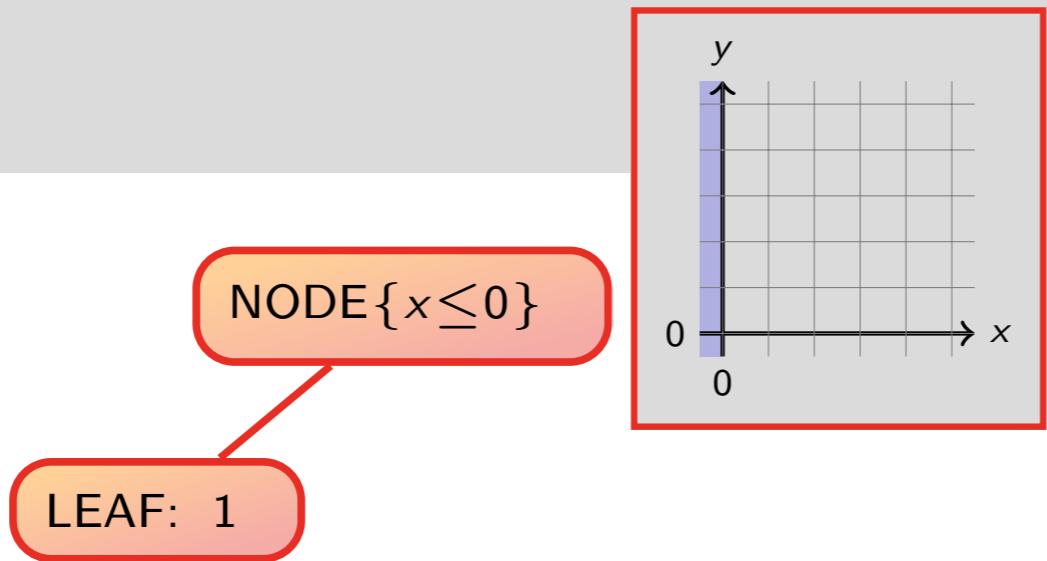
NODE $\{x \leq 0\}$

LEAF: 1

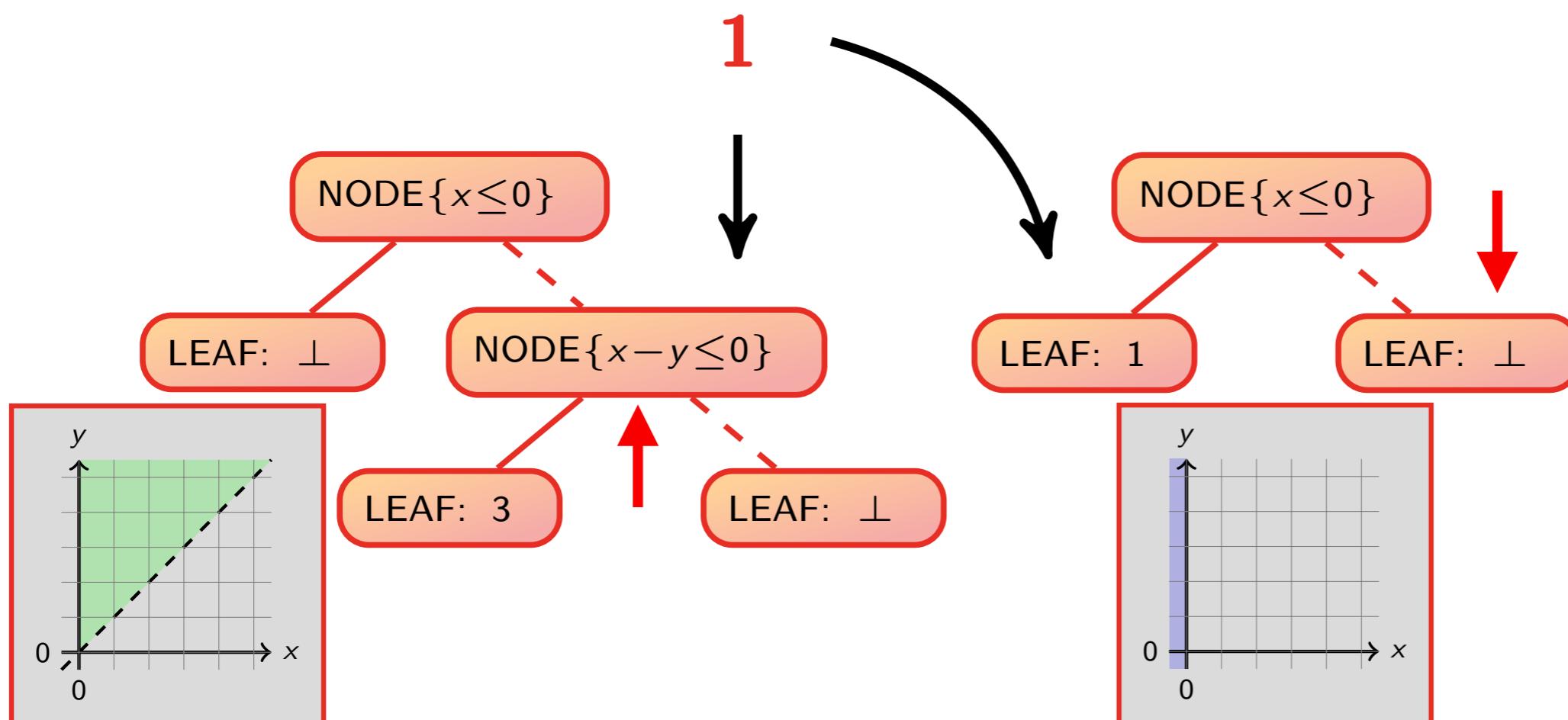
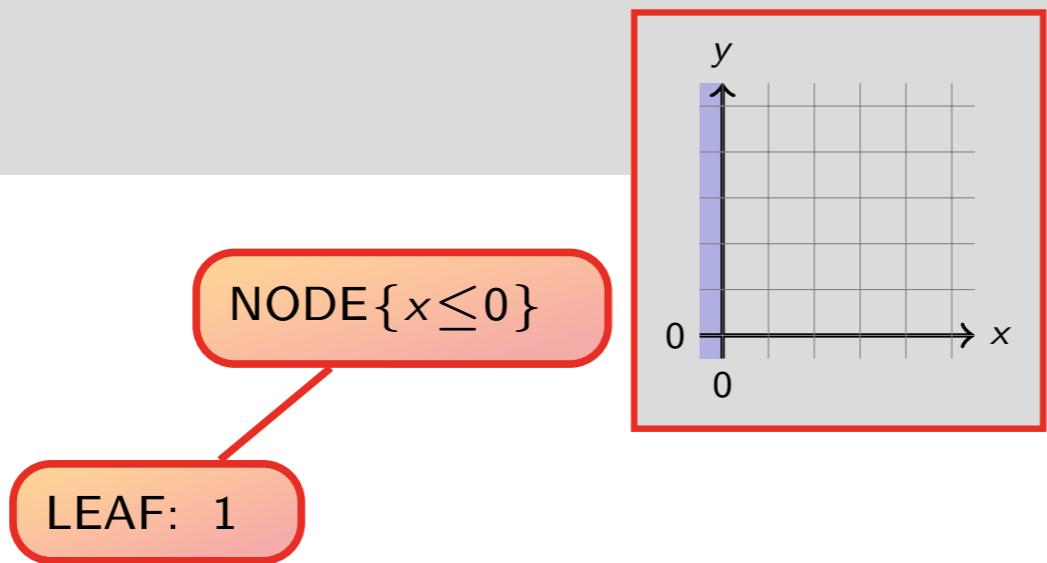
LEAF: \perp



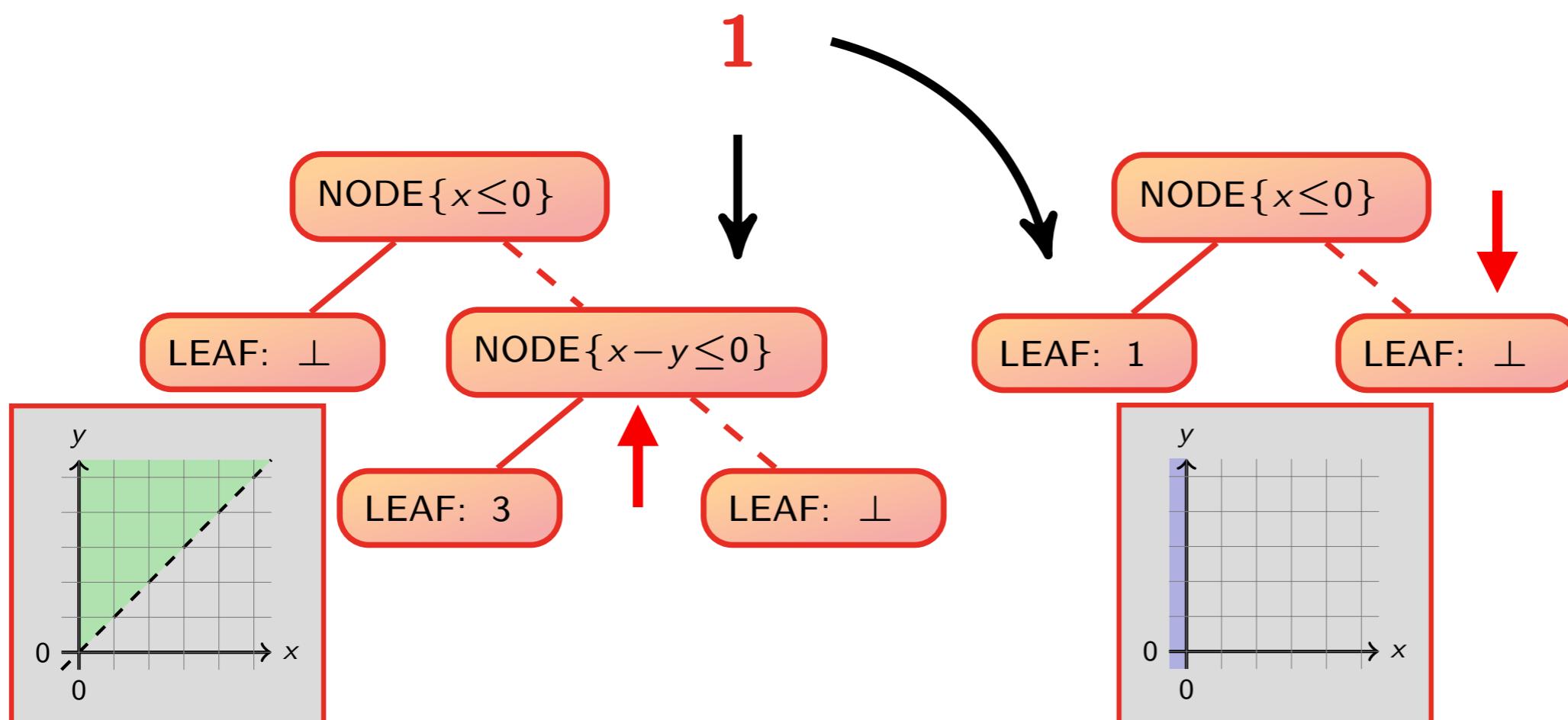
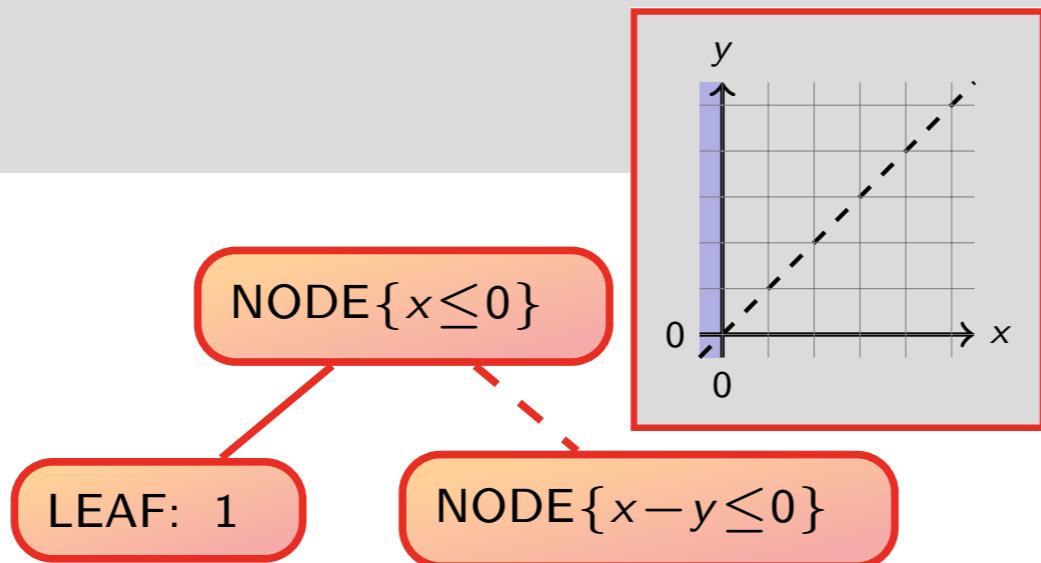
Join



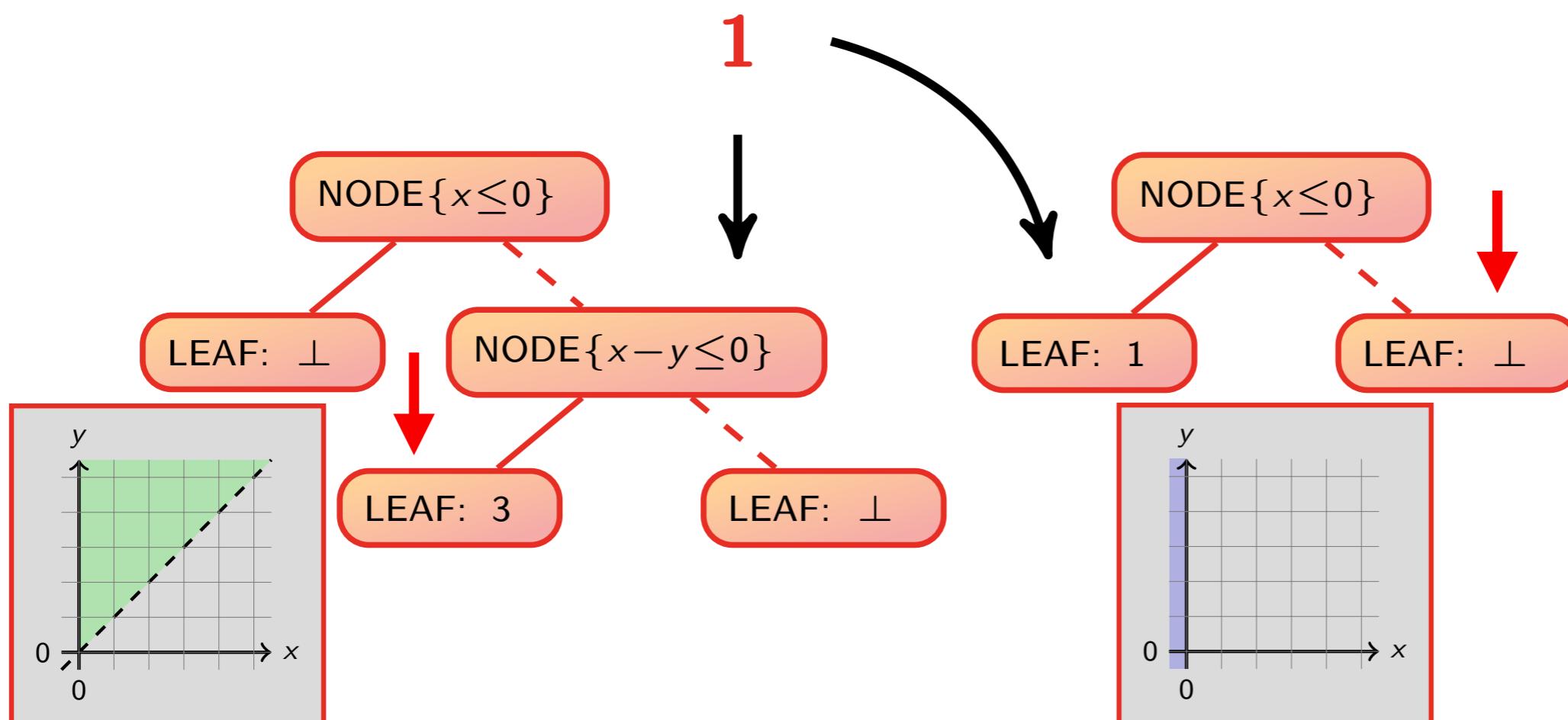
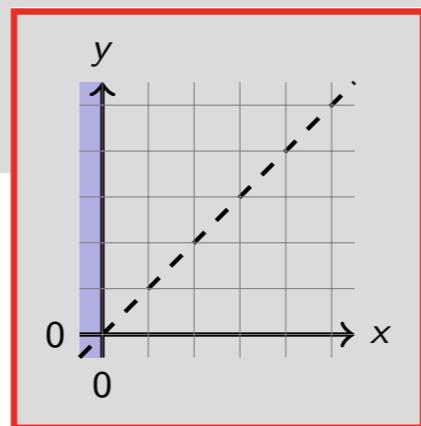
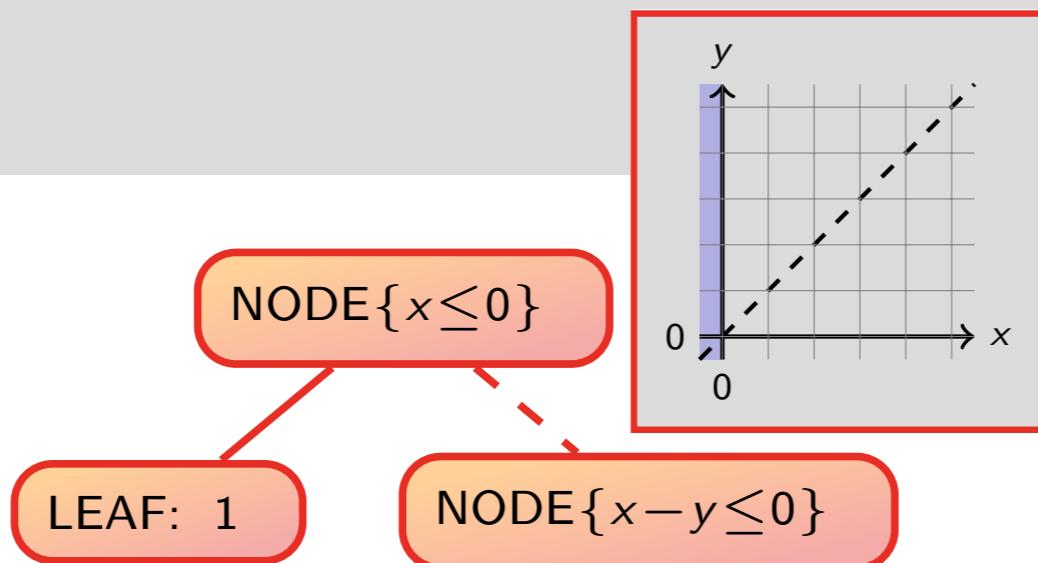
Join



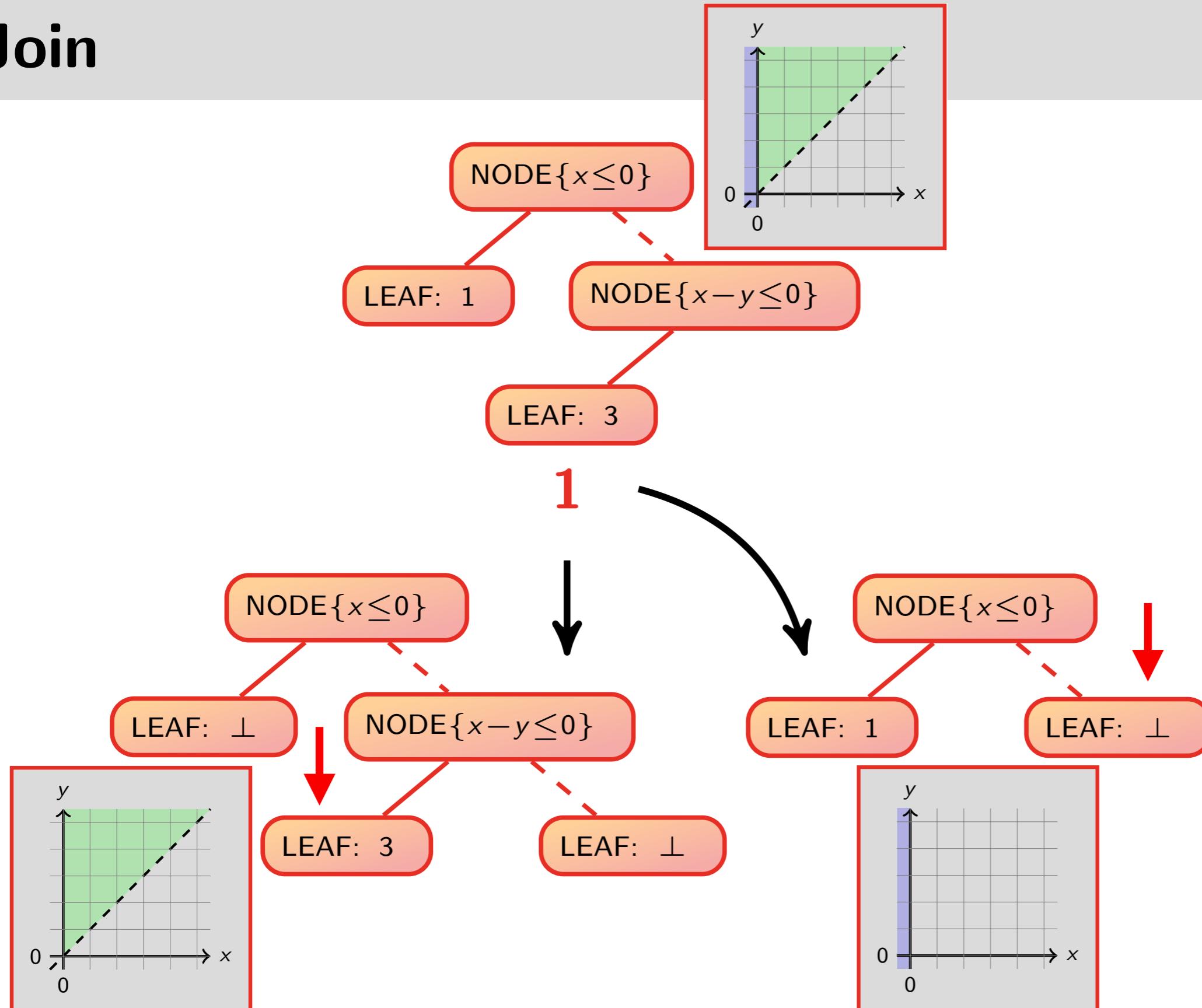
Join



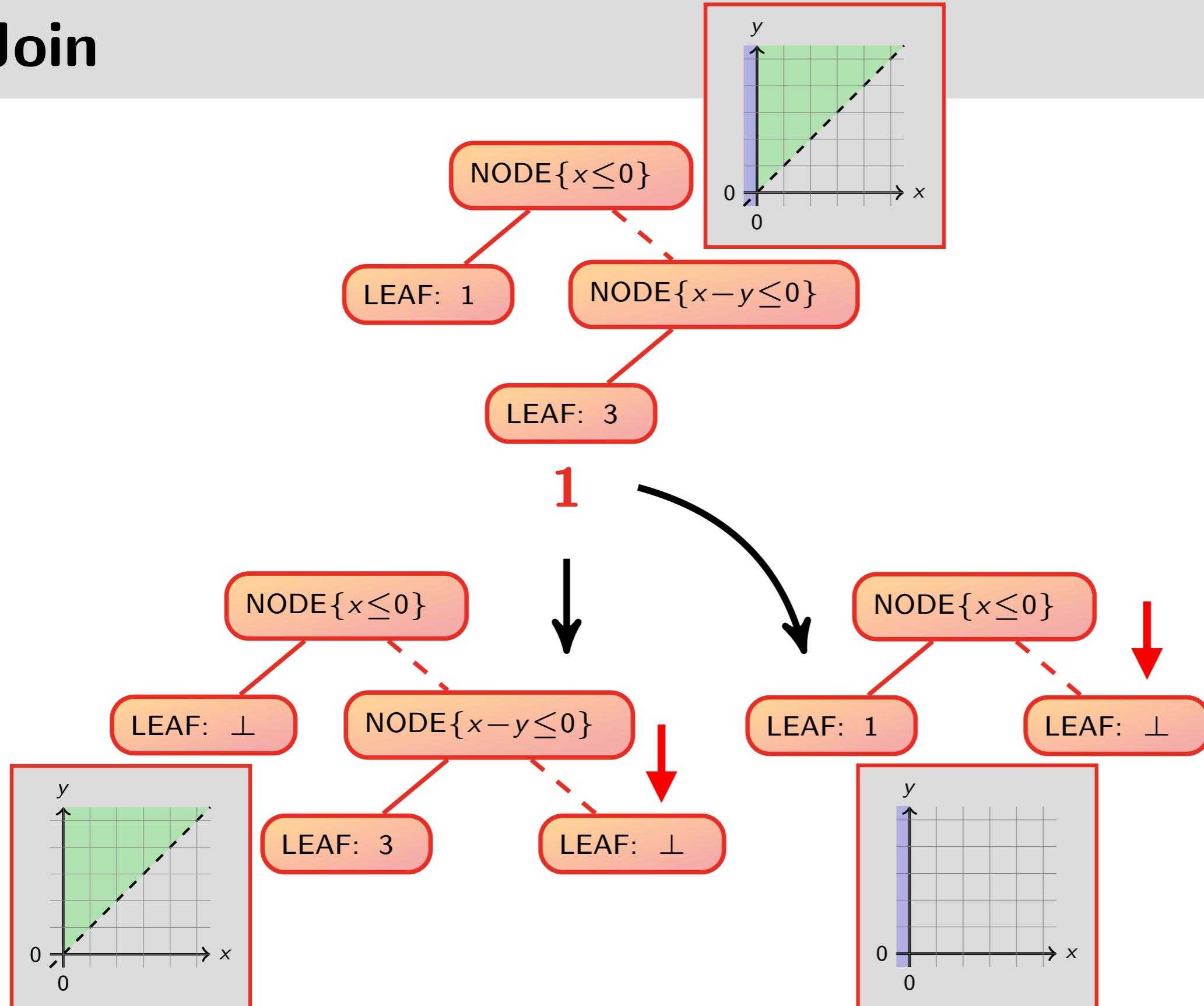
Join



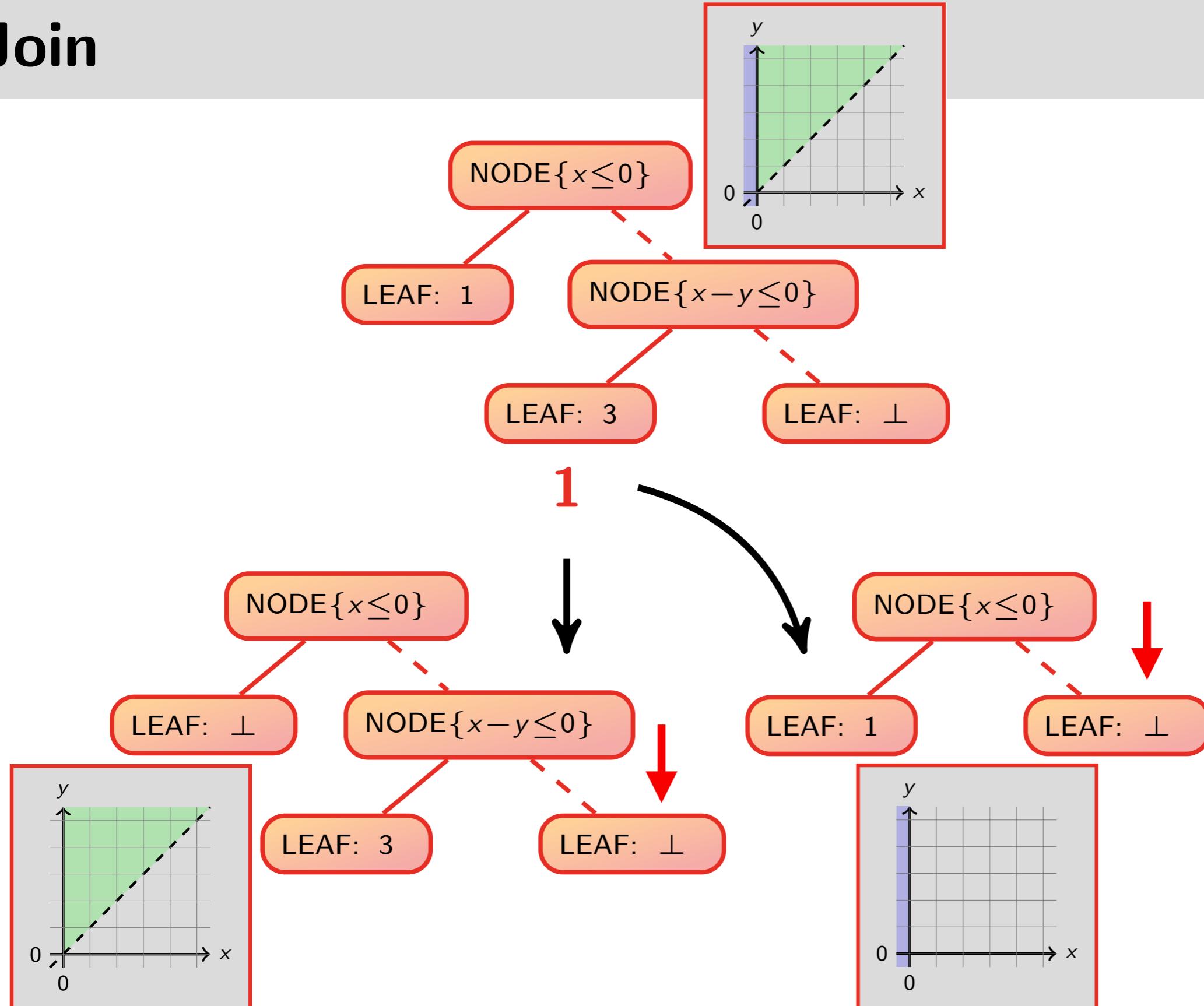
Join



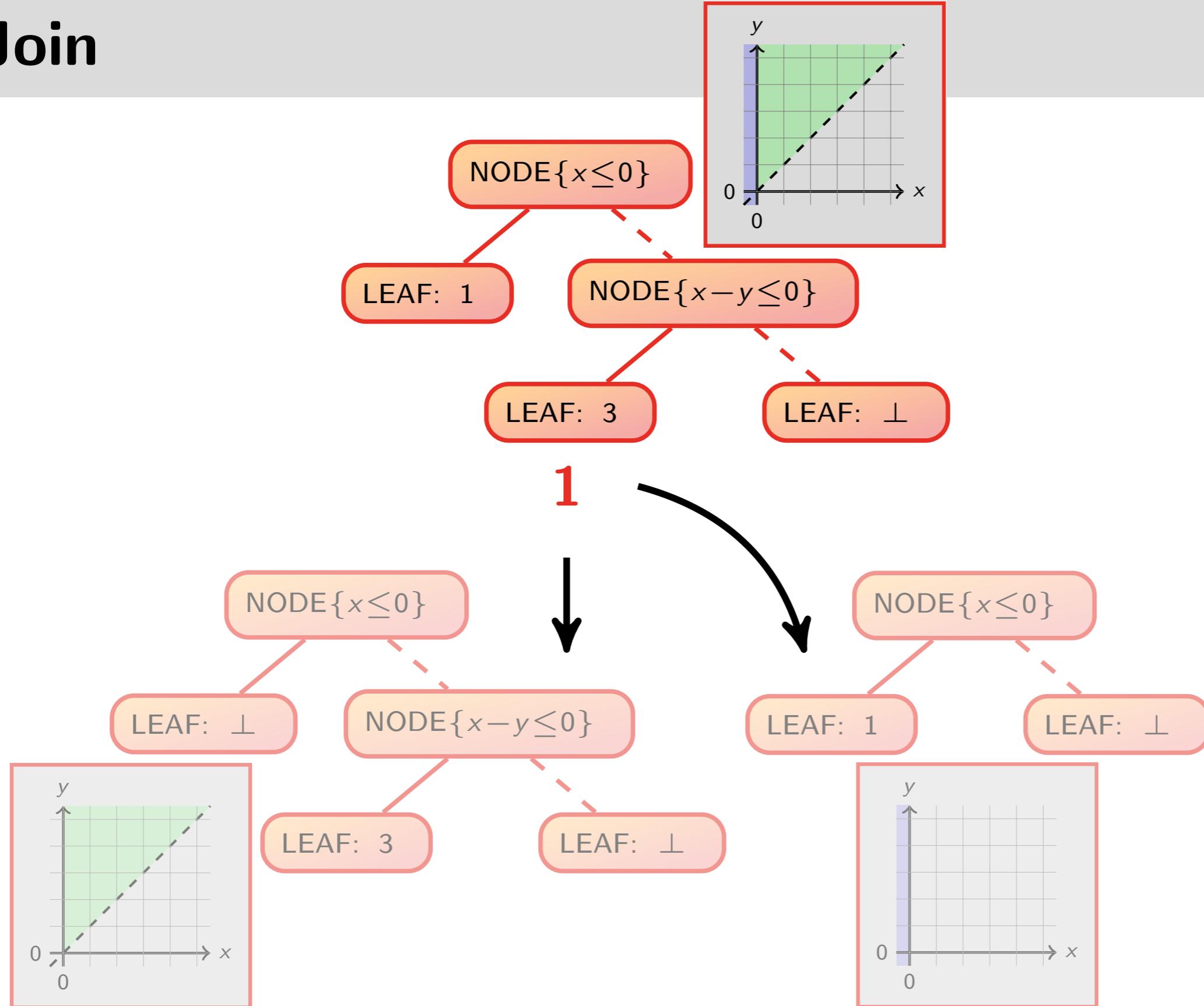
Join



Join



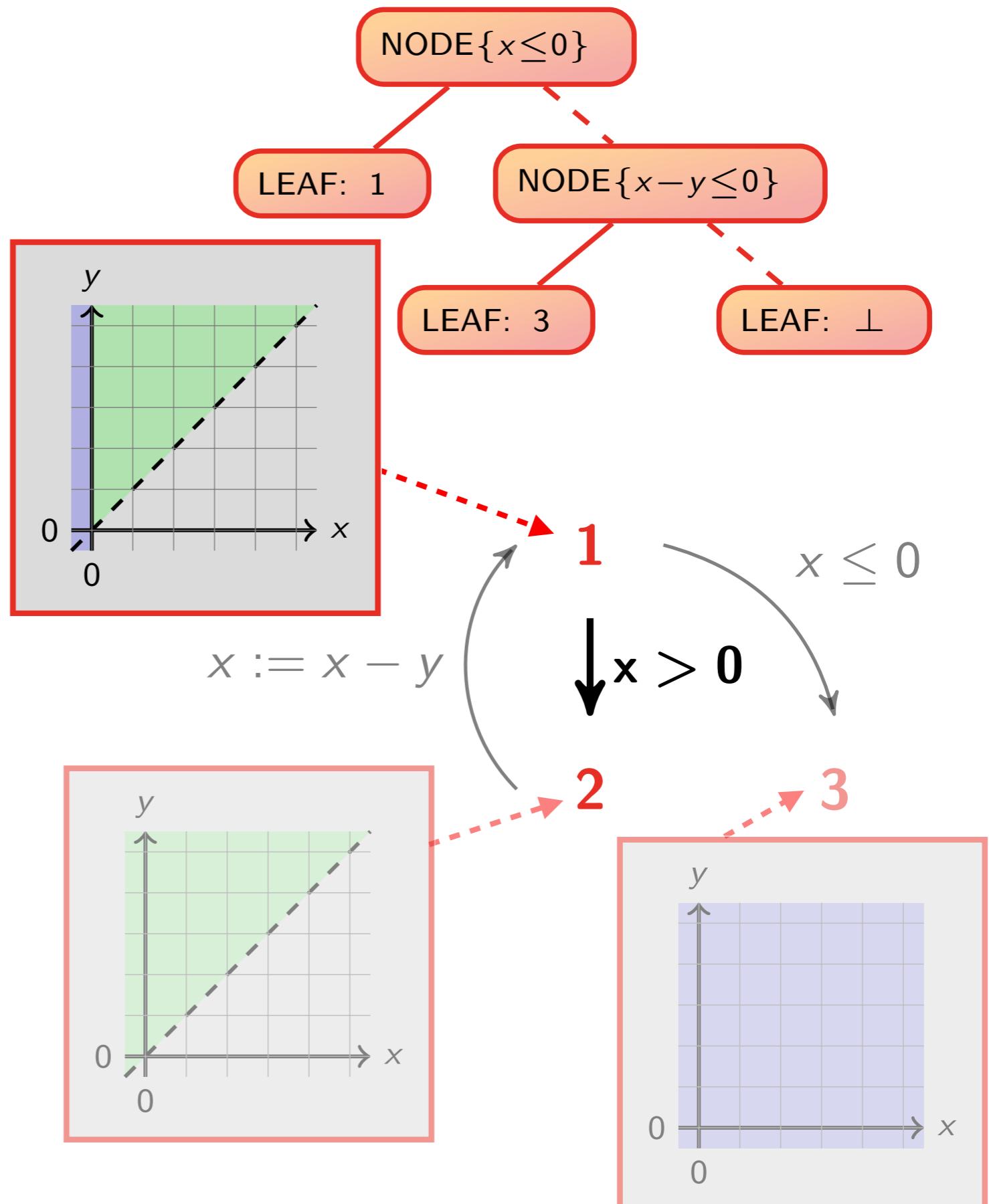
Join



Example

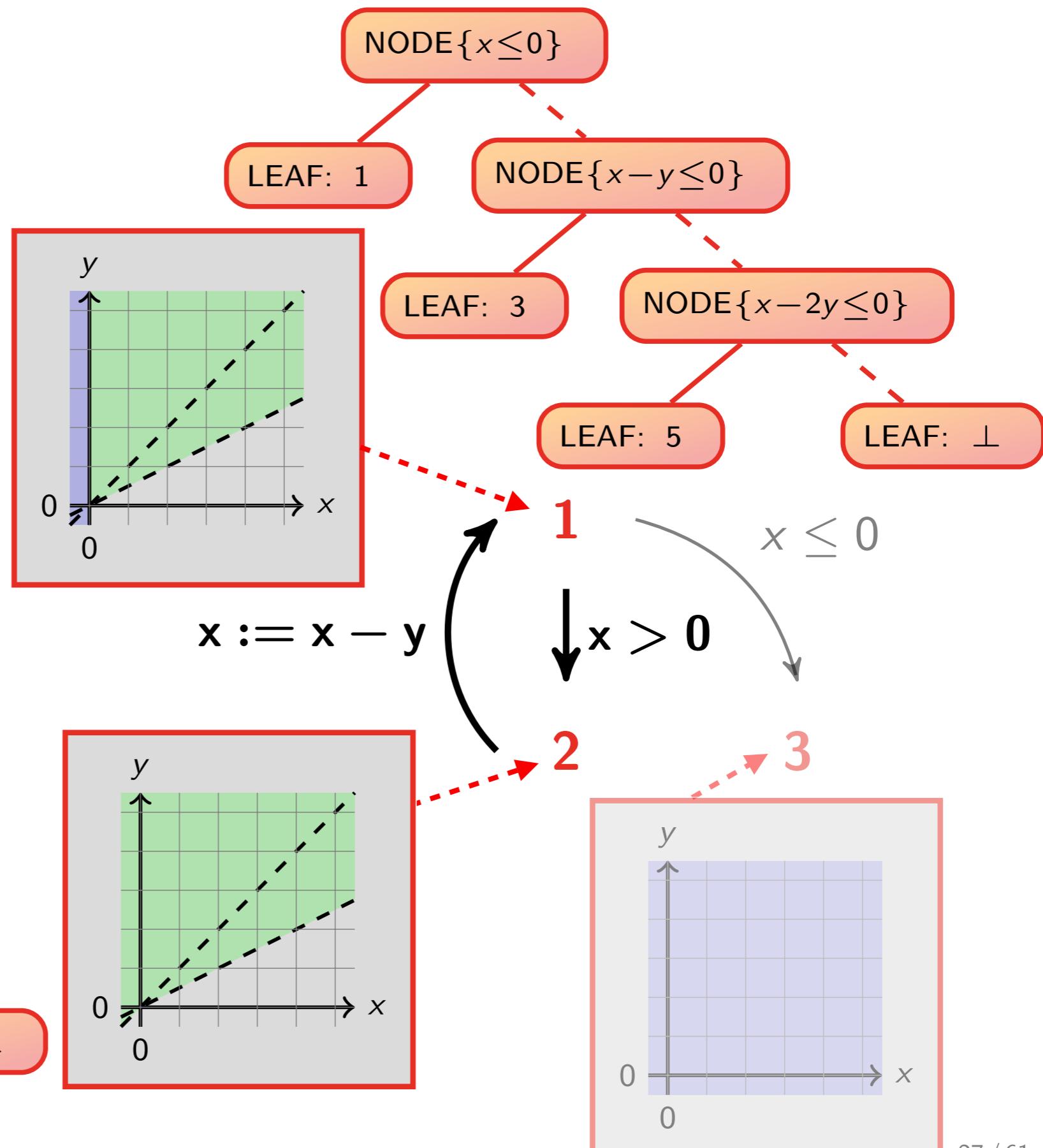
```
int : x, y
while 1(x > 0) do
    2x := x - y
od3
```

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

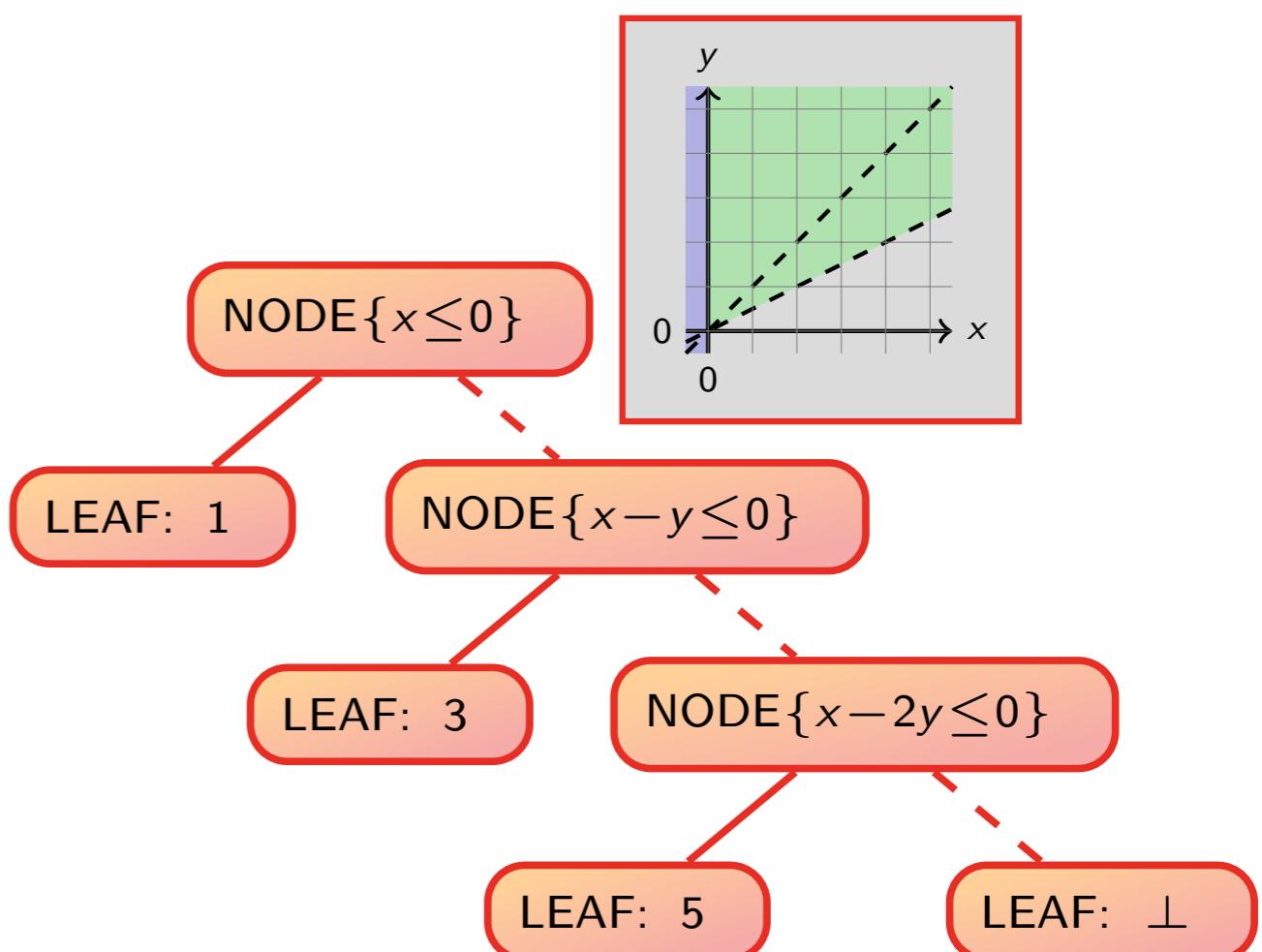
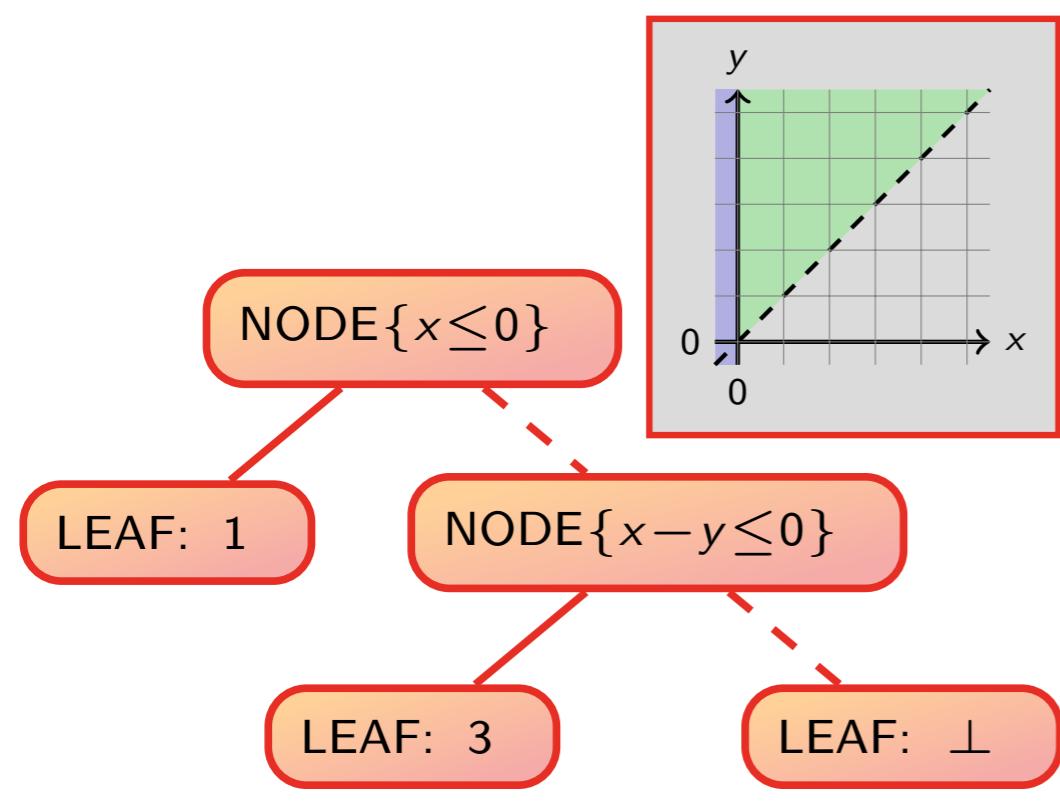


Example

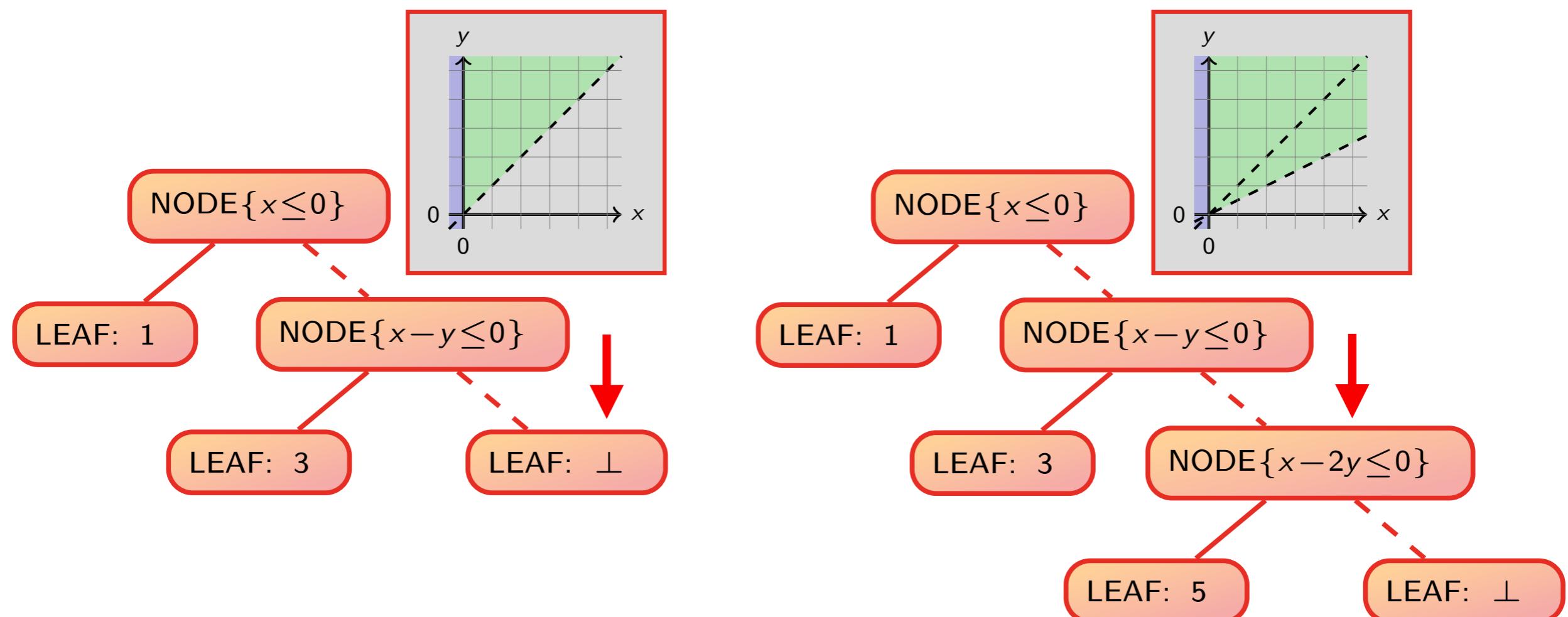
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



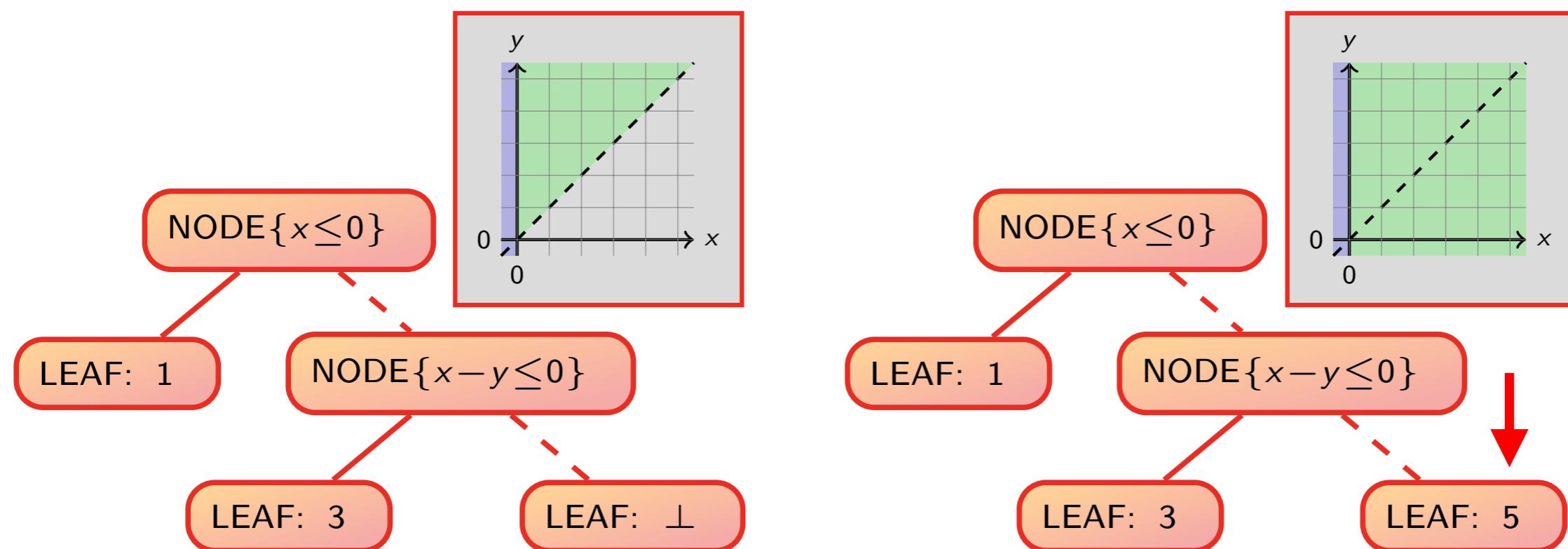
Widening



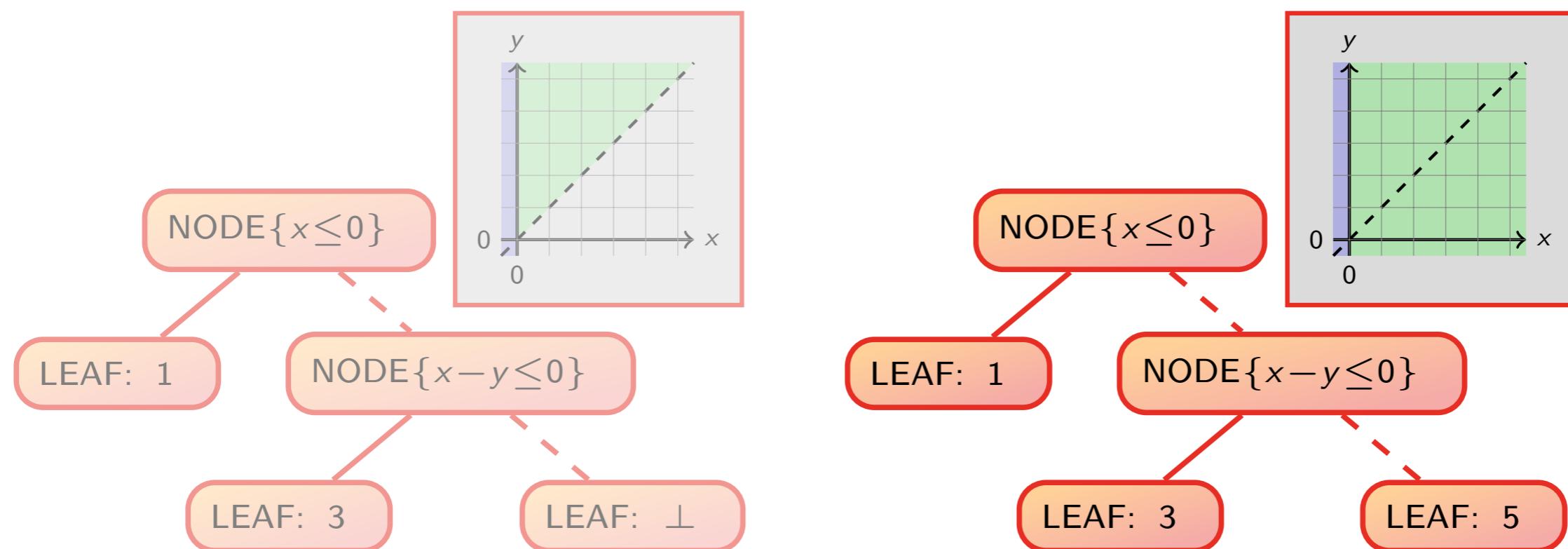
Widening



Widening

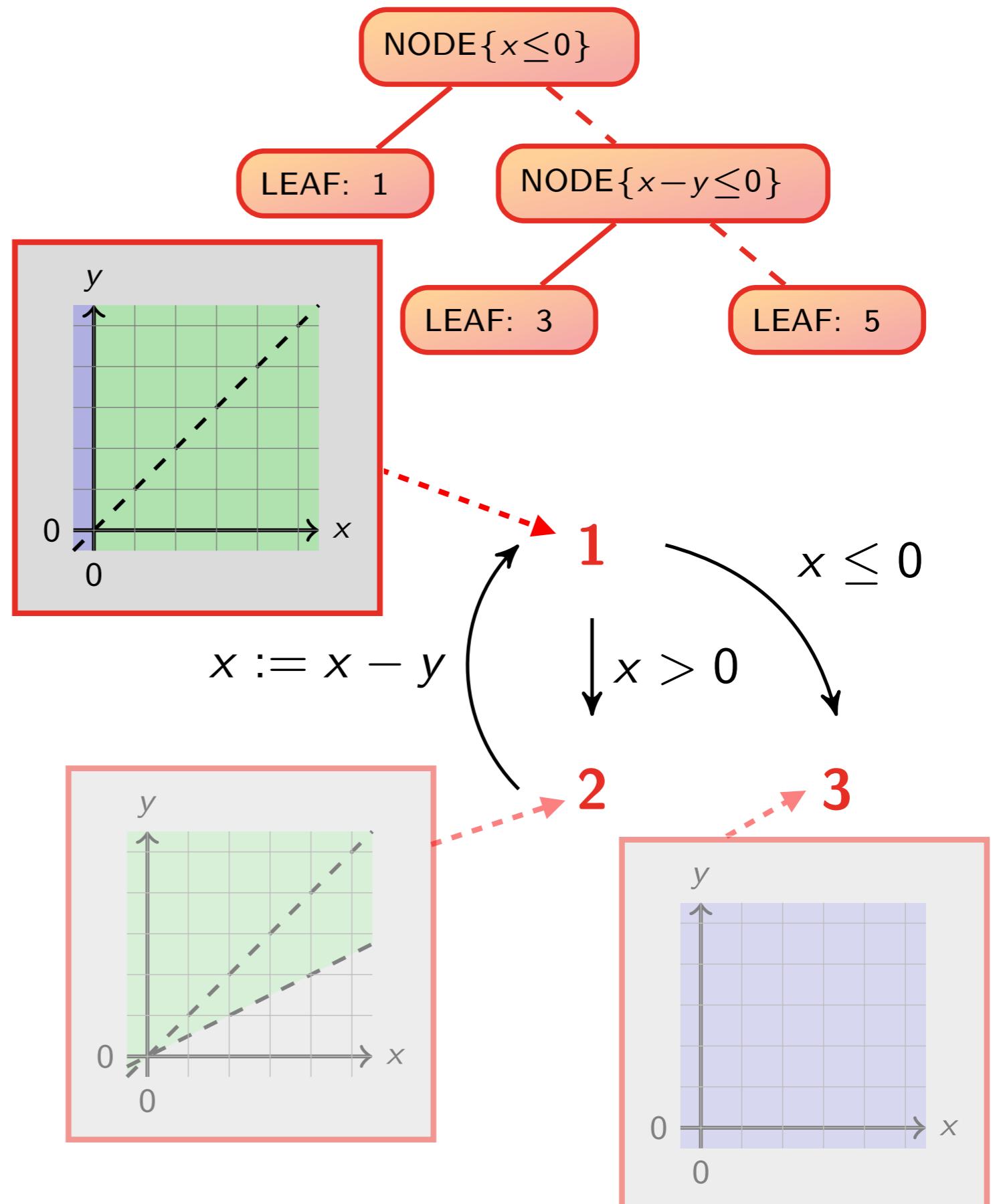


Widening



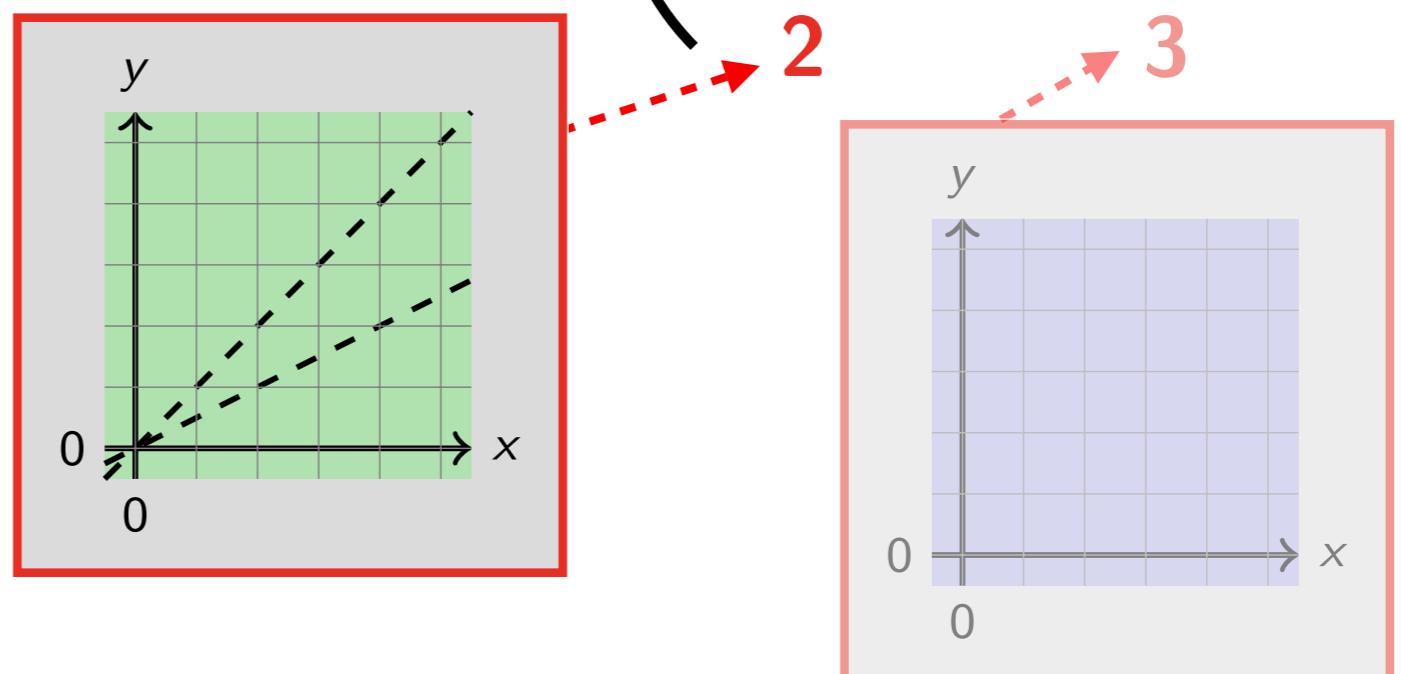
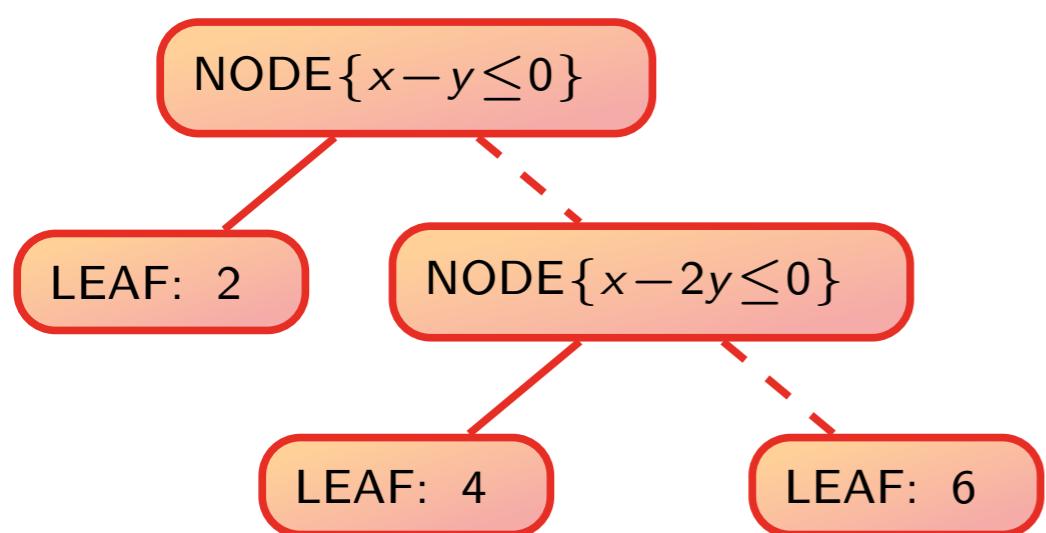
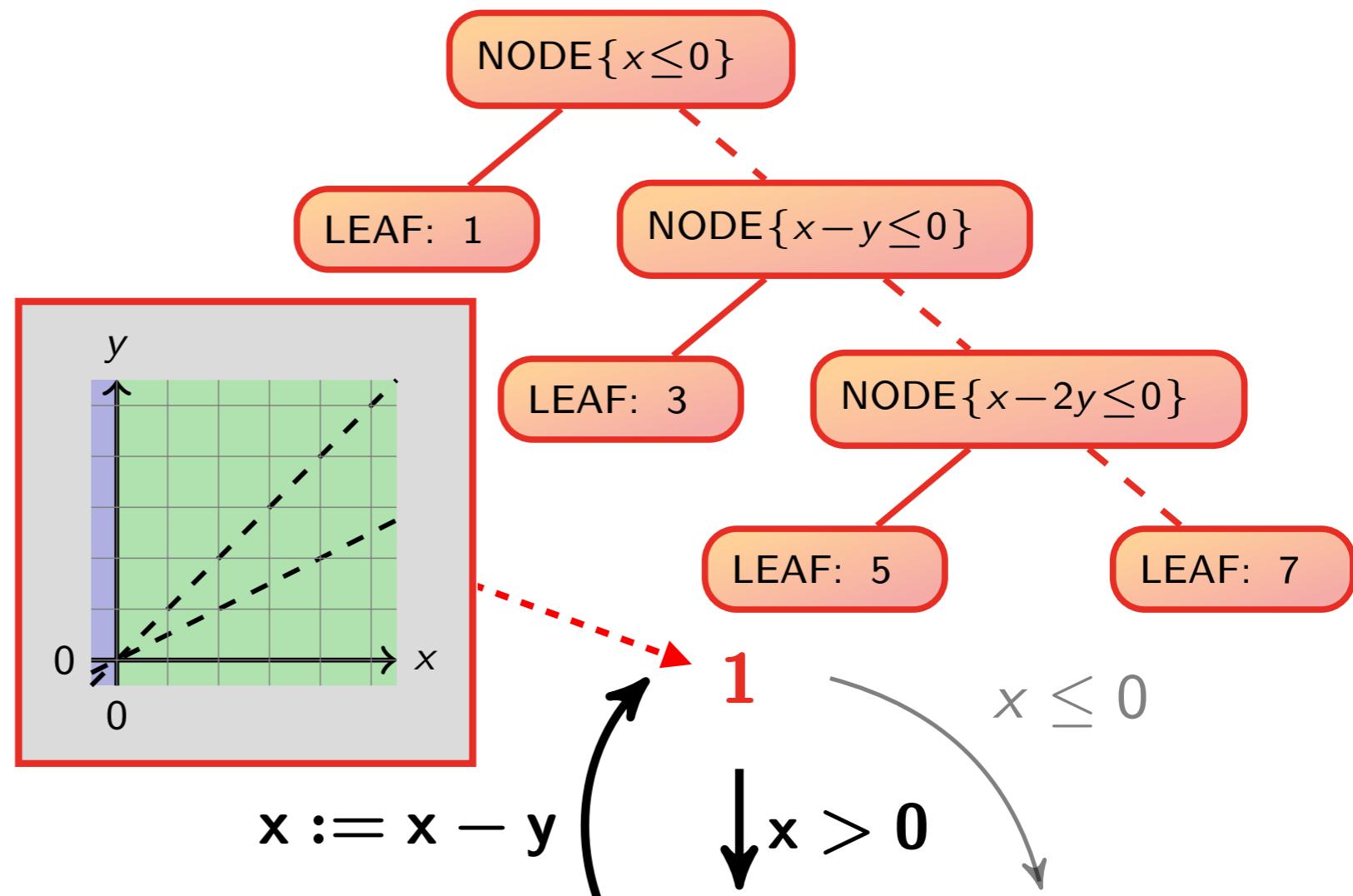
Example

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

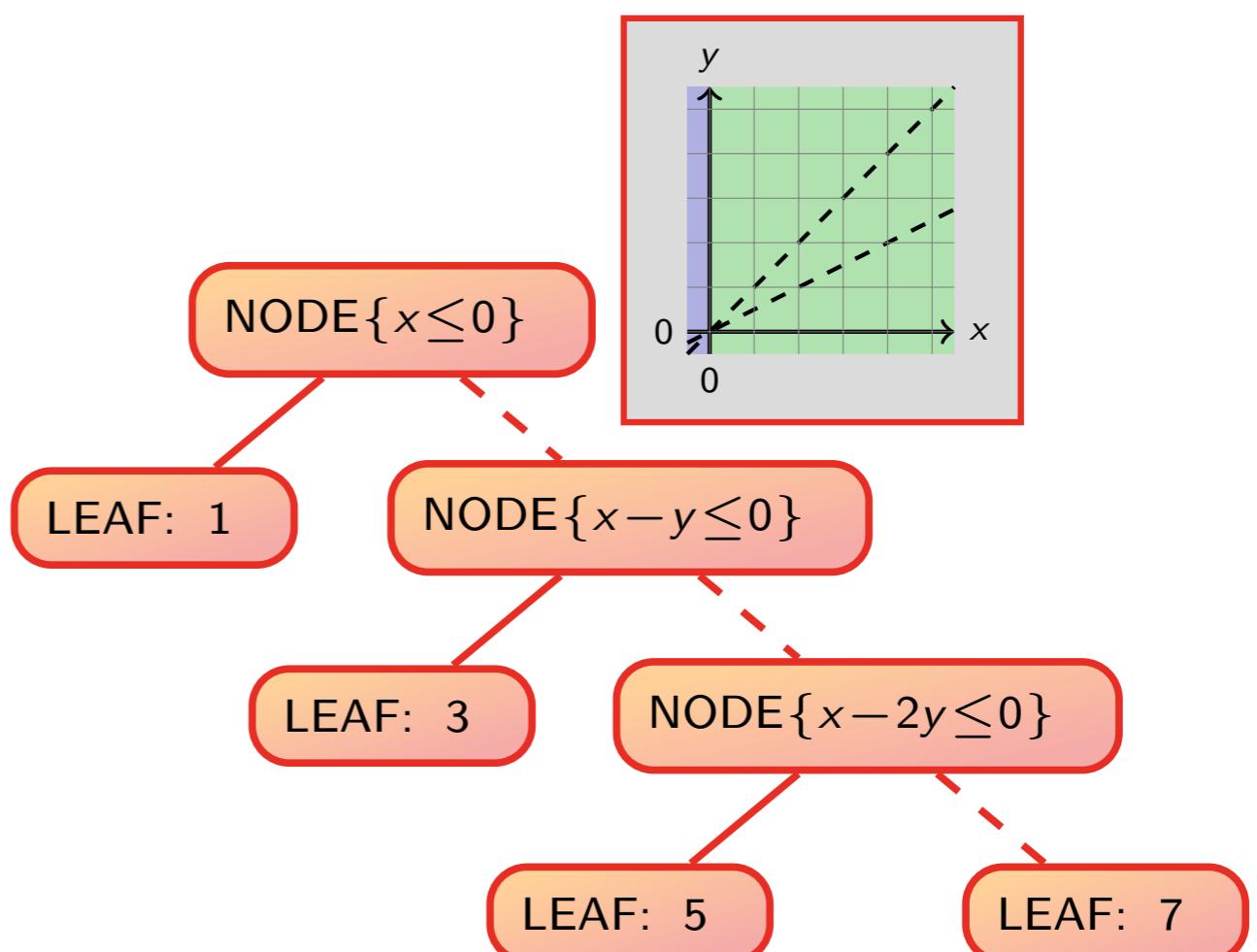
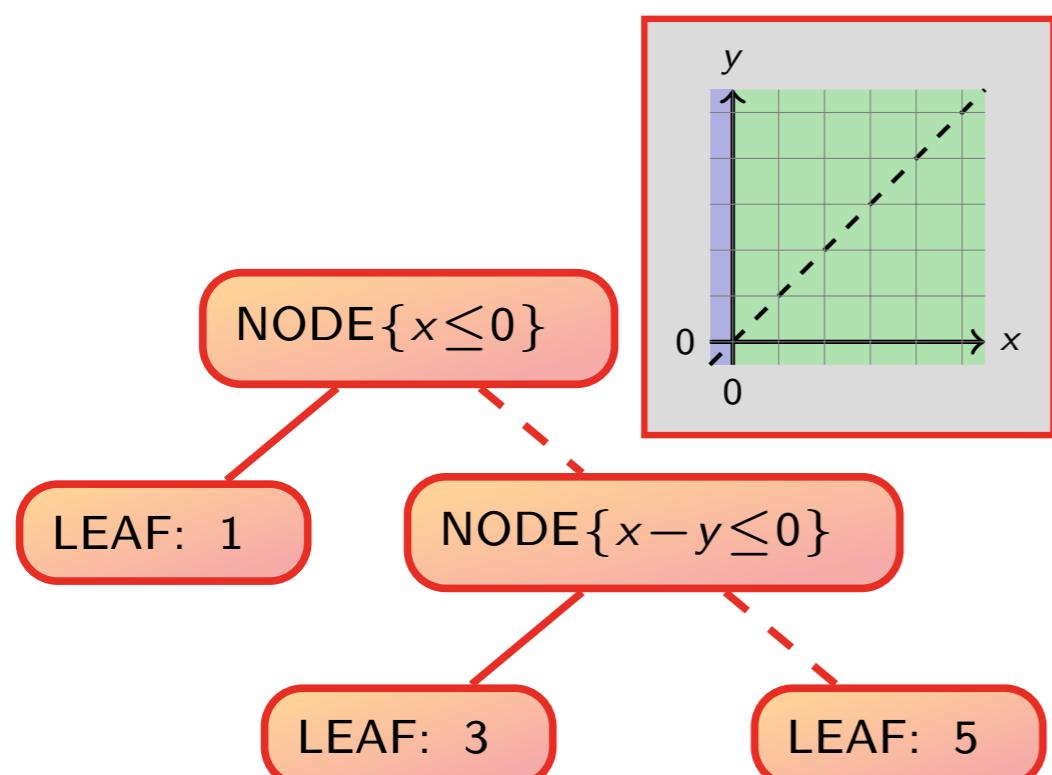


Example

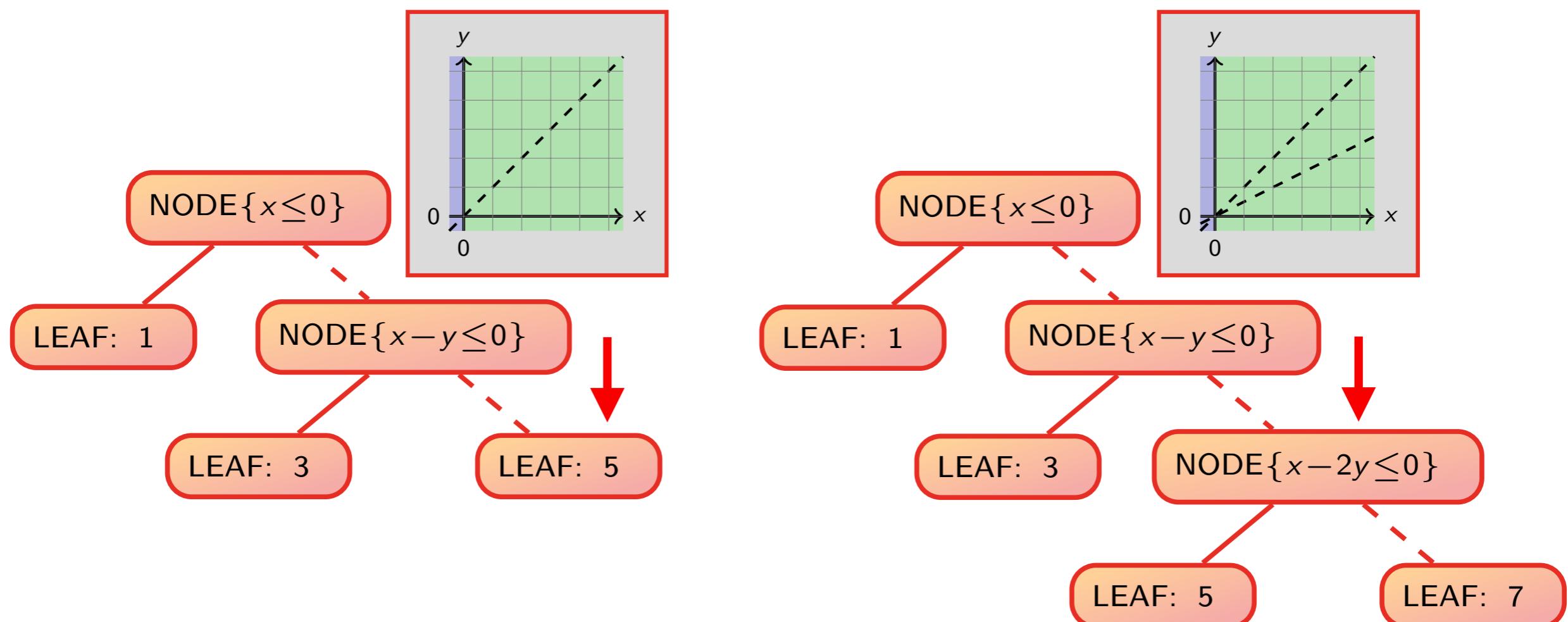
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



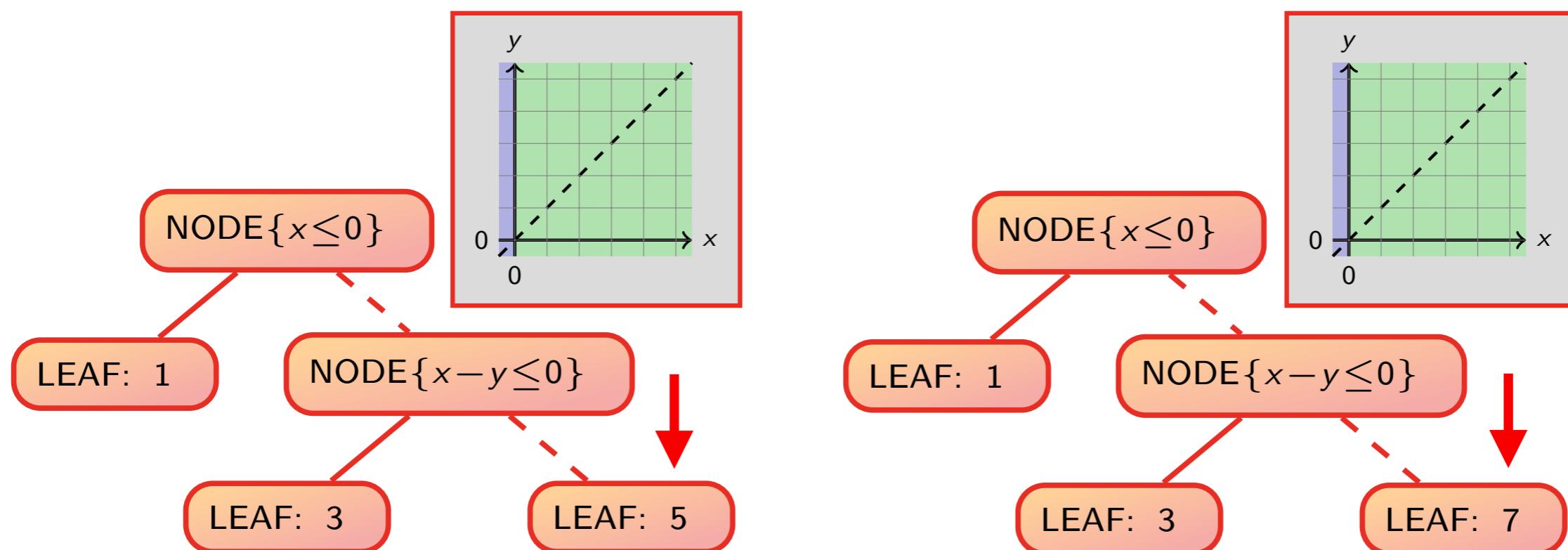
Widening



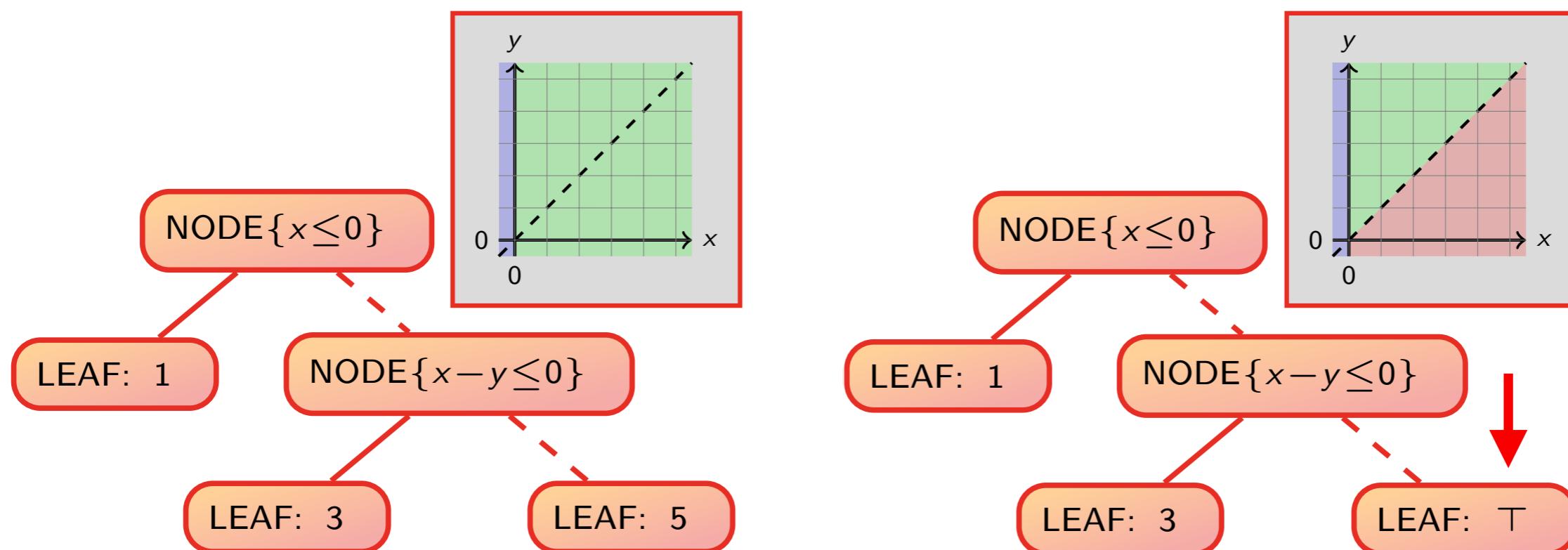
Widening



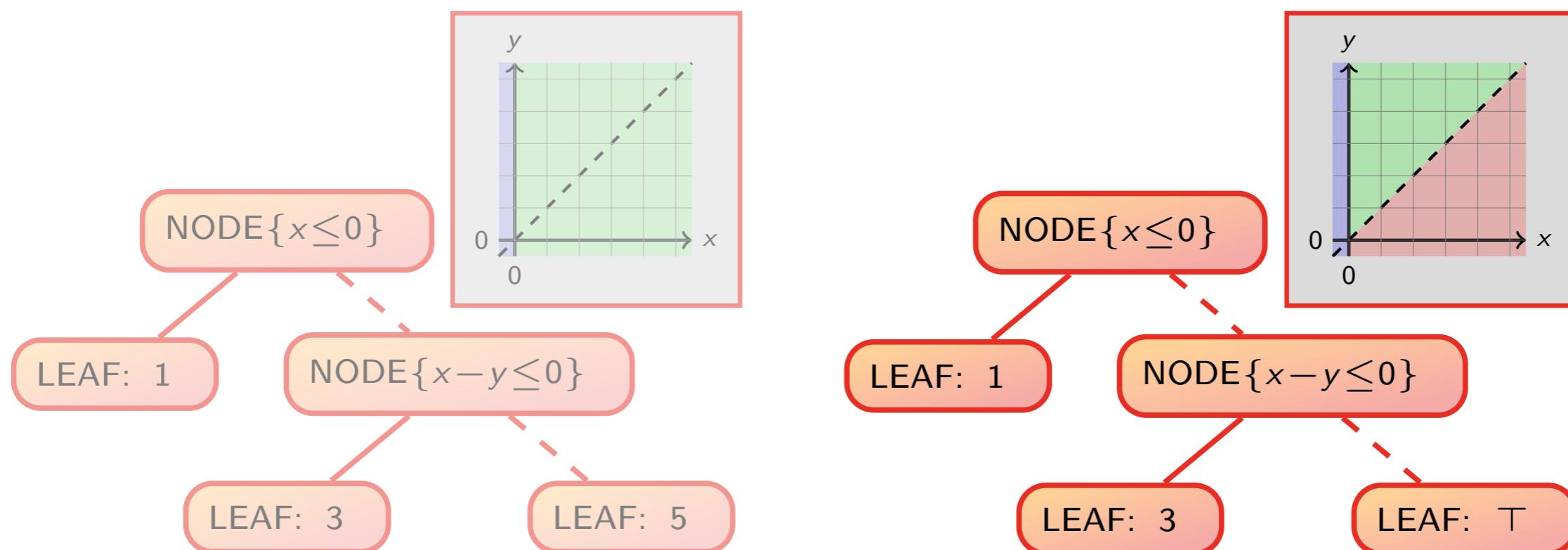
Widening



Widening

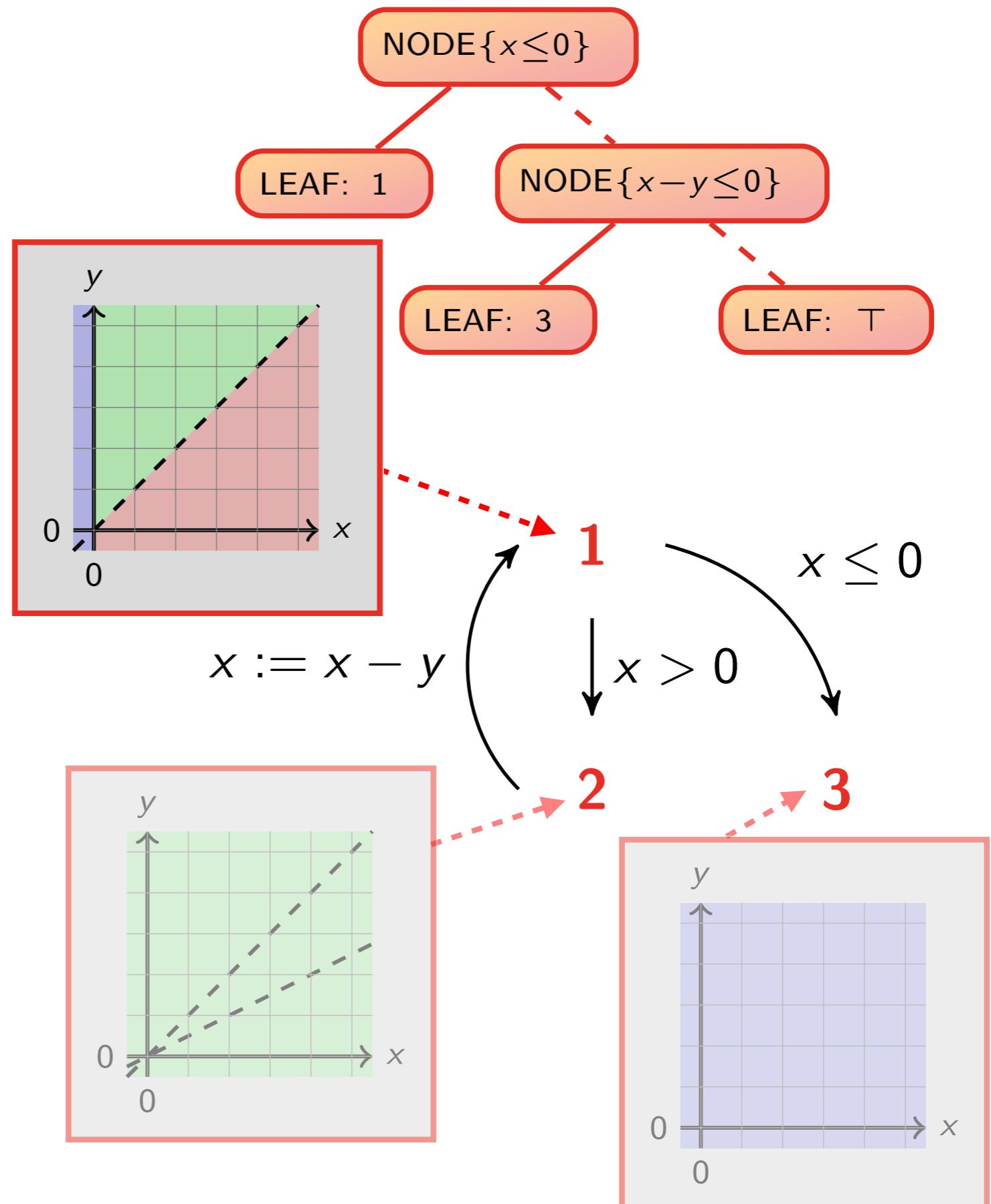


Widening



Example

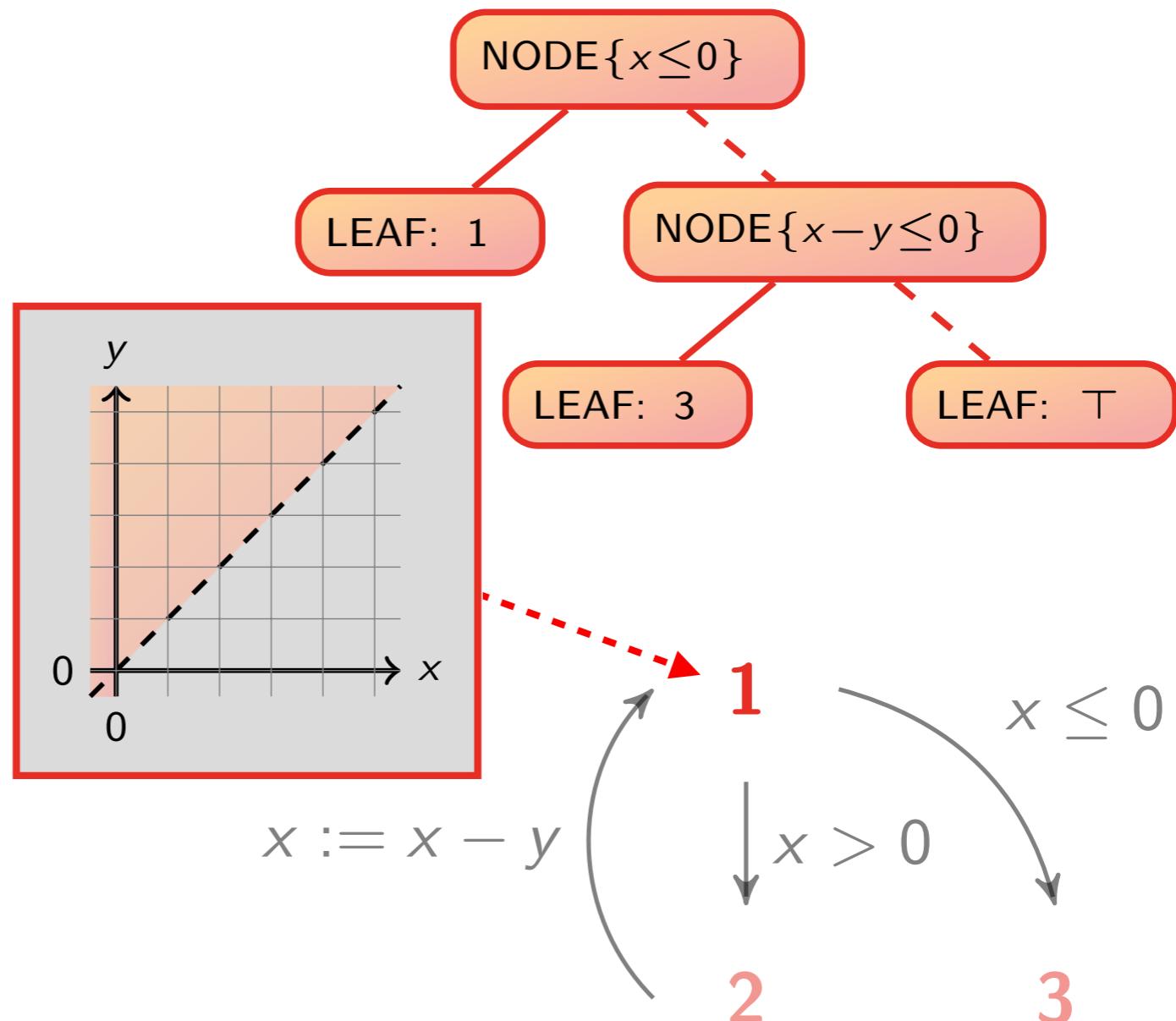
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



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**

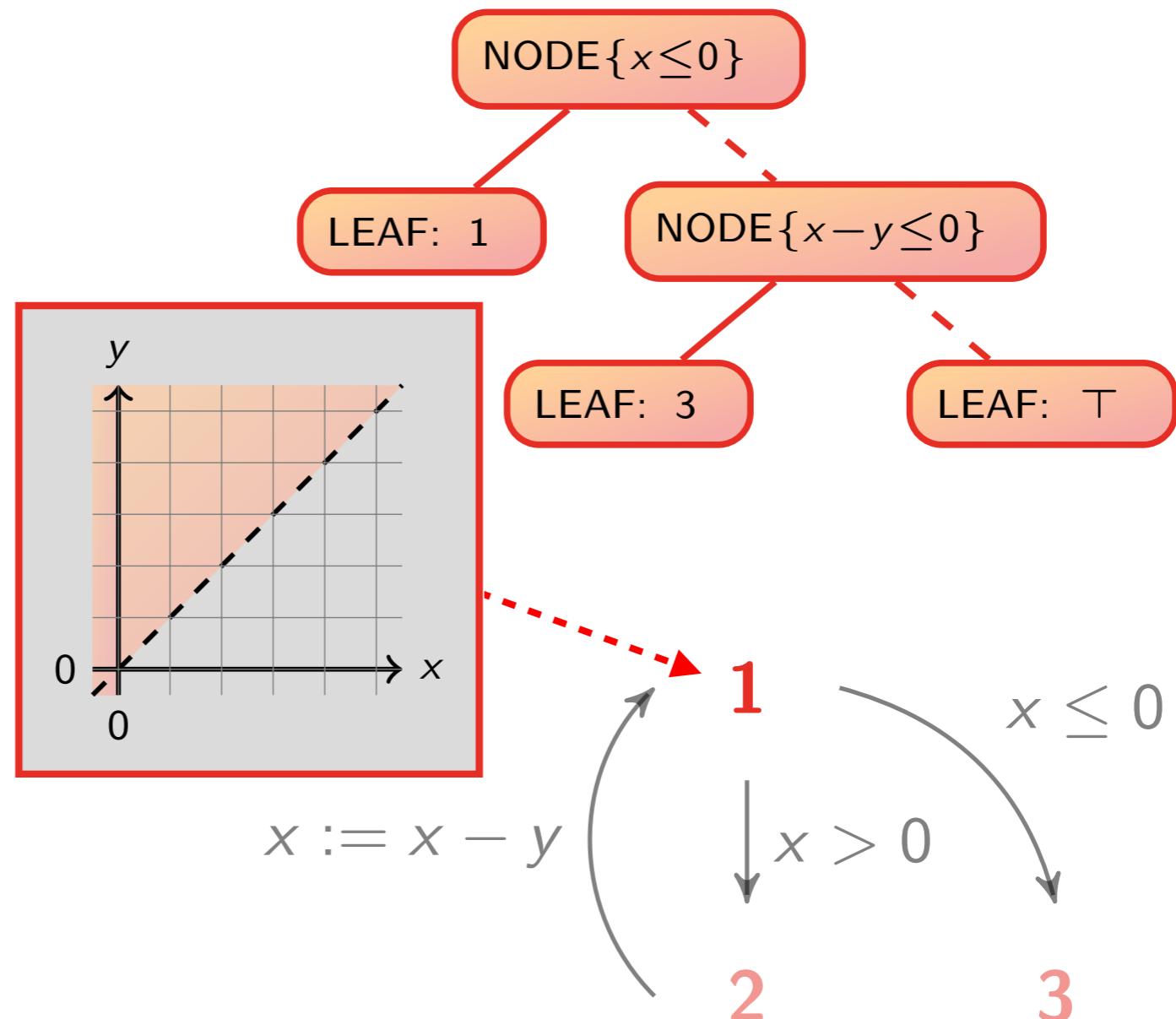


Example

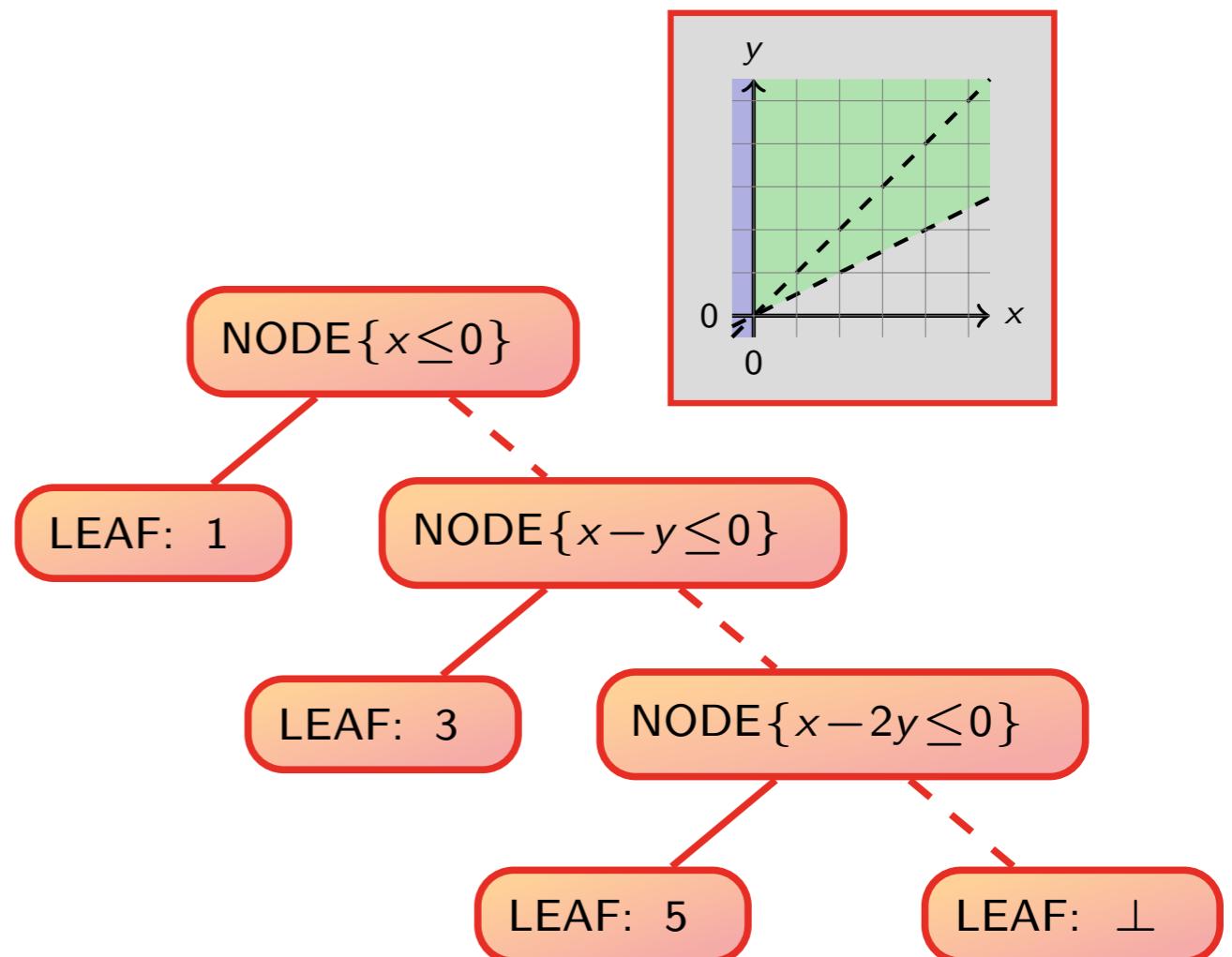
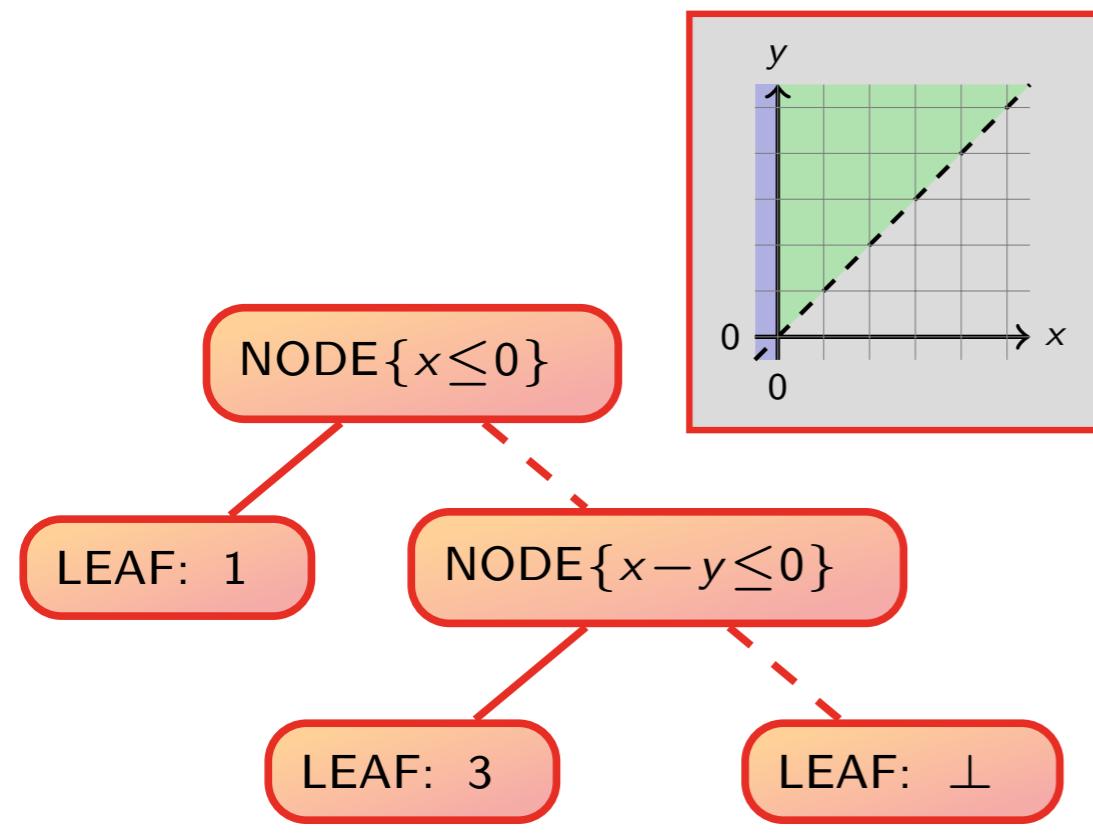
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```

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

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



Better Widening



Precise Widening Operators
for Convex Polyhedra*

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

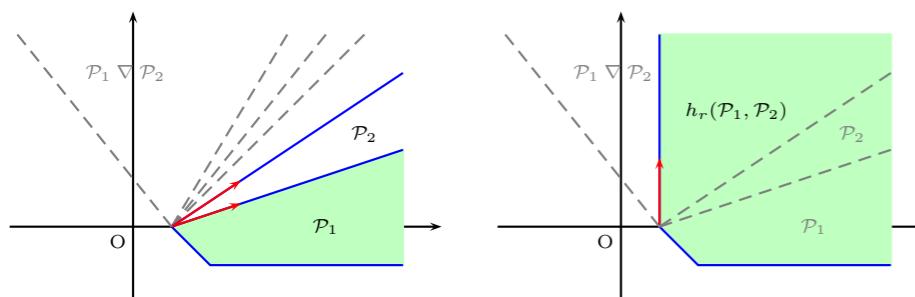
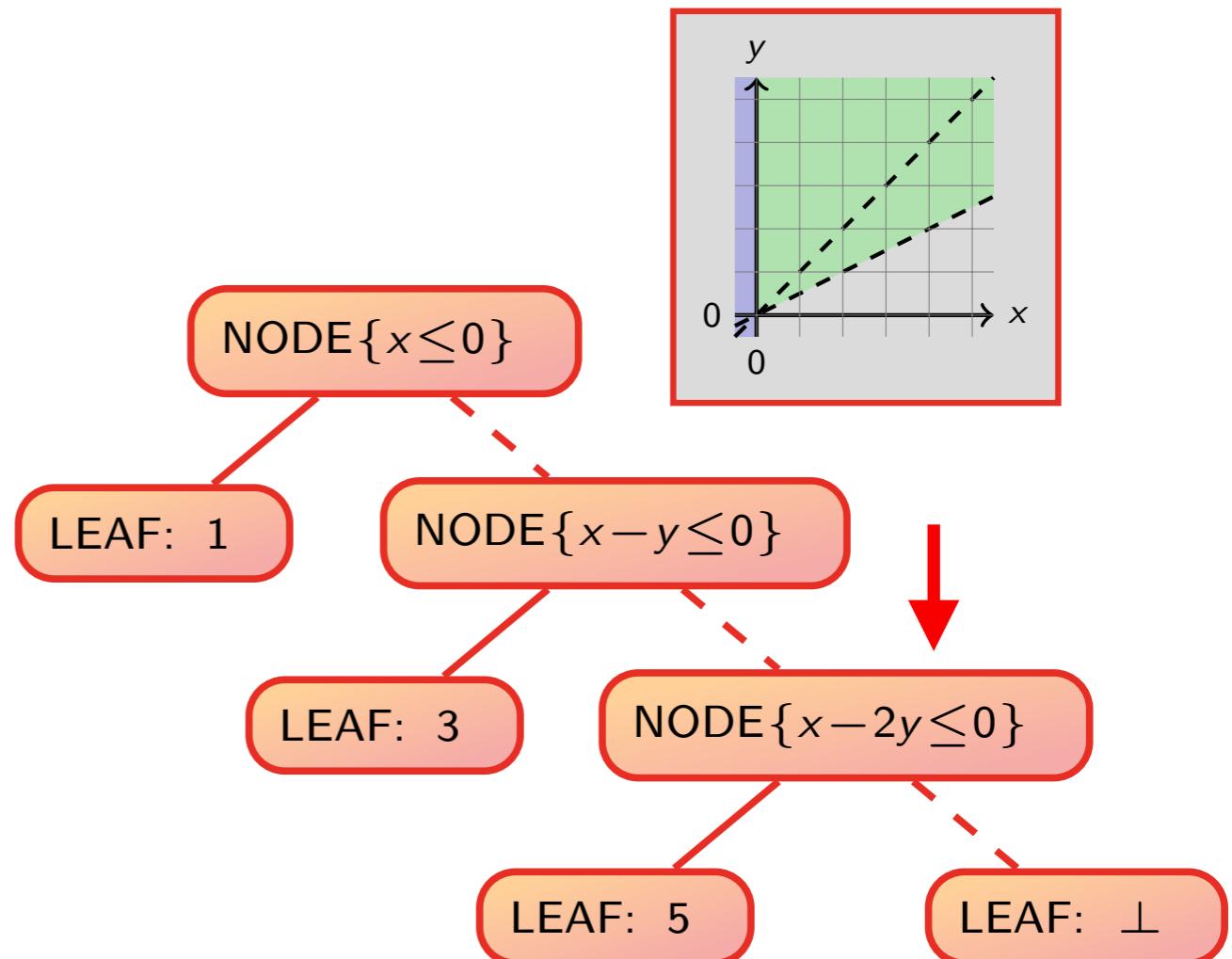
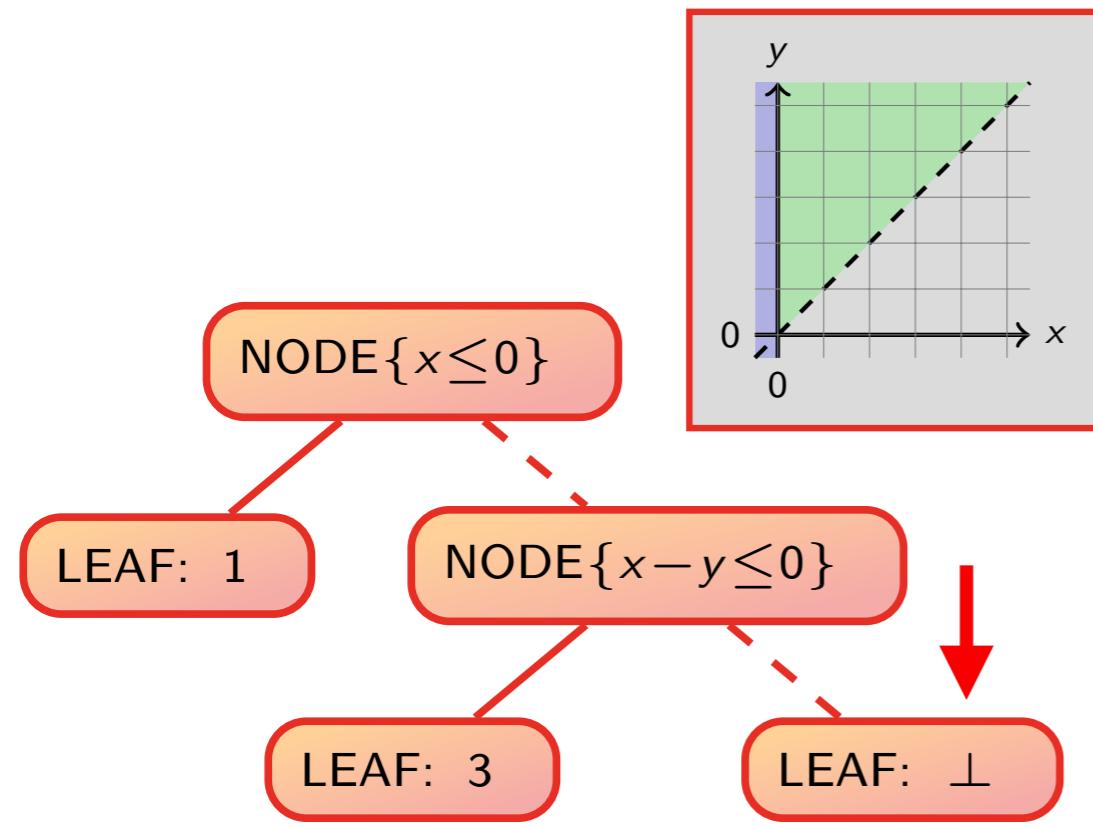


Fig. 2. The heuristics h_r improving on the standard widening.

Better Widening



Precise Widening Operators
for Convex Polyhedra*

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

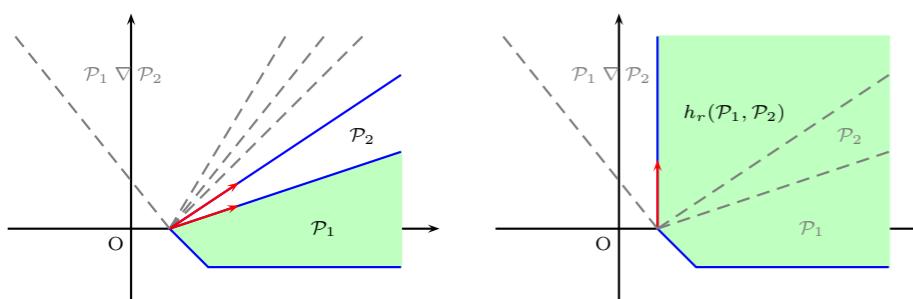
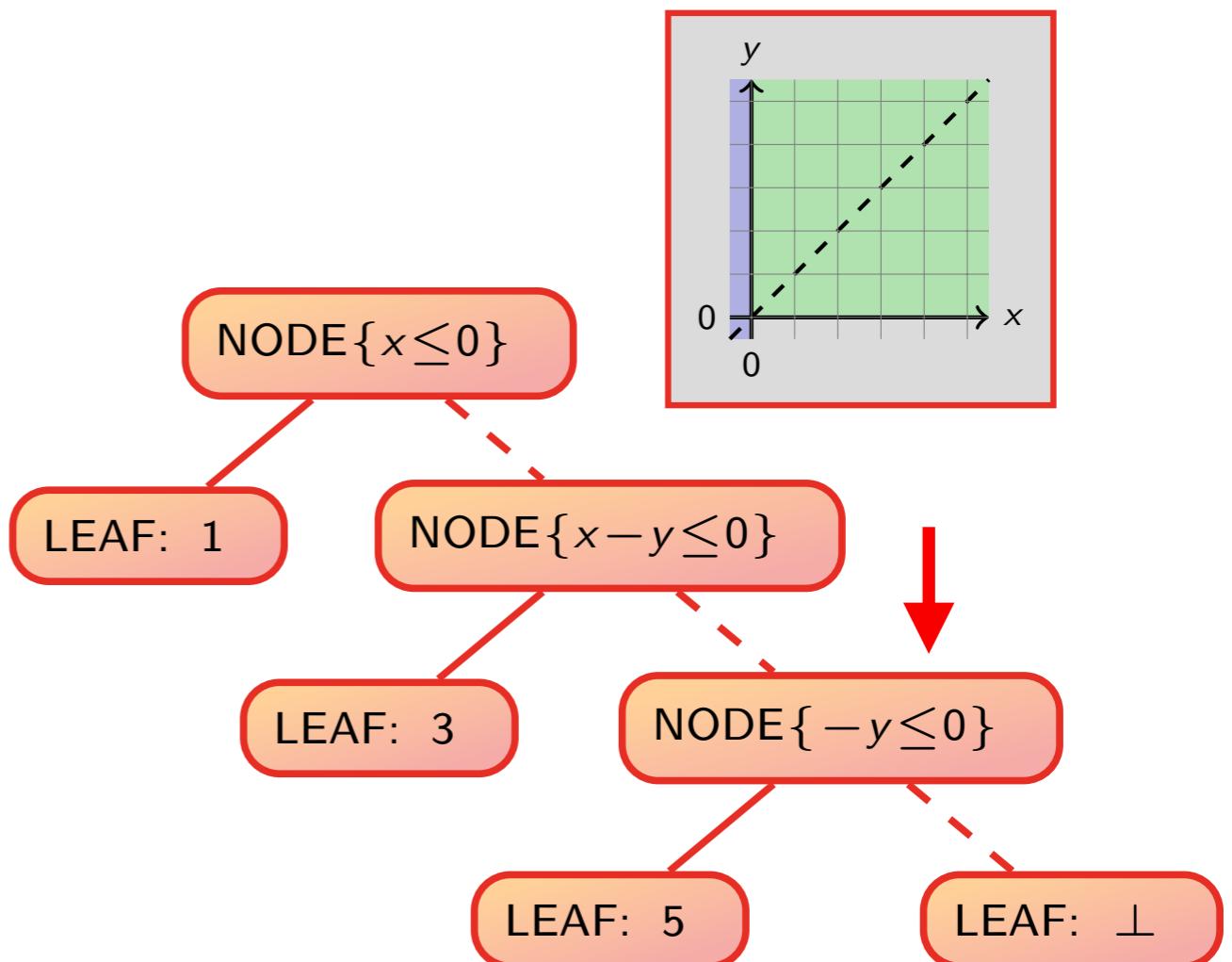
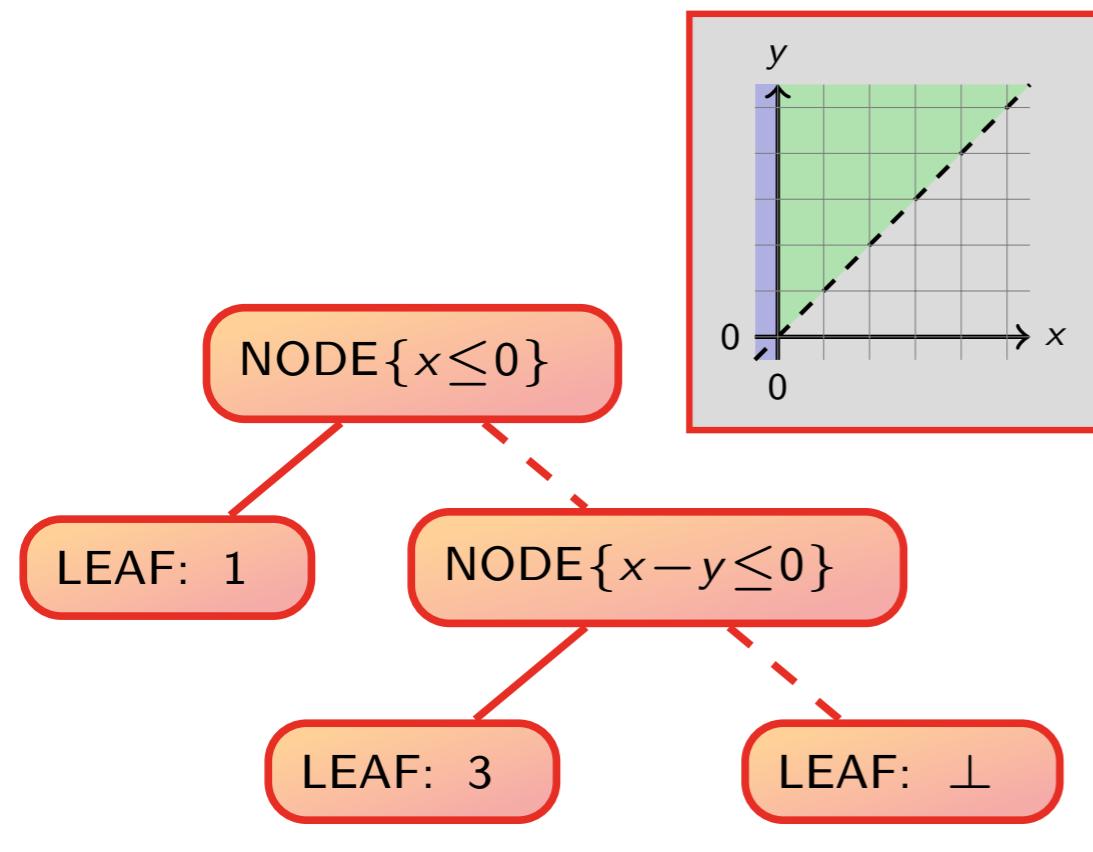


Fig. 2. The heuristics h_r improving on the standard widening.

Better Widening



Precise Widening Operators
for Convex Polyhedra*

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

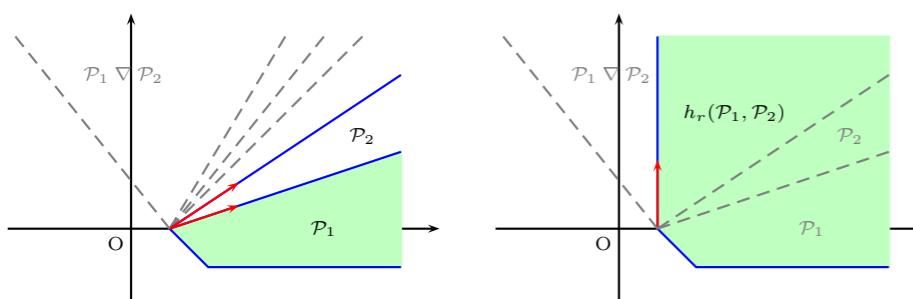
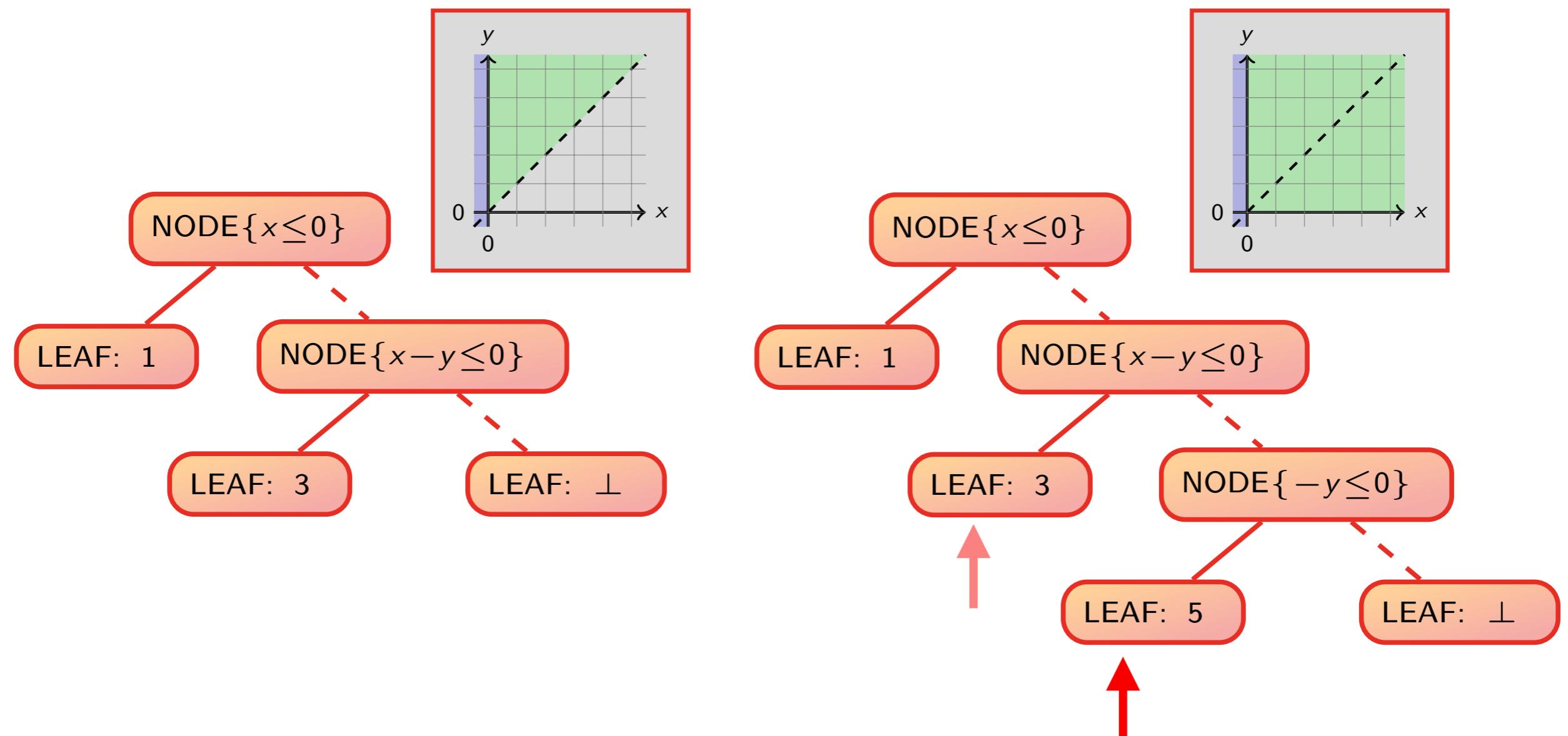
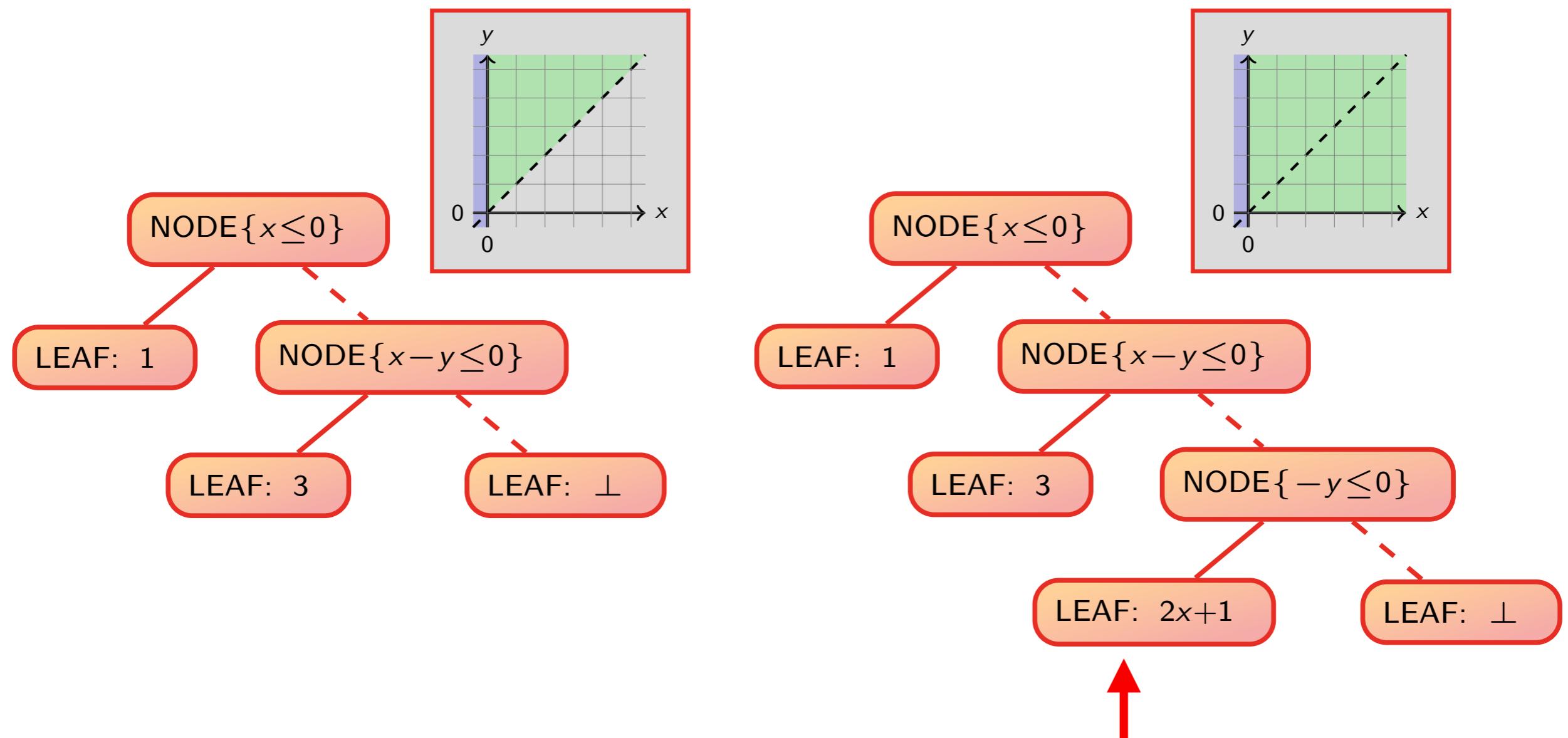


Fig. 2. The heuristics h_r improving on the standard widening.

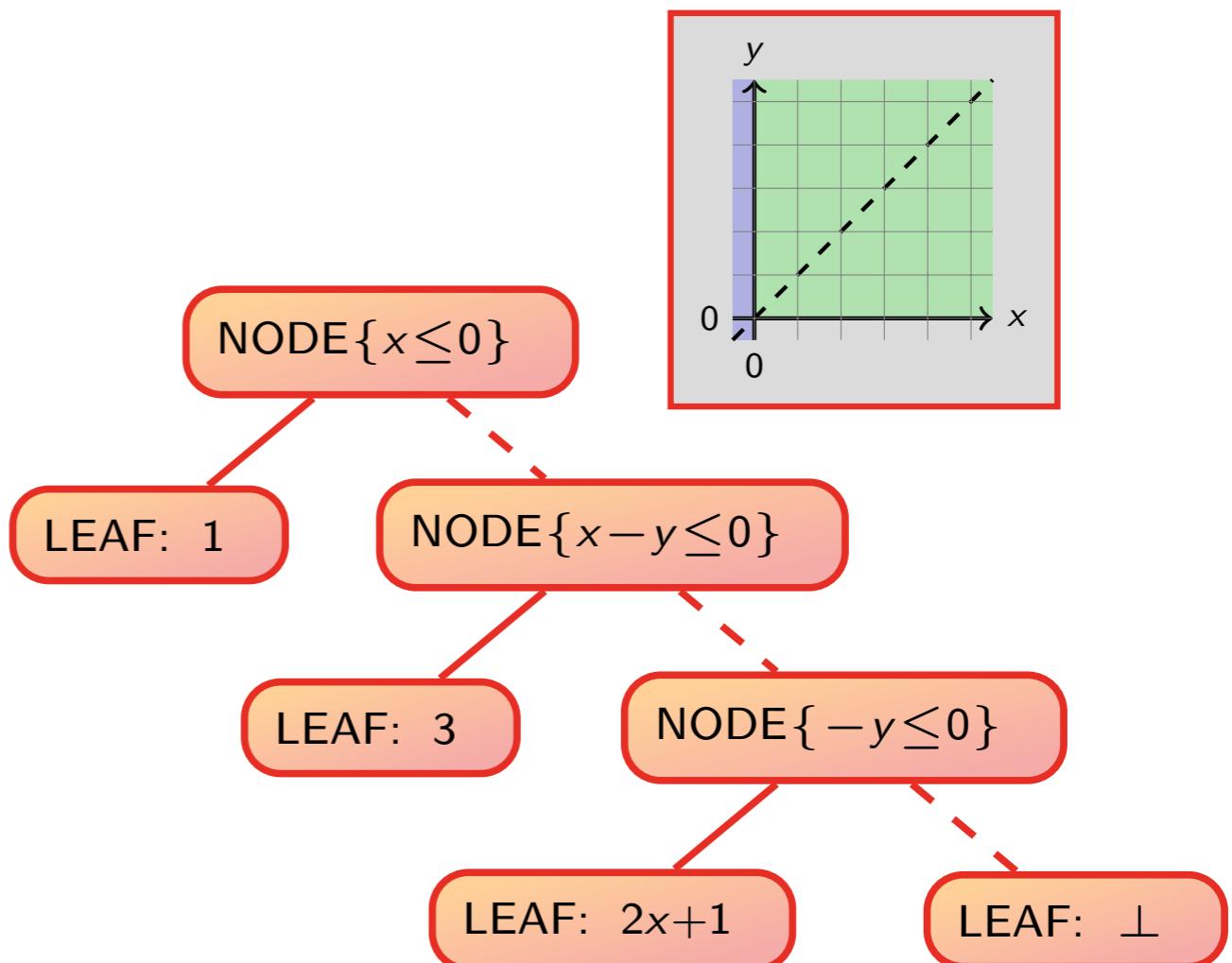
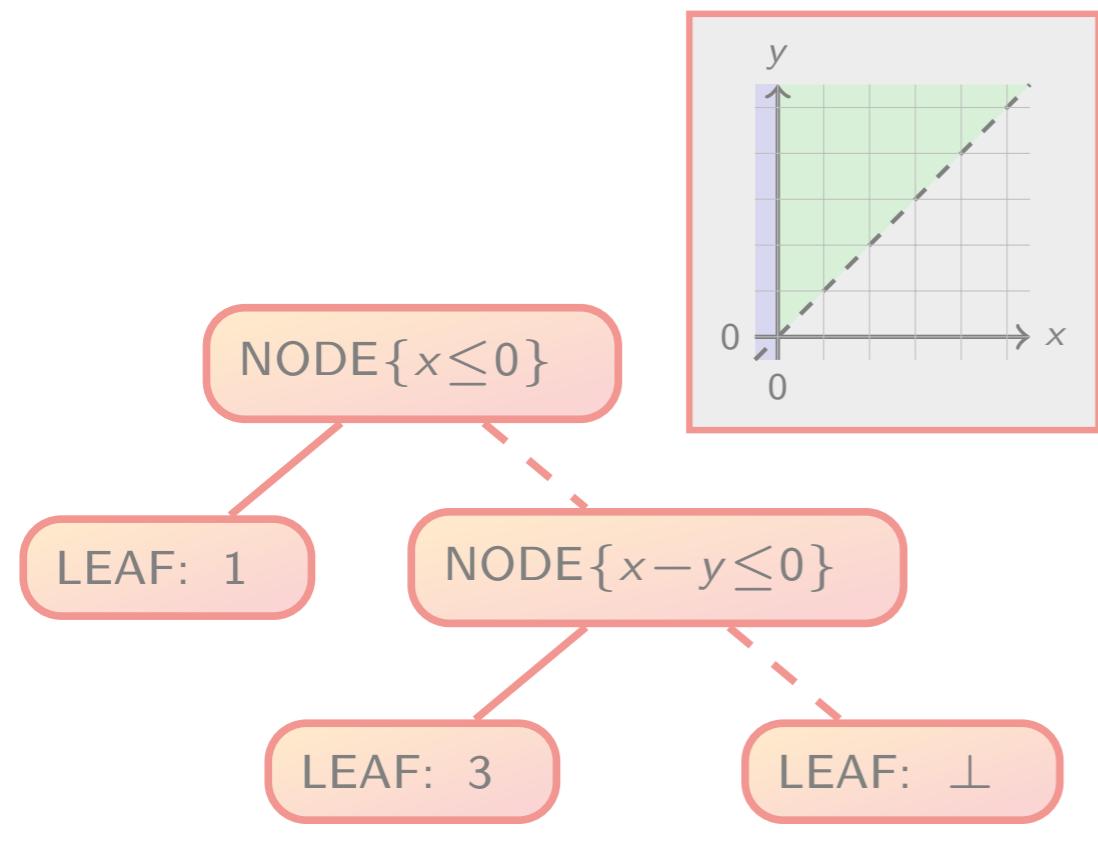
Better Widening



Better Widening

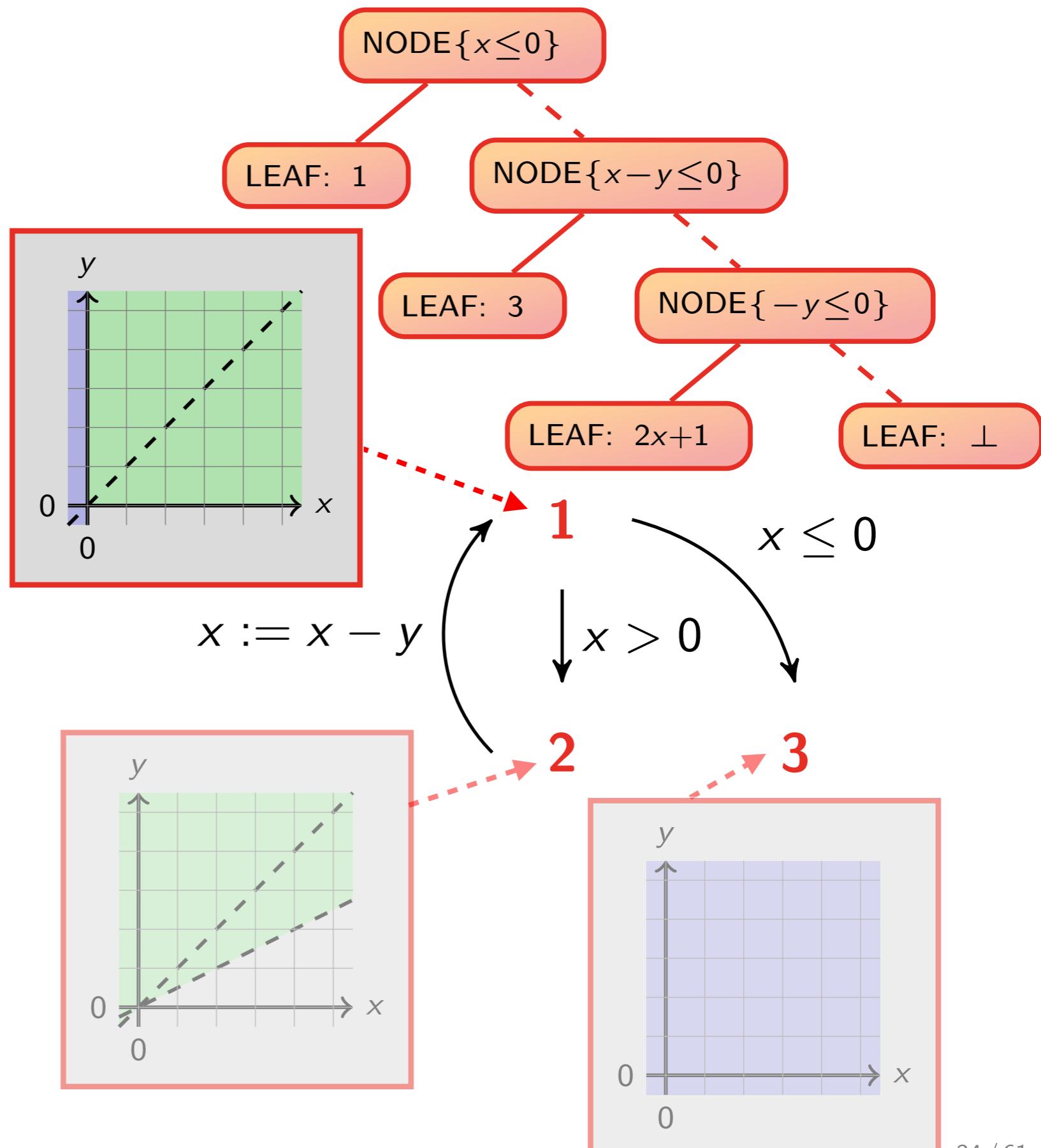


Better Widening



Example

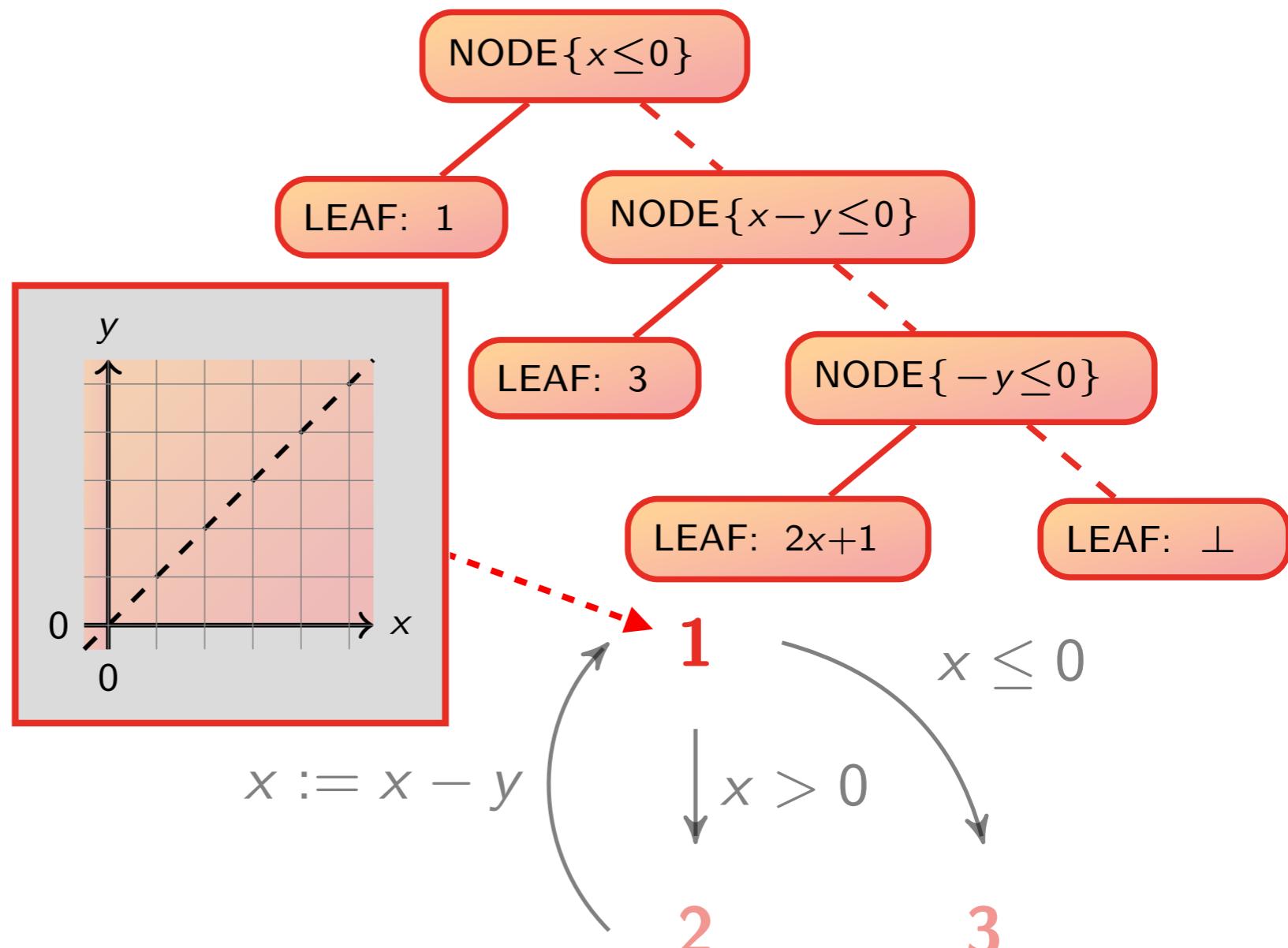
```
int : x, y
while 1(x > 0) do
  2x := x - y
od3
```



Example

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

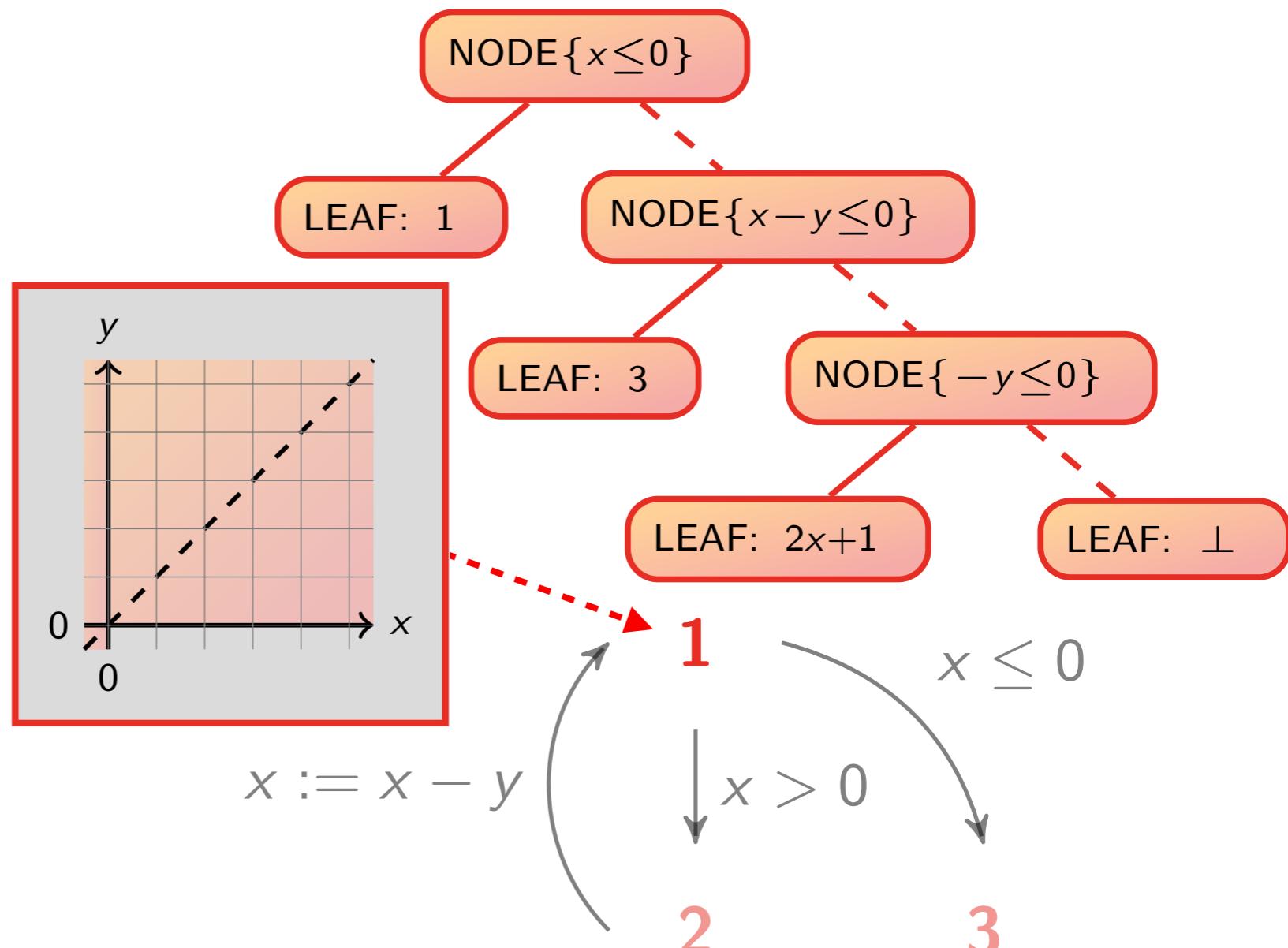
the analysis gives the **weakest precondition** $x \leq 0 \vee y > 0$



Example

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

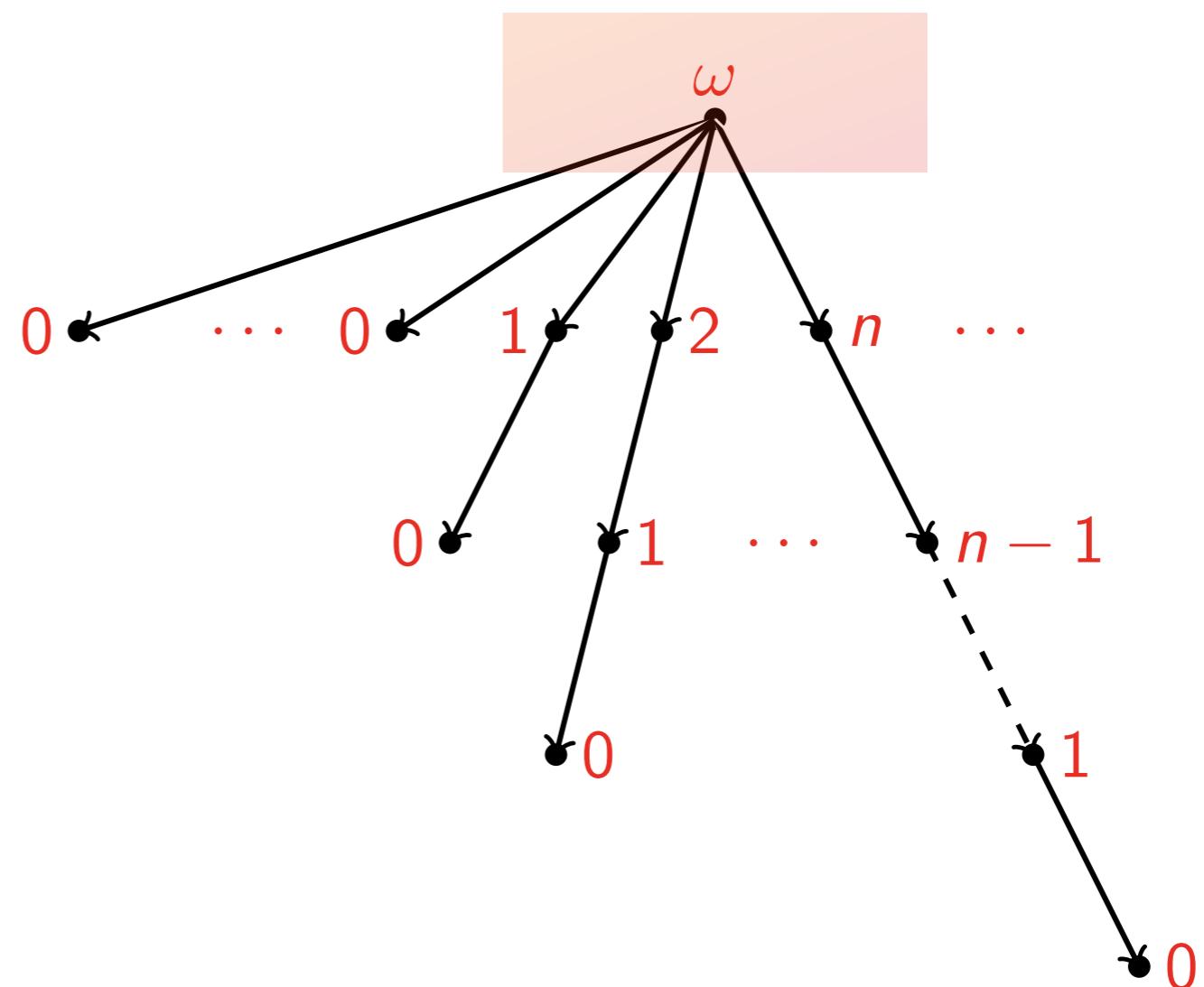
the analysis gives the **weakest precondition** $x \leq 0 \vee y > 0$



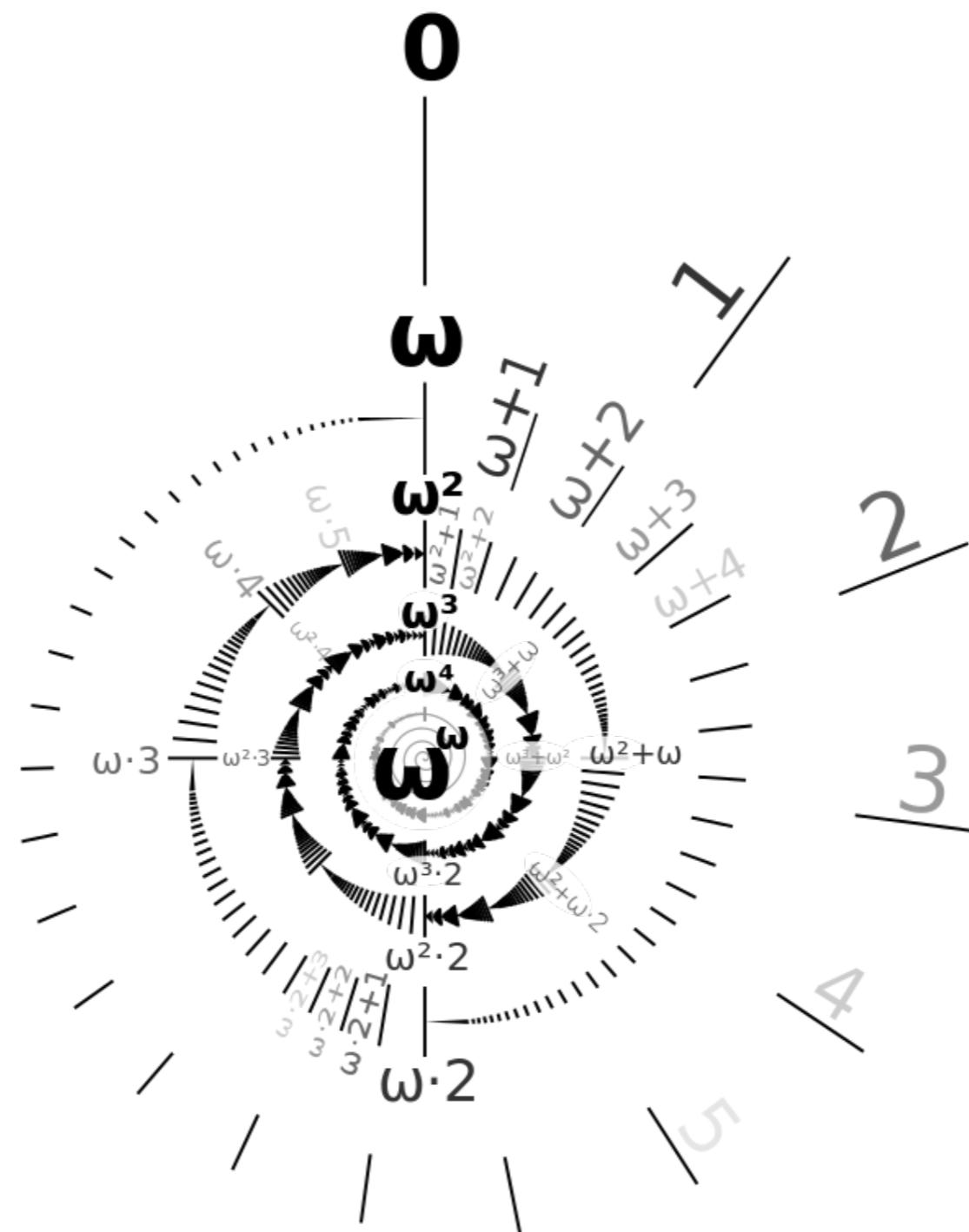
- **remark:** natural-valued ranking functions are **not sufficient!**

Example

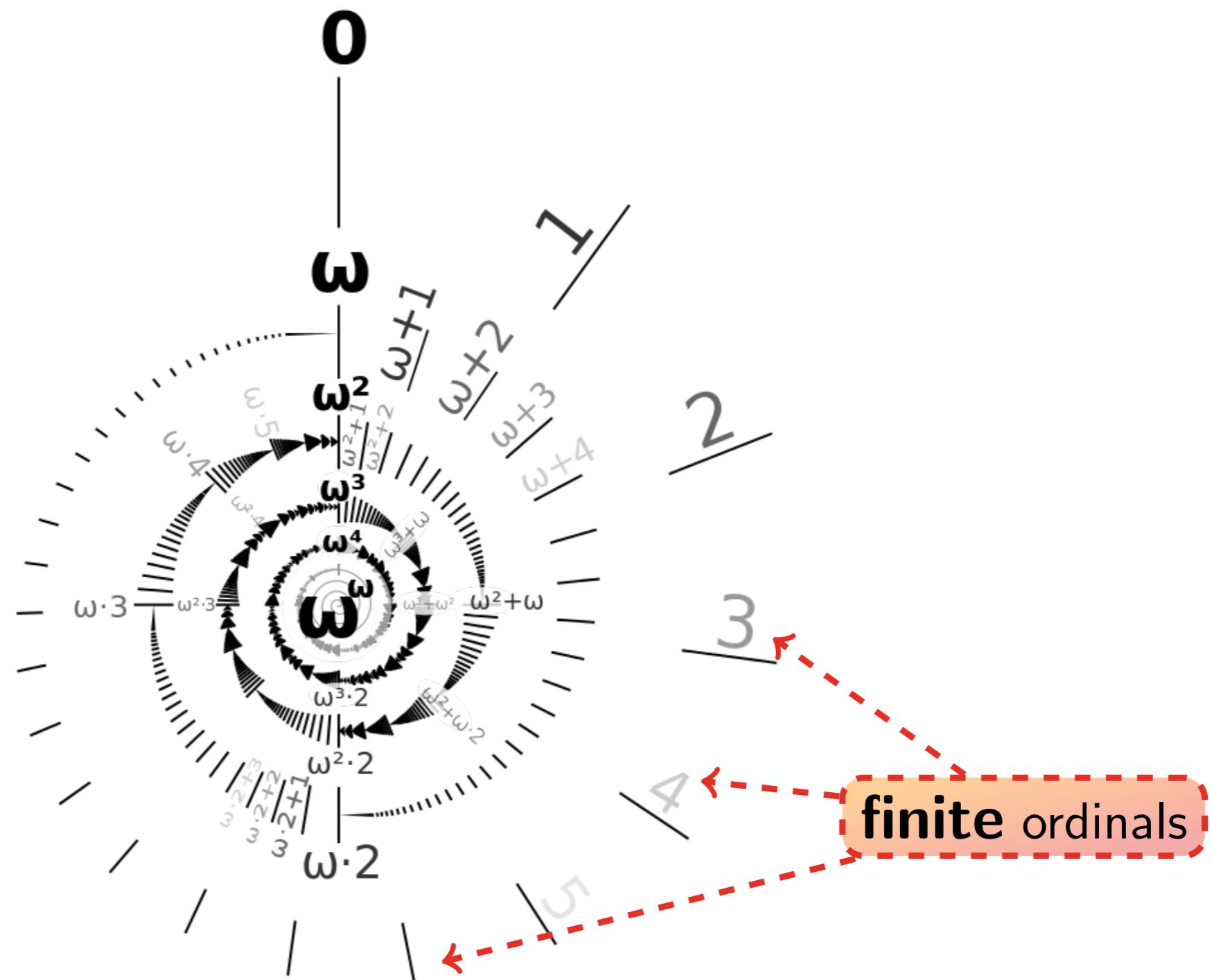
```
int : x
x := ?
while (x > 0) do
  x := x - 1
od
```



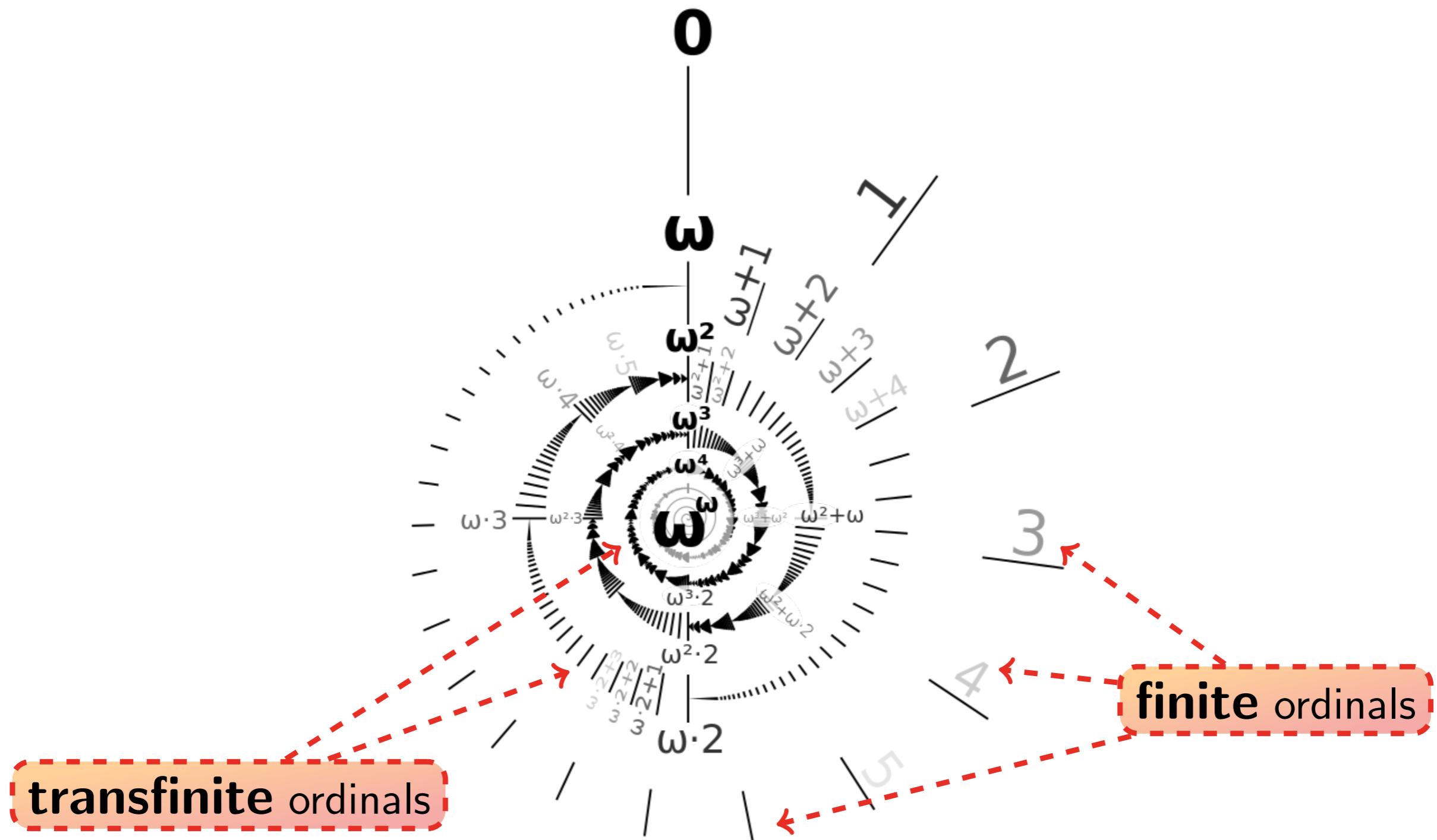
Ordinals



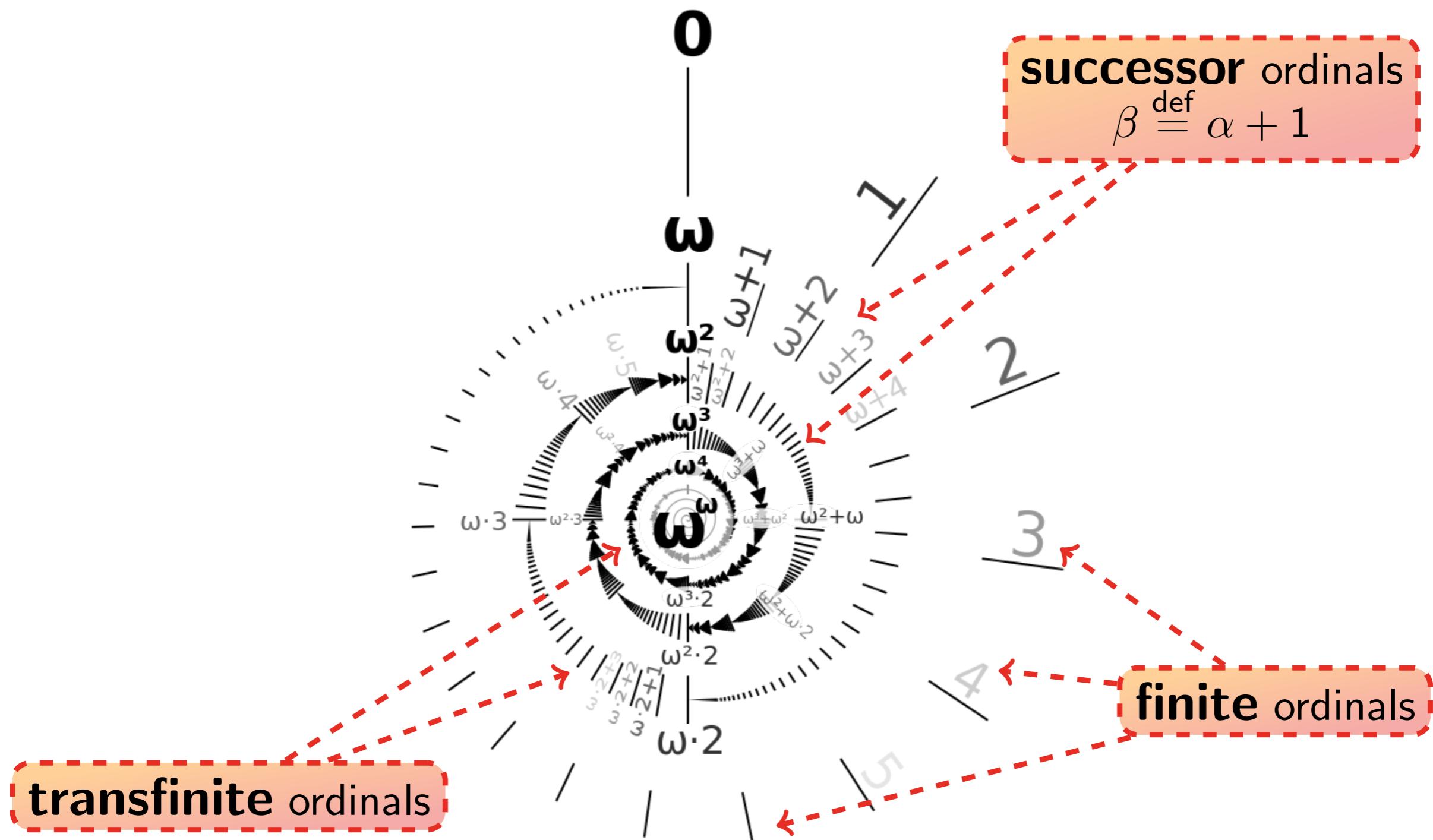
Ordinals



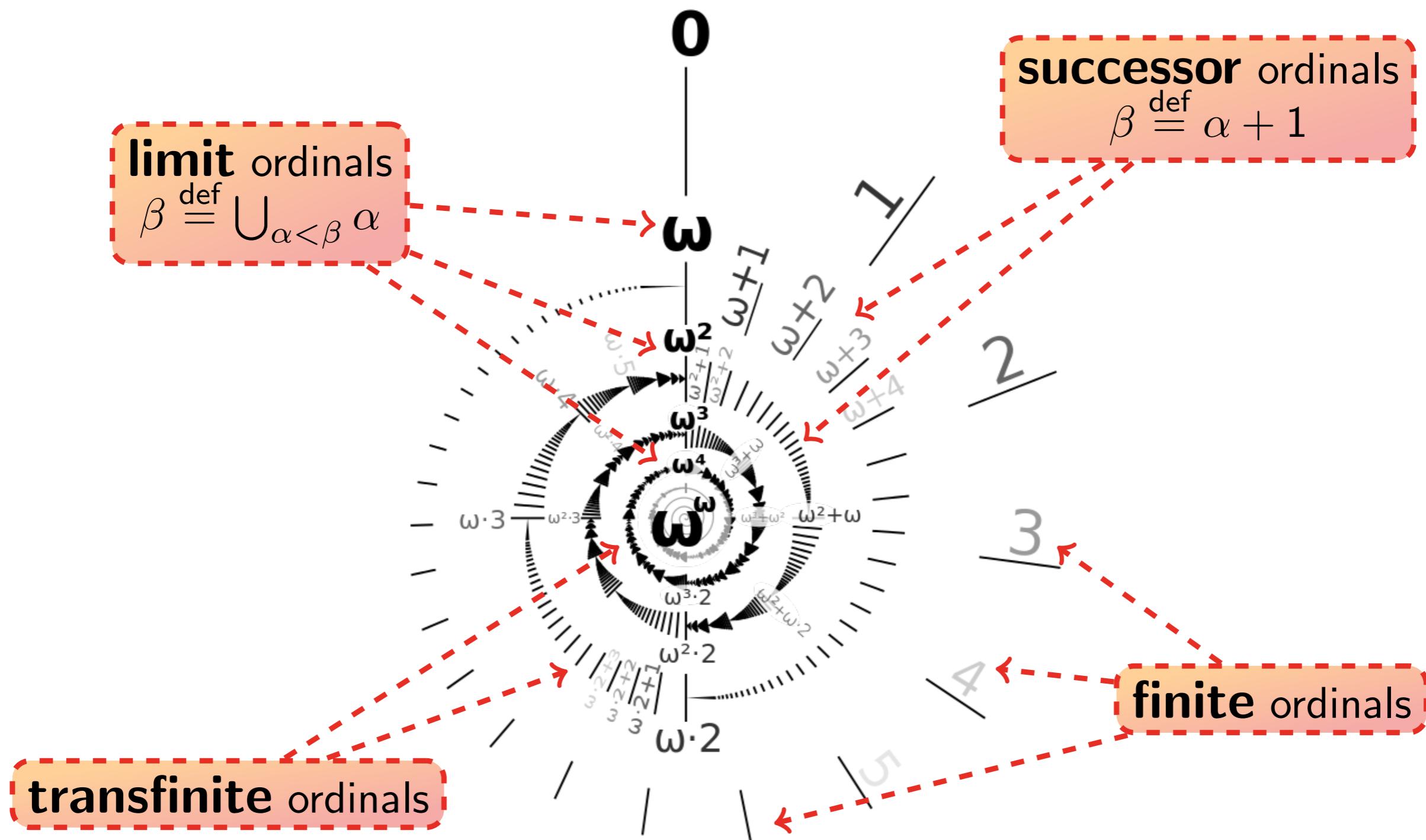
Ordinals

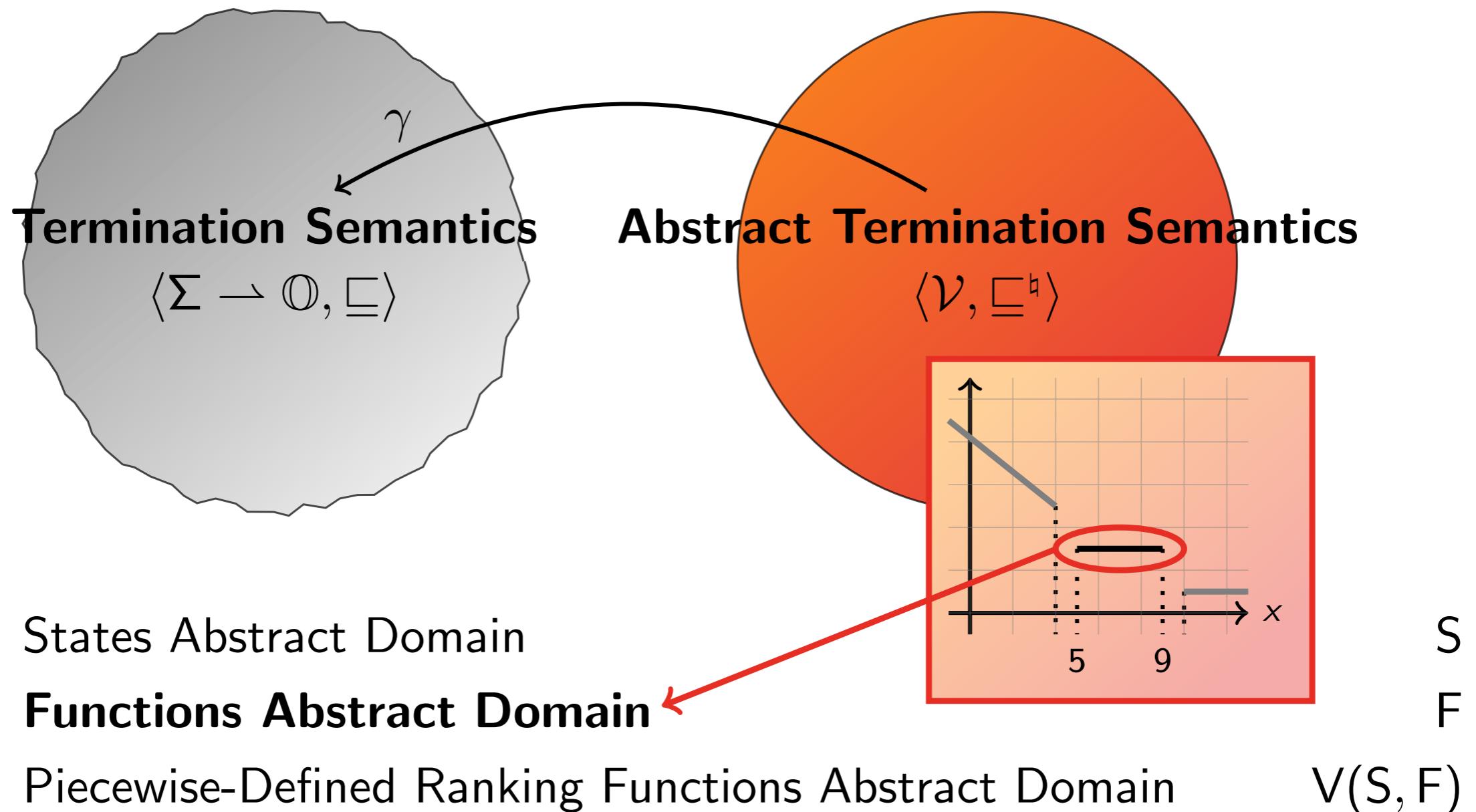


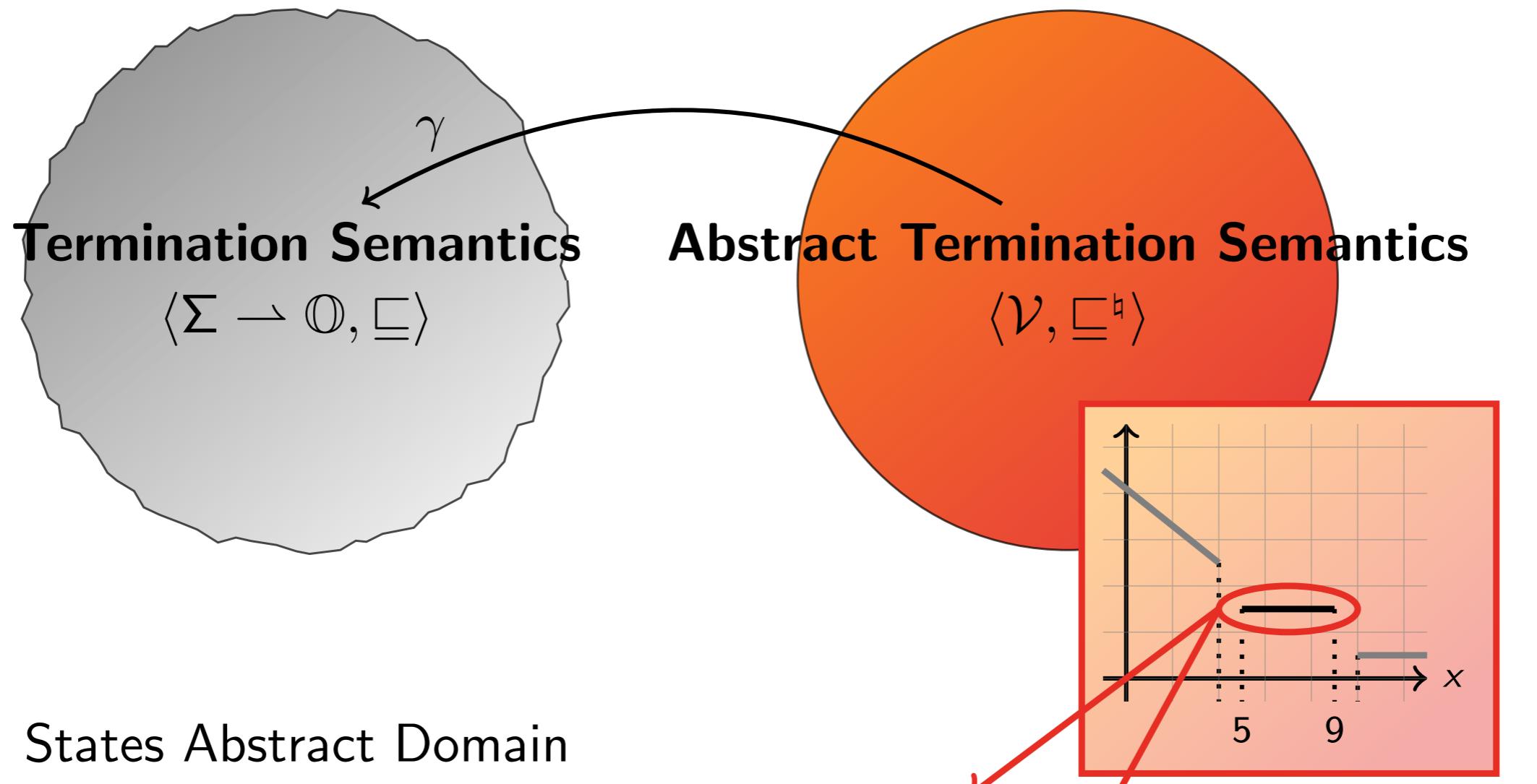
Ordinals



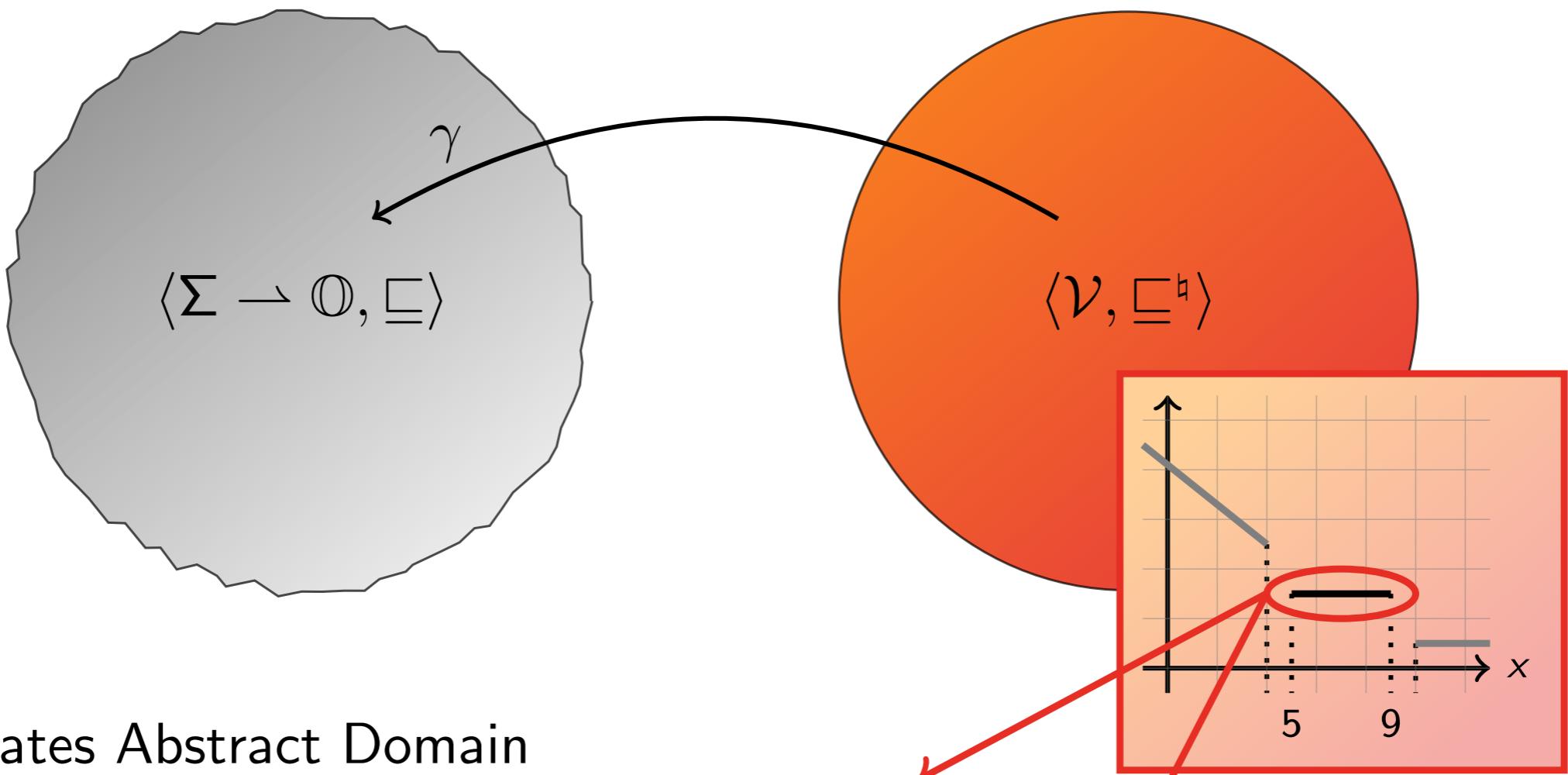
Ordinals







- States Abstract Domain
- **Natural-Valued Functions Abstract Domain**
- **Ordinal-Valued Functions Abstract Domain**
- 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

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

$\Downarrow \quad x_1 := ?$

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

Backward Non-Deterministic Assignments

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

Example

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

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

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

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
        else / * x1 < 0 * /
        if 7( ? ) then
            8x1 := x1 + 1
        else
            9x2 := x2 - 1
    od11
10x1 := ?

```

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

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

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

Guarantee and Recurrence Properties

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

Caterina Urban and Antoine Miné
ENS & CNRS & INRIA, France
(urban,mine)@inria.fr

Abstract. We present new static analysis methods for proving fairness properties of programs. In particular, with reference to the hierarchy of temporal properties proposed by Manna 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 proposed by Cousot and Cousot. Specifically, static analyses of guarantees and recurrence temporal properties are automatically derived by abstraction of the program operational trace semantics.
These methods automatically infer sufficient preconditions for the temporal properties by creating relating abstract domains based on problem-defined ranking functions. We compare these abstract domains with two abstract operators, including a dual ordering.
To illustrate the potential of the proposed methods, we have implemented a research prototype static analyzer, for programs written in a C-like syntax, that yielded interesting preliminary results.

1 Introduction

Temporal properties play a major role in the specification and verification of programs. The hierarchy of temporal properties proposed by Manna and Pnueli [14] distinguishes four basic classes:

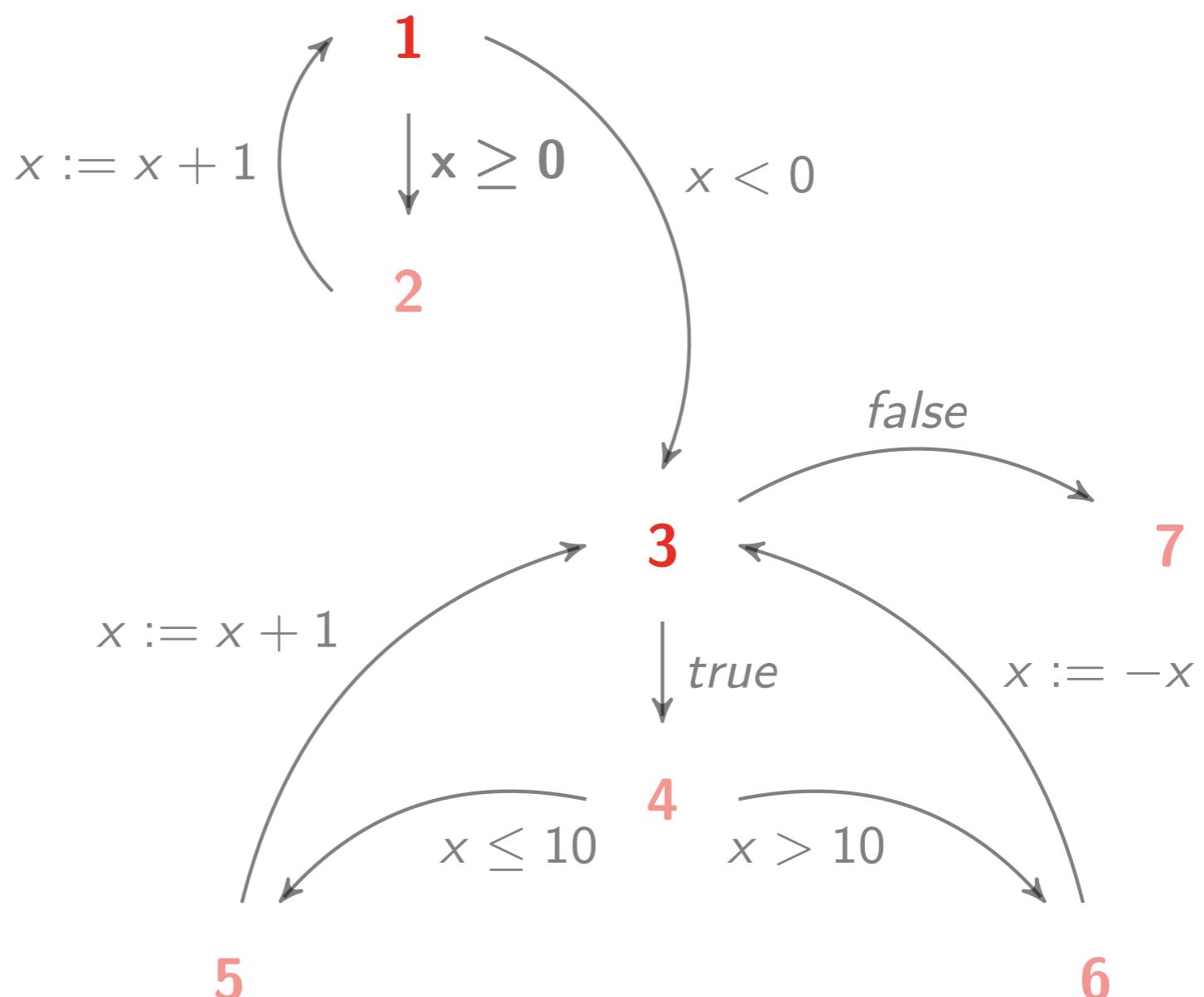
- safety properties: “something good always happens”, i.e., the program never reaches an unacceptable state (e.g., partial correctness, mutual exclusion);
- progress properties: “something good happens at least once”, i.e., the program eventually reaches a desirable state (e.g., termination);
- recurrence 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 Abstraction Interpretation [2], a general theory of semantic

^{*} The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 260448 (ARTEMIS project MiB4E) (see Article II.8 of the EU-Governing Agreement).

Example

```
int : x, y
while 1( $x \geq 0$ ) do
  2 $x := x + 1$ 
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 $x := x + 1$ 
  else
    6 $x := -x$ 
od7
```

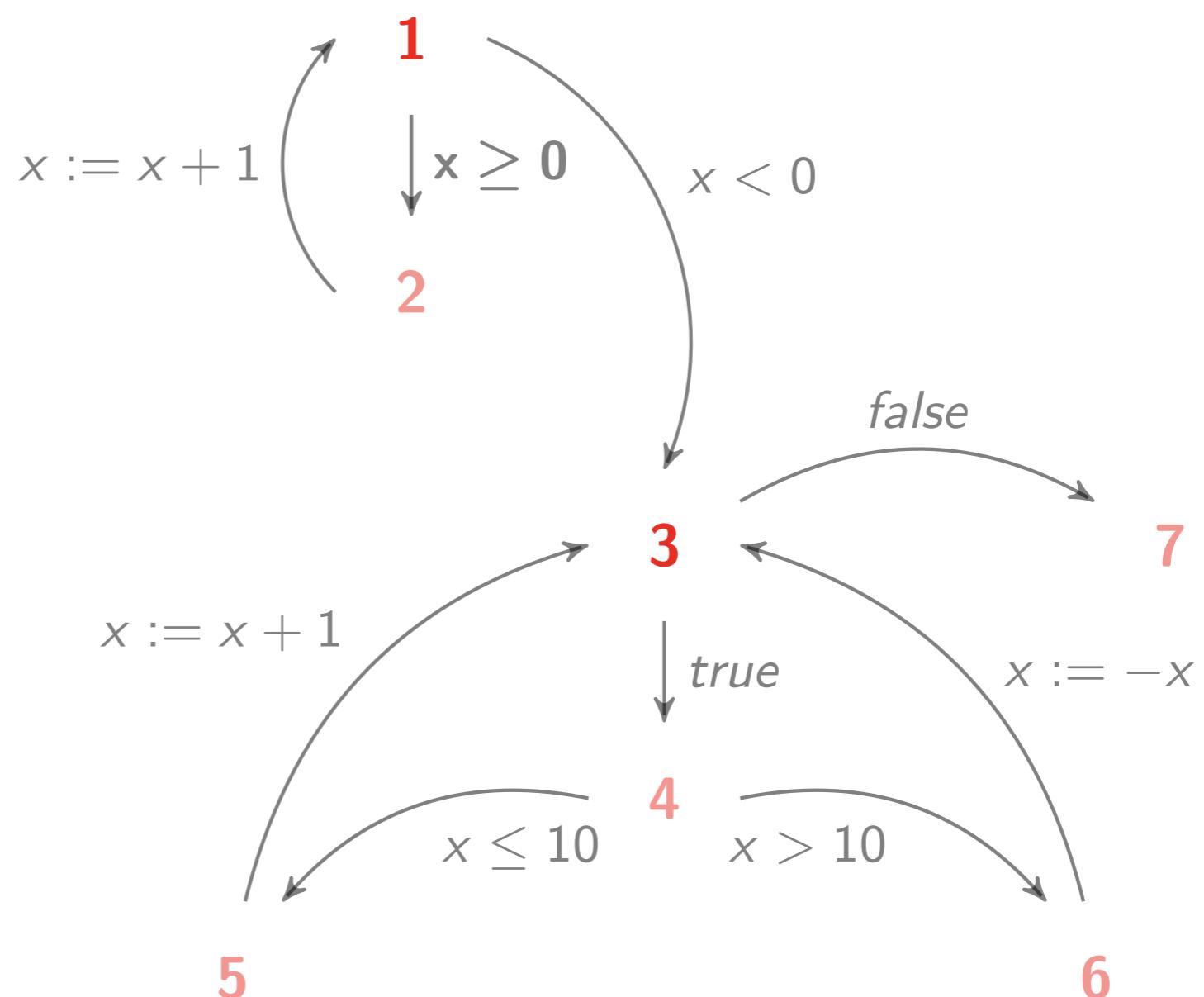


Example

```
int : x, y
while 1( $x \geq 0$ ) do
  2 $x := x + 1$ 
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 $x := x + 1$ 
  else
    6 $x := -x$ 
od7
```

Property

$$\diamondsuit x = 3$$



Example

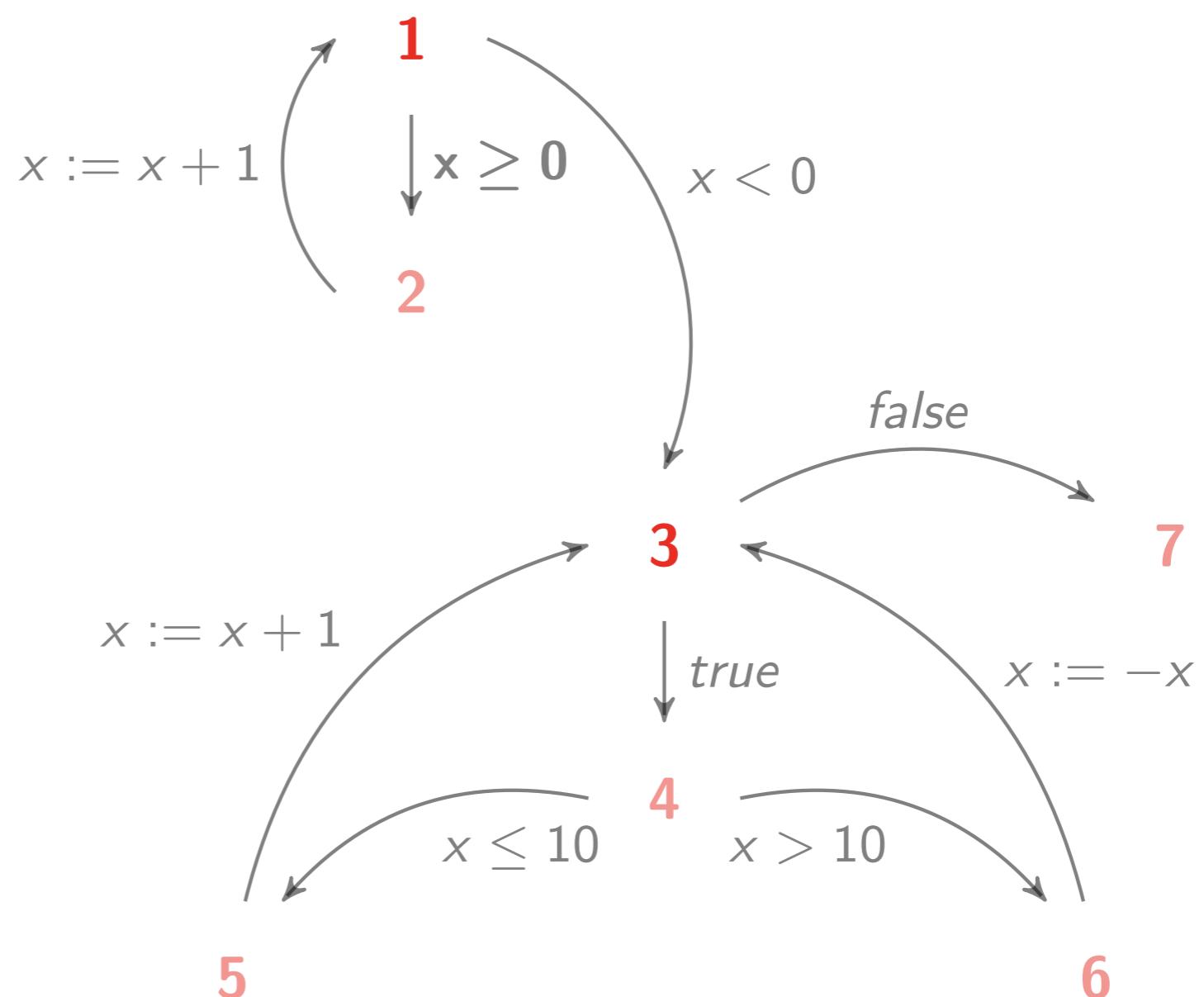
```

int : x, y
while 1( $x \geq 0$ ) do
  2 $x := x + 1$ 
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 $x := x + 1$ 
  else
    6 $x := -x$ 
od7

```

Property

$$\diamondsuit x = 3$$



Example

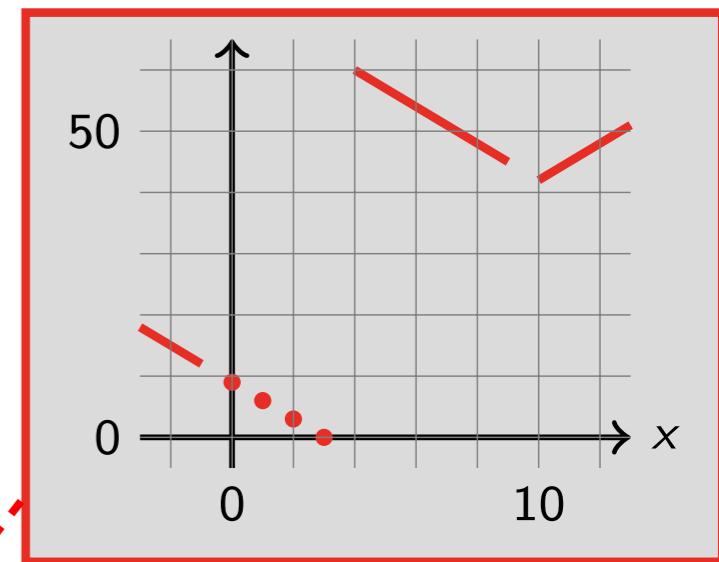
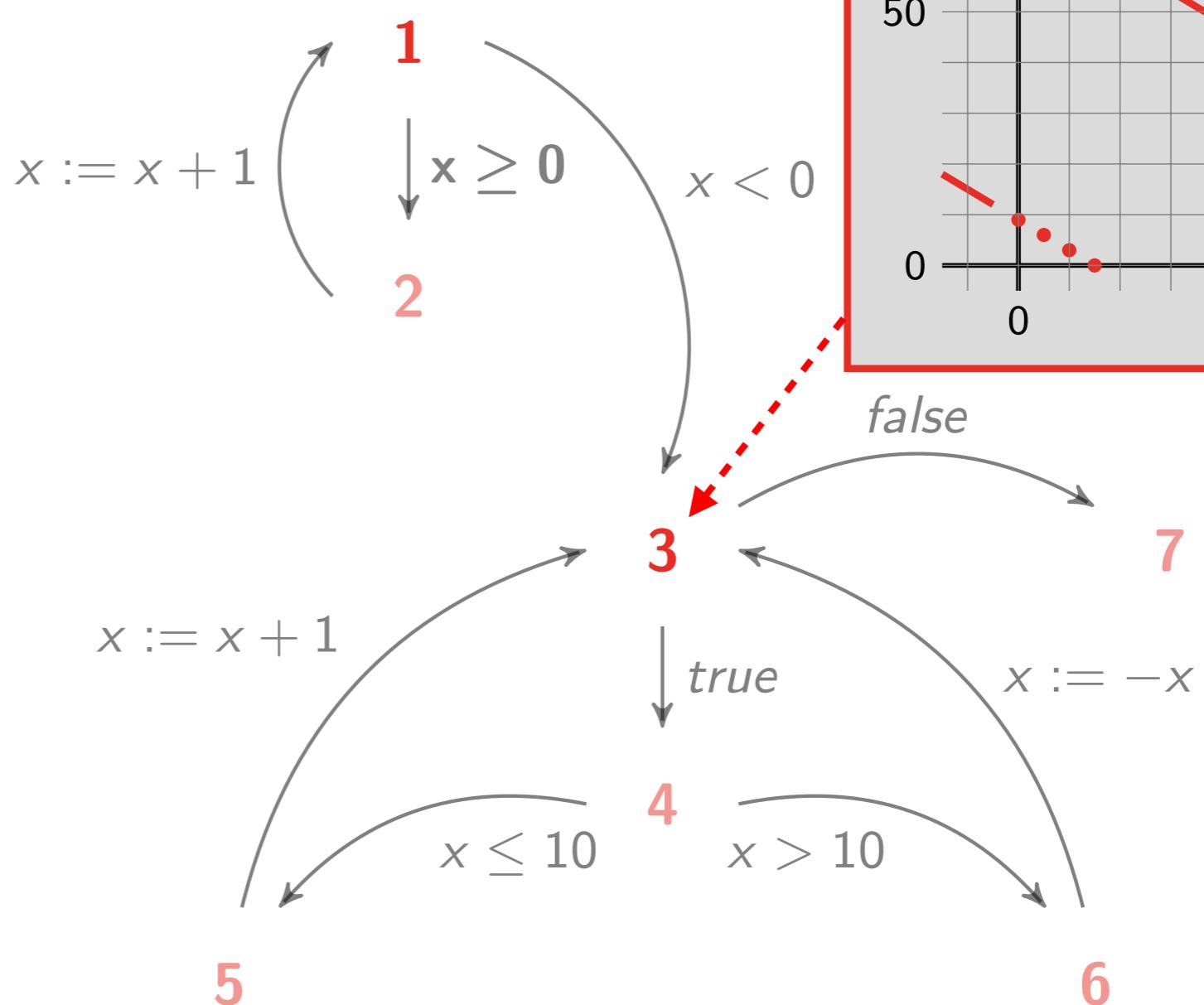
```

int : x, y
while 1( $x \geq 0$ ) do
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
od7

```

Property

$$\diamondsuit x = 3$$



Example

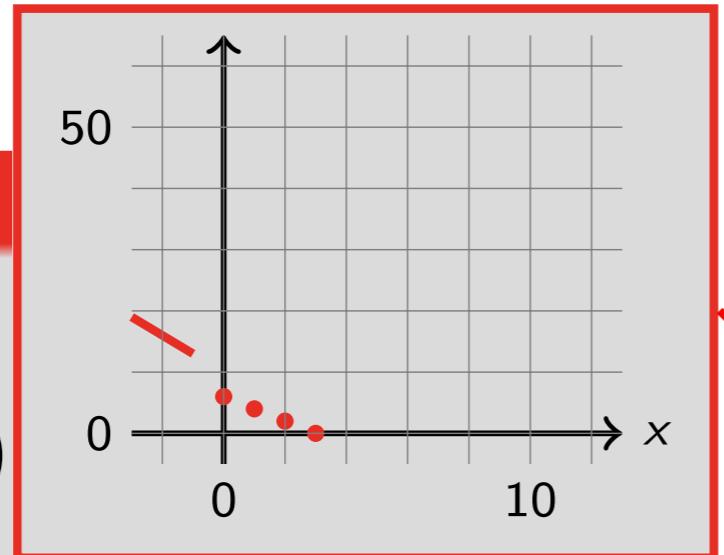
```

int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7

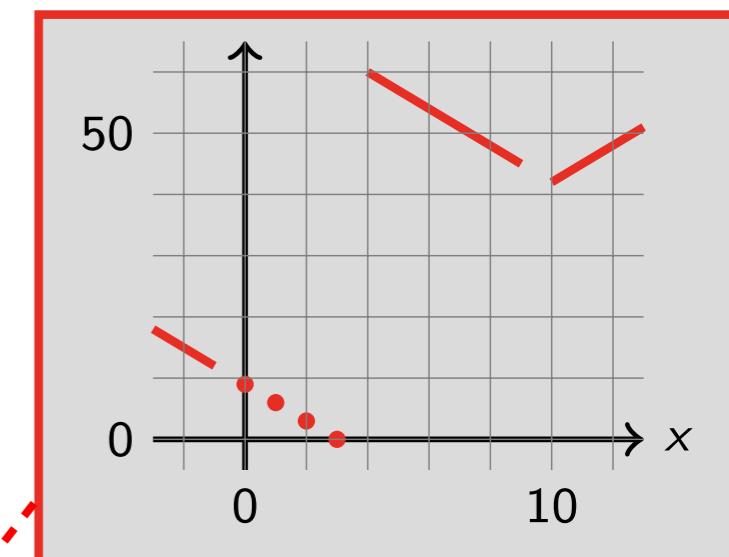
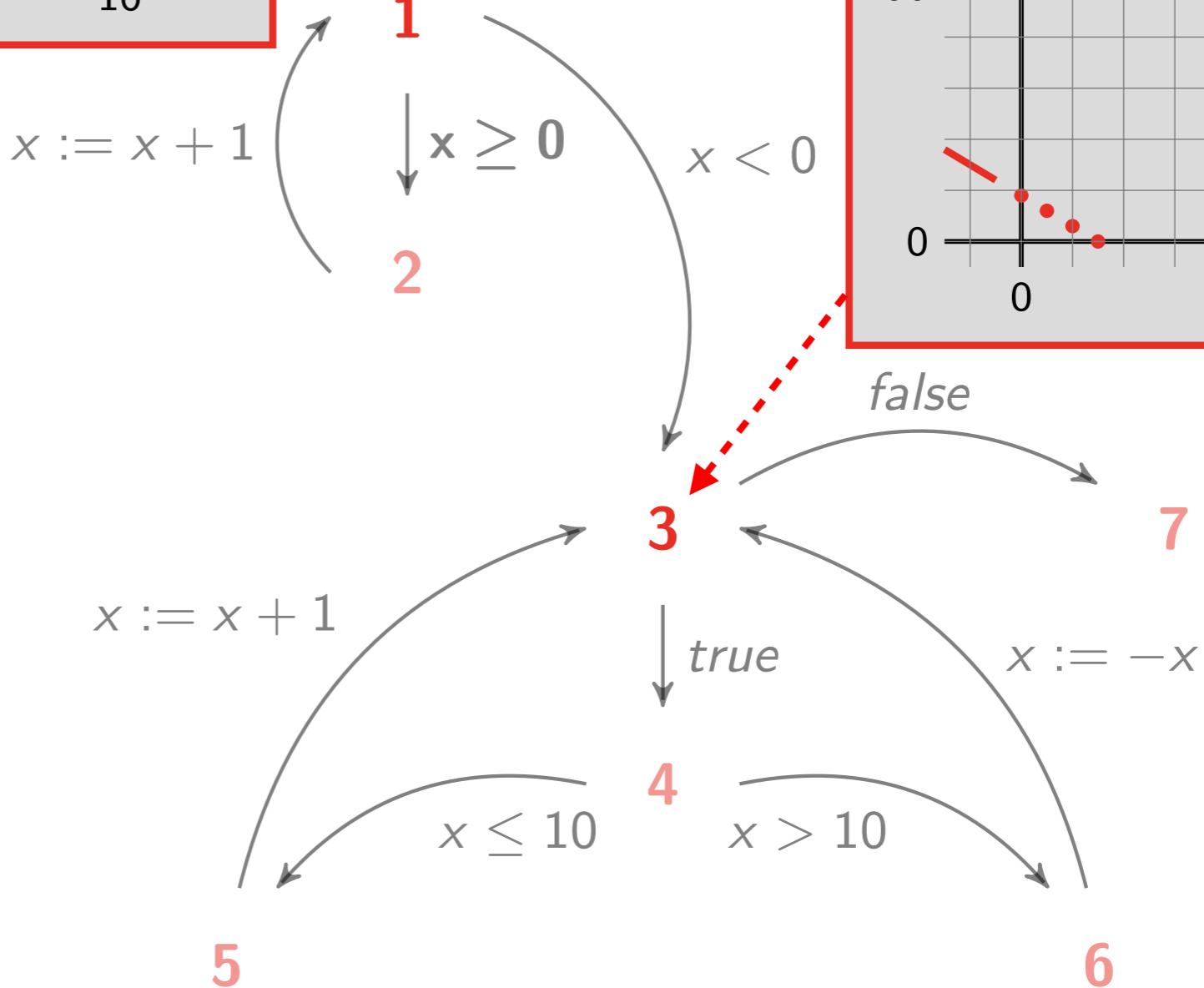
```

Property

$$\diamondsuit x = 3$$



the analysis gives $x \leq 3$ as
sufficient precondition

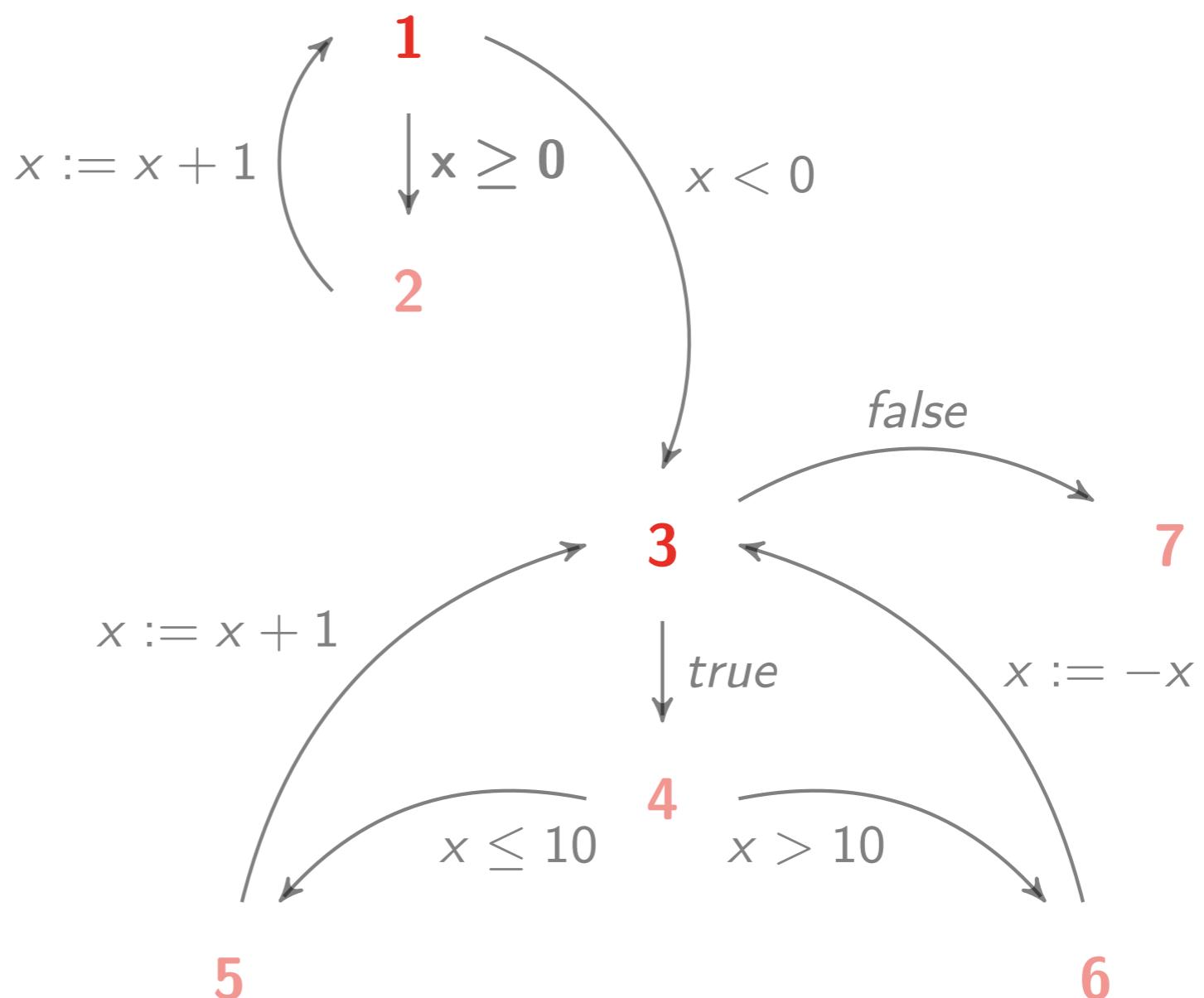


Example

```
int : x, y
while 1( $x \geq 0$ ) do
  2 $x := x + 1$ 
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 $x := x + 1$ 
  else
    6 $x := -x$ 
od7
```

Property

$$\square \diamond x = 3$$



Example

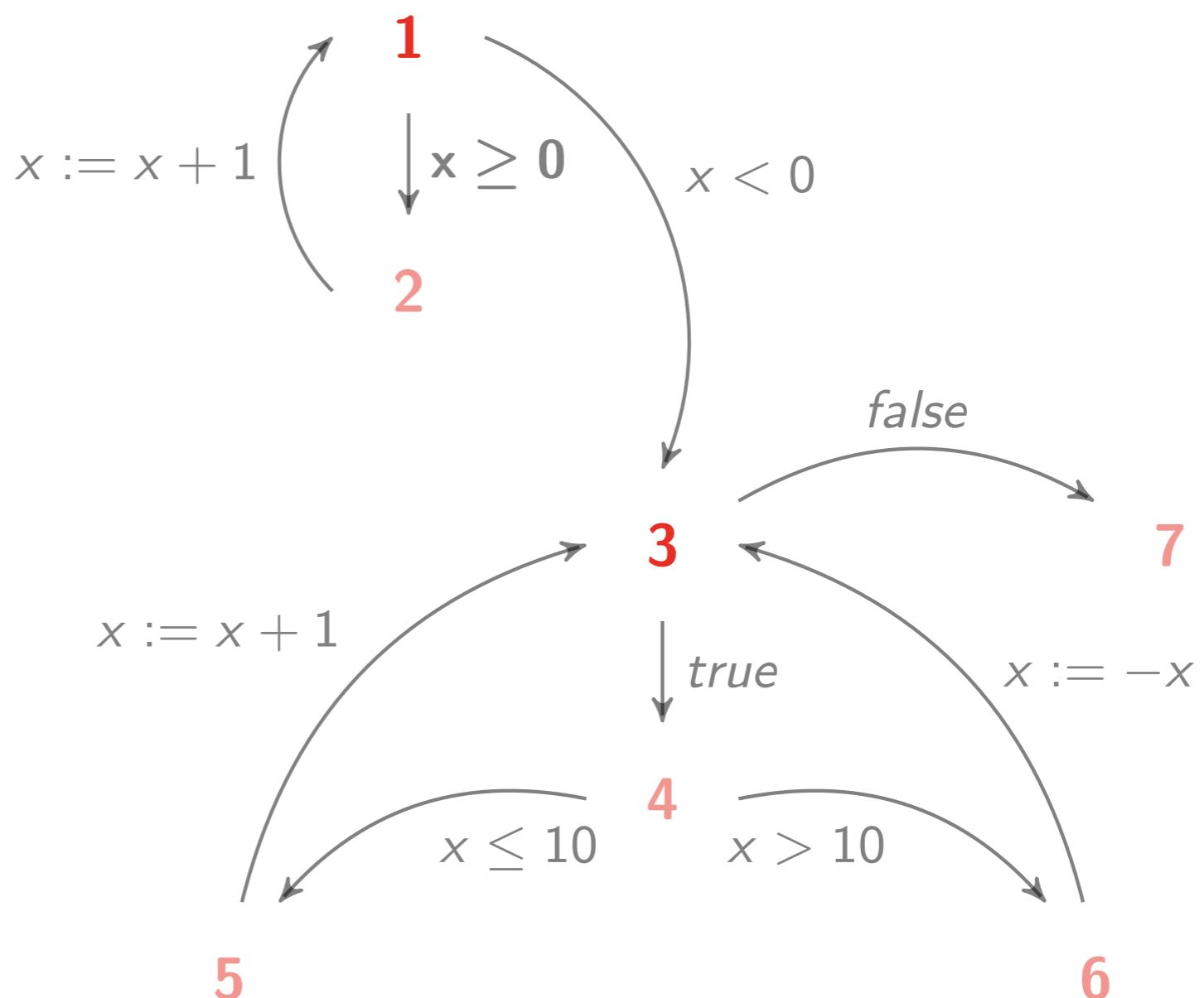
```

int : x, y
while 1( $x \geq 0$ ) do
  2 $x := x + 1$ 
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 $x := x + 1$ 
  else
    6 $x := -x$ 
od7

```

Property

$$\square \diamond x = 3$$



Example

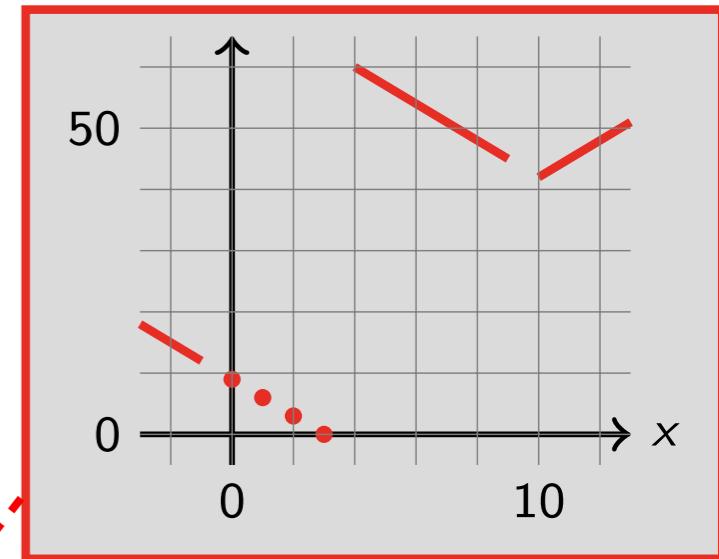
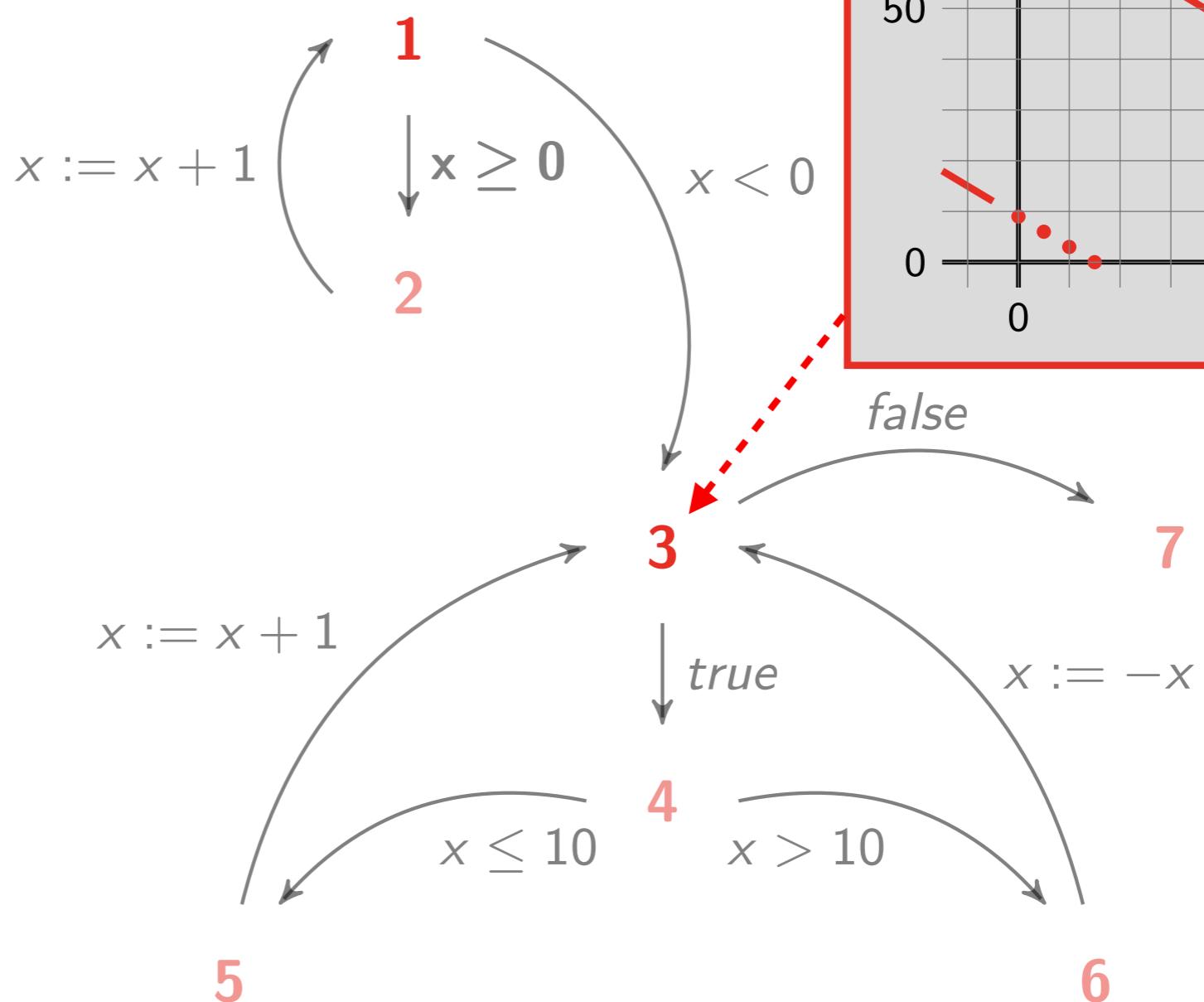
```

int : x, y
while 1( $x \geq 0$ ) do
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
od7

```

Property

$$\square \diamond x = 3$$



Example

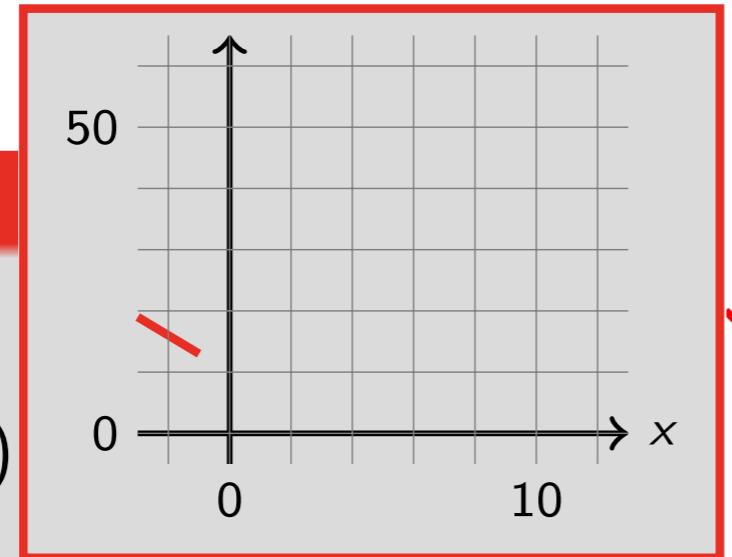
```

int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
od7

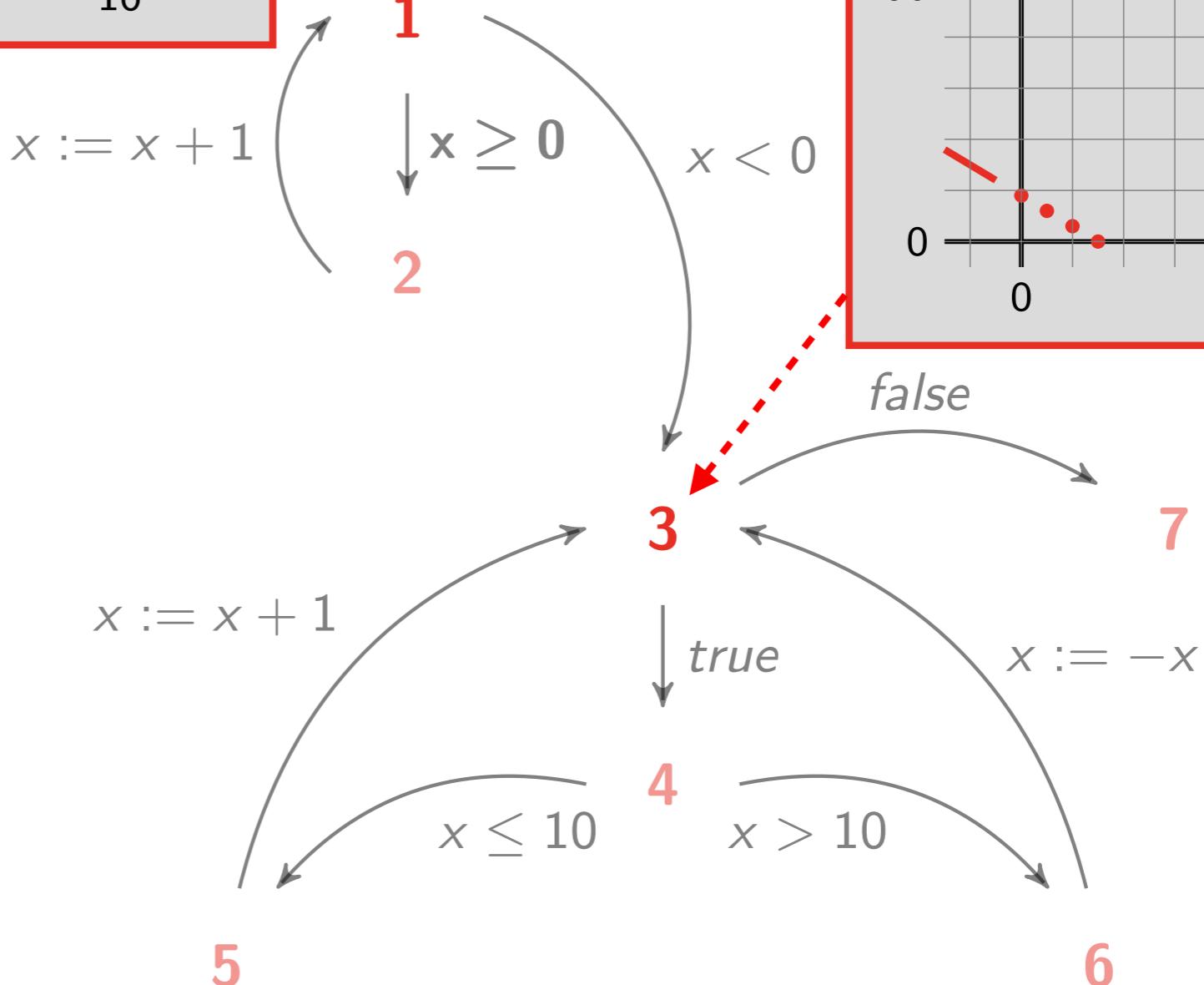
```

Property

$$\square \diamond x = 3$$



the analysis gives $x < 0$ as
sufficient precondition



5

Guarantee Properties

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

$$\mathcal{T}_t \in \Sigma \rightarrow \emptyset$$

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

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

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

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

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

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

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

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

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

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

$$F_g^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \varphi \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \wedge s \notin \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

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

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

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

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

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

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

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

$$F_g^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \varphi \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \wedge s \notin \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

$s \in \varphi$

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

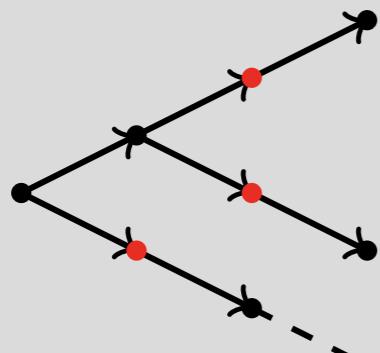
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

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

$$F_g^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \varphi \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \wedge s \notin \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

Example



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

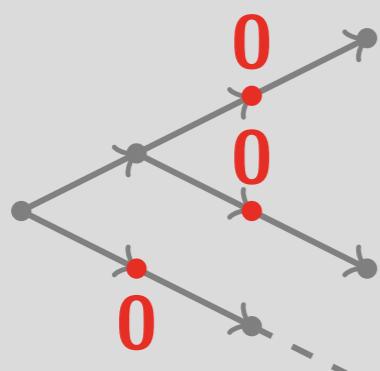
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

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

$$F_g^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \varphi \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \wedge s \notin \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

Example



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

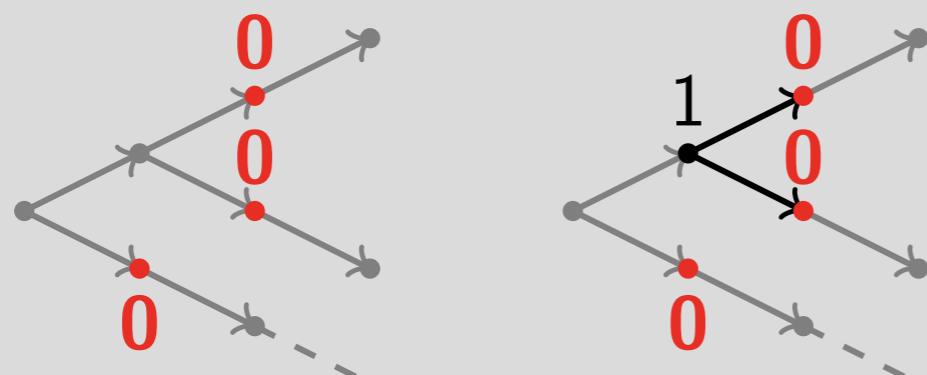
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

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

$$F_g^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \varphi \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \wedge s \notin \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

Example



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

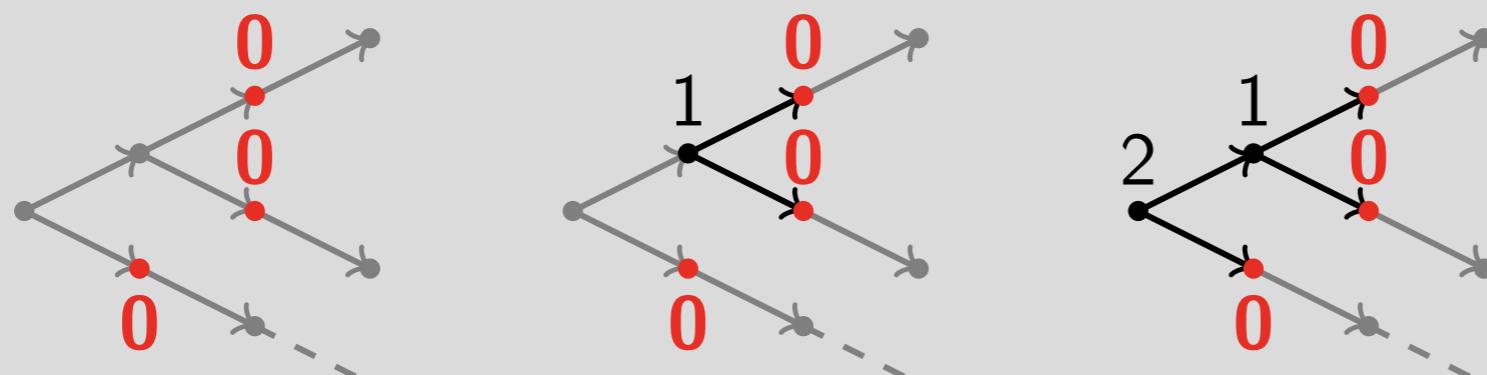
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

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

$$F_g^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \varphi \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \wedge s \notin \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

Example



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

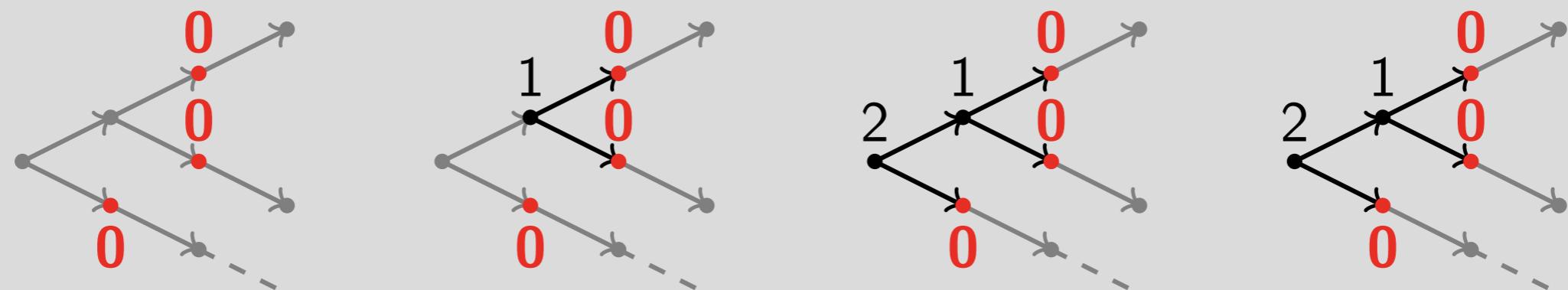
$$\mathcal{T}_g^\varphi \in \Sigma \rightarrow \mathbb{O}$$

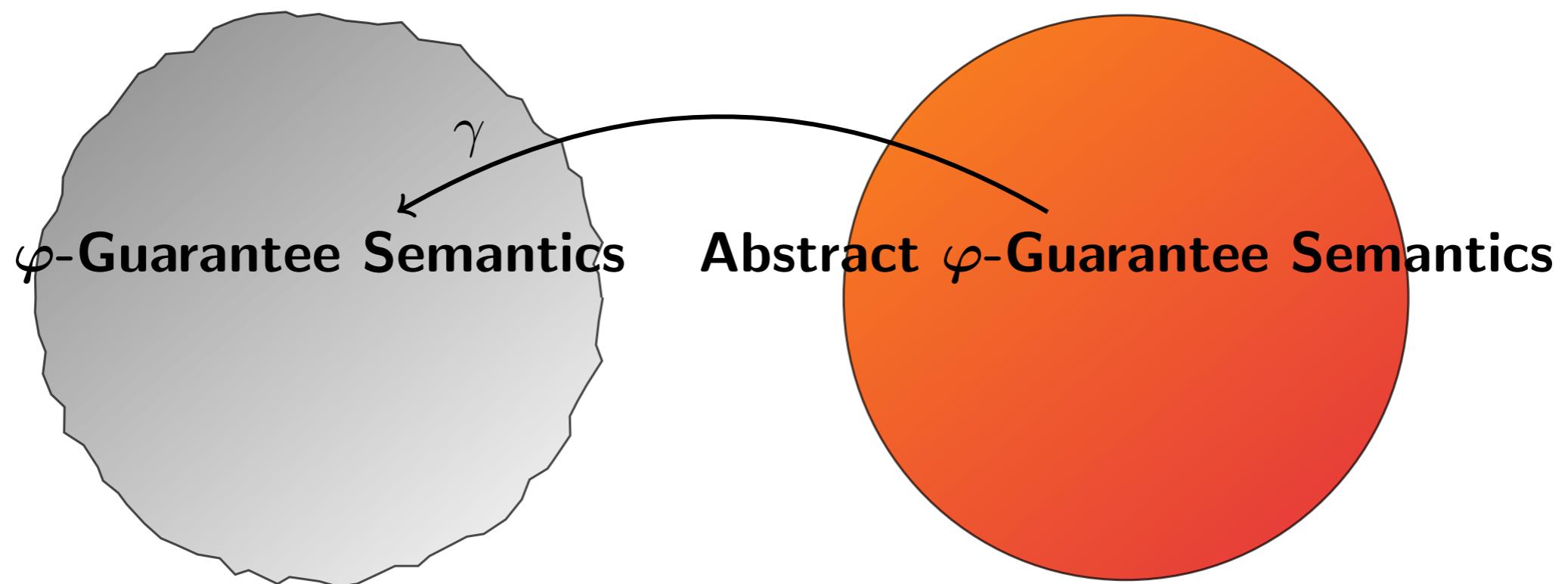
$$\mathcal{T}_g^\varphi \stackrel{\text{def}}{=} \text{Ifp } F_g^\varphi$$

$$F_g^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \varphi \\ \sup\{ v(s') + 1 \mid s \rightarrow s' \} & s \in \widetilde{\text{pre}}(\text{dom}(v)) \wedge s \notin \varphi \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

Example



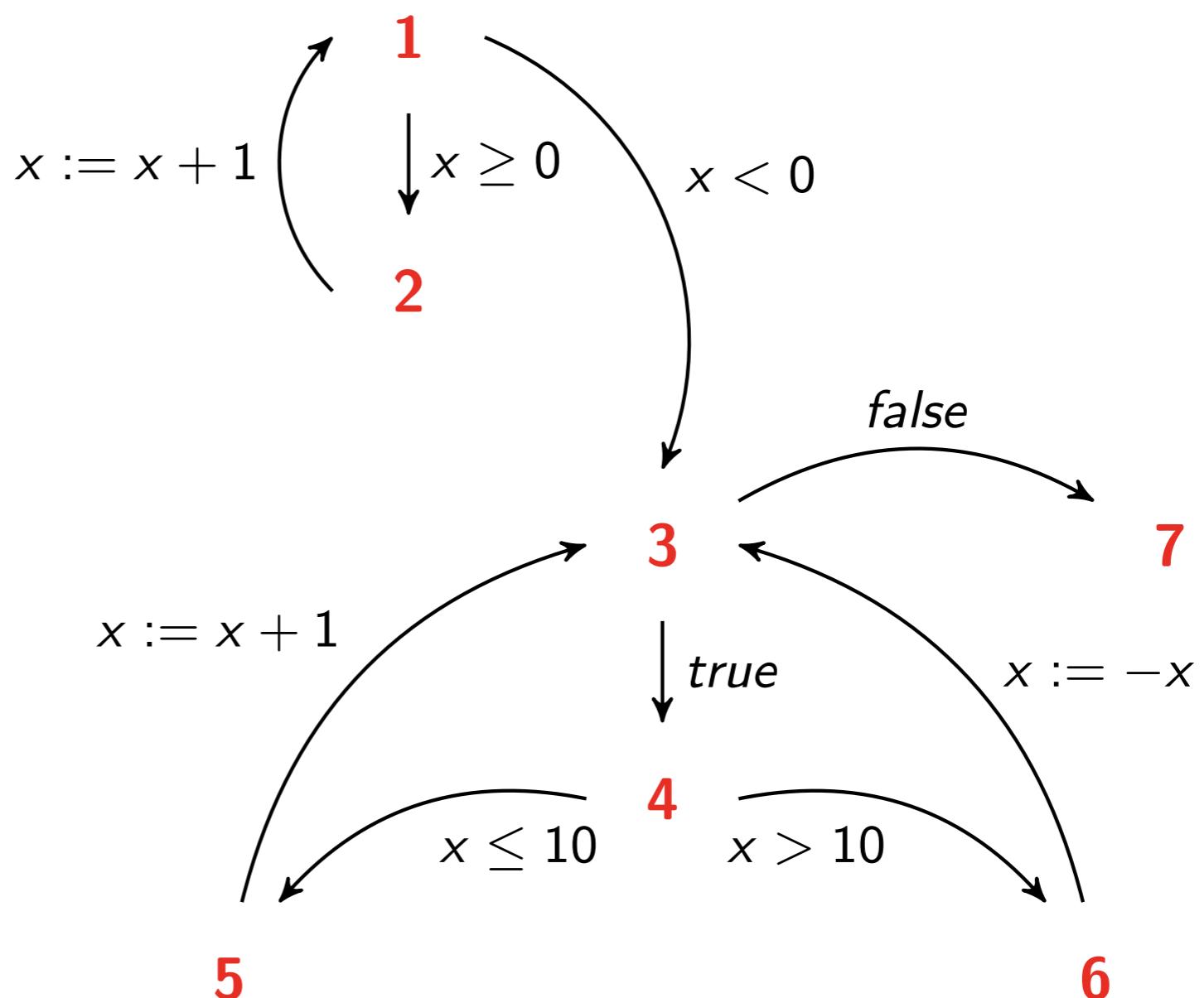


Example

```
int : x, y
while 1( $x \geq 0$ ) do
  2  $x := x + 1$ 
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5  $x := x + 1$ 
  else
    6  $x := -x$ 
od7
```

Property

$$\diamondsuit x = 3$$

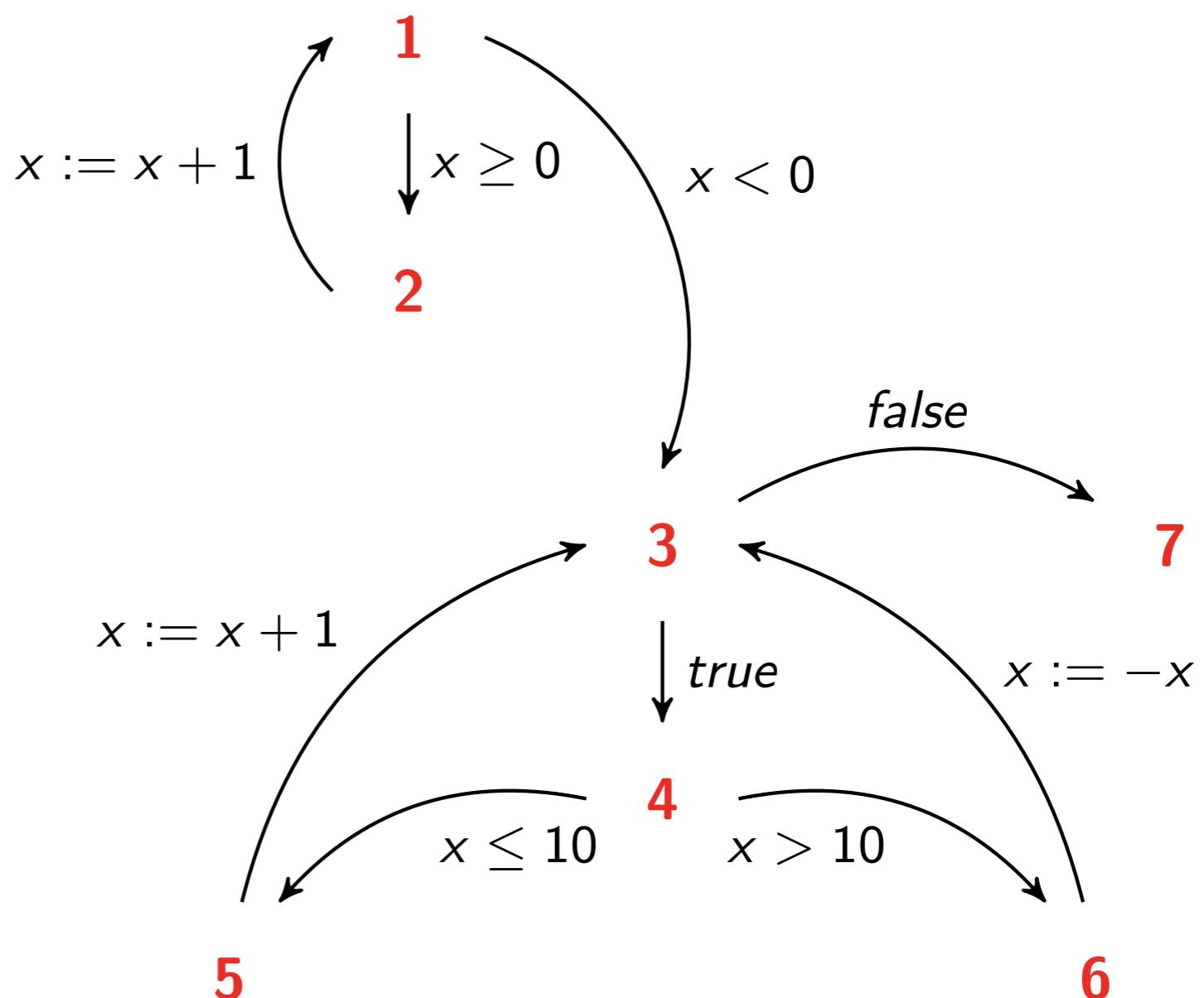


Example

```
int : x, y
while 1( $x \geq 0$ ) do
  2  $x := x + 1$ 
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5  $x := x + 1$ 
  else
    6  $x := -x$ 
od7
```

Property

$$\diamondsuit x = 3$$



Example

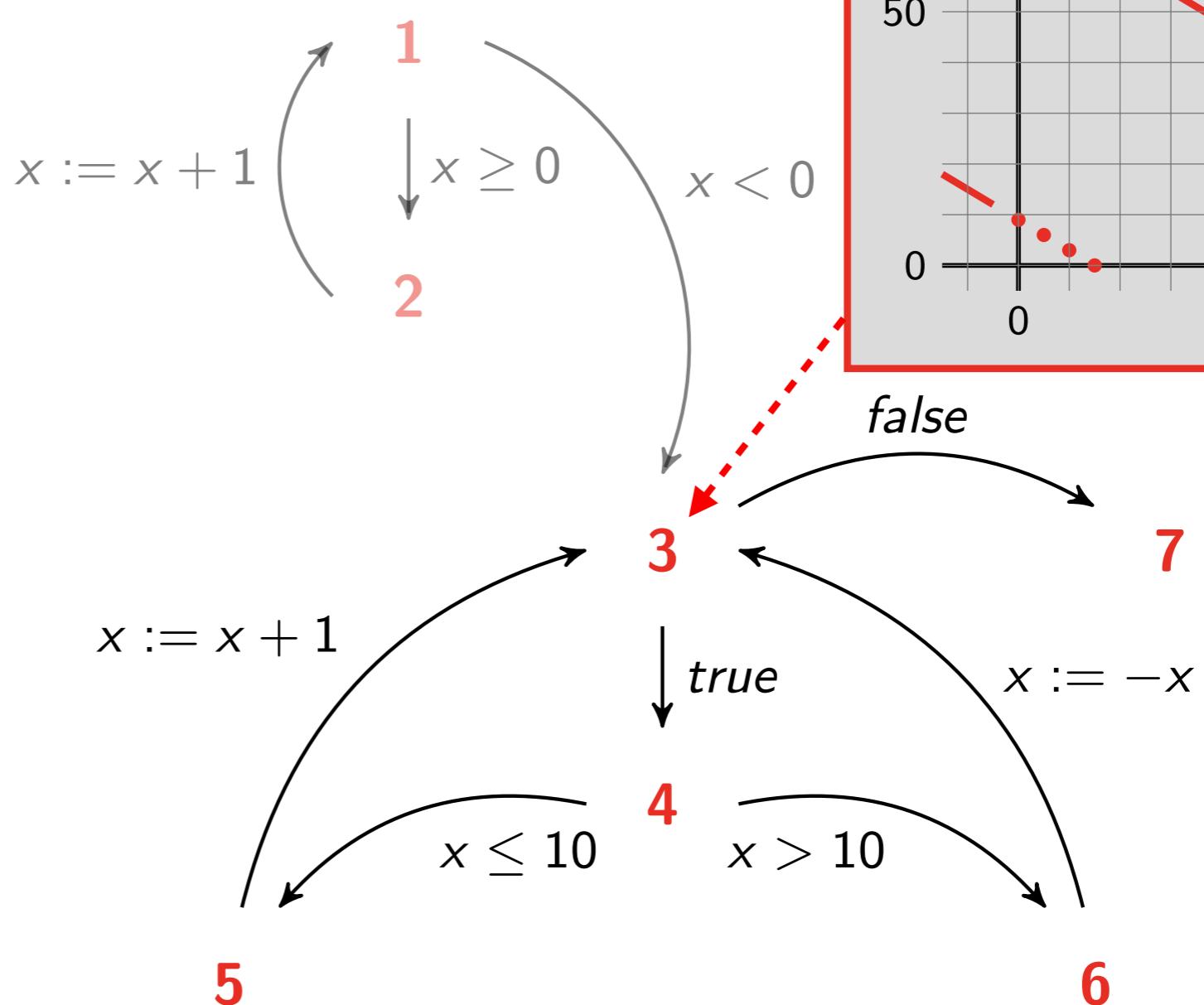
```

int : x, y
while 1( $x \geq 0$ ) do
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
od7

```

Property

$$\diamondsuit x = 3$$

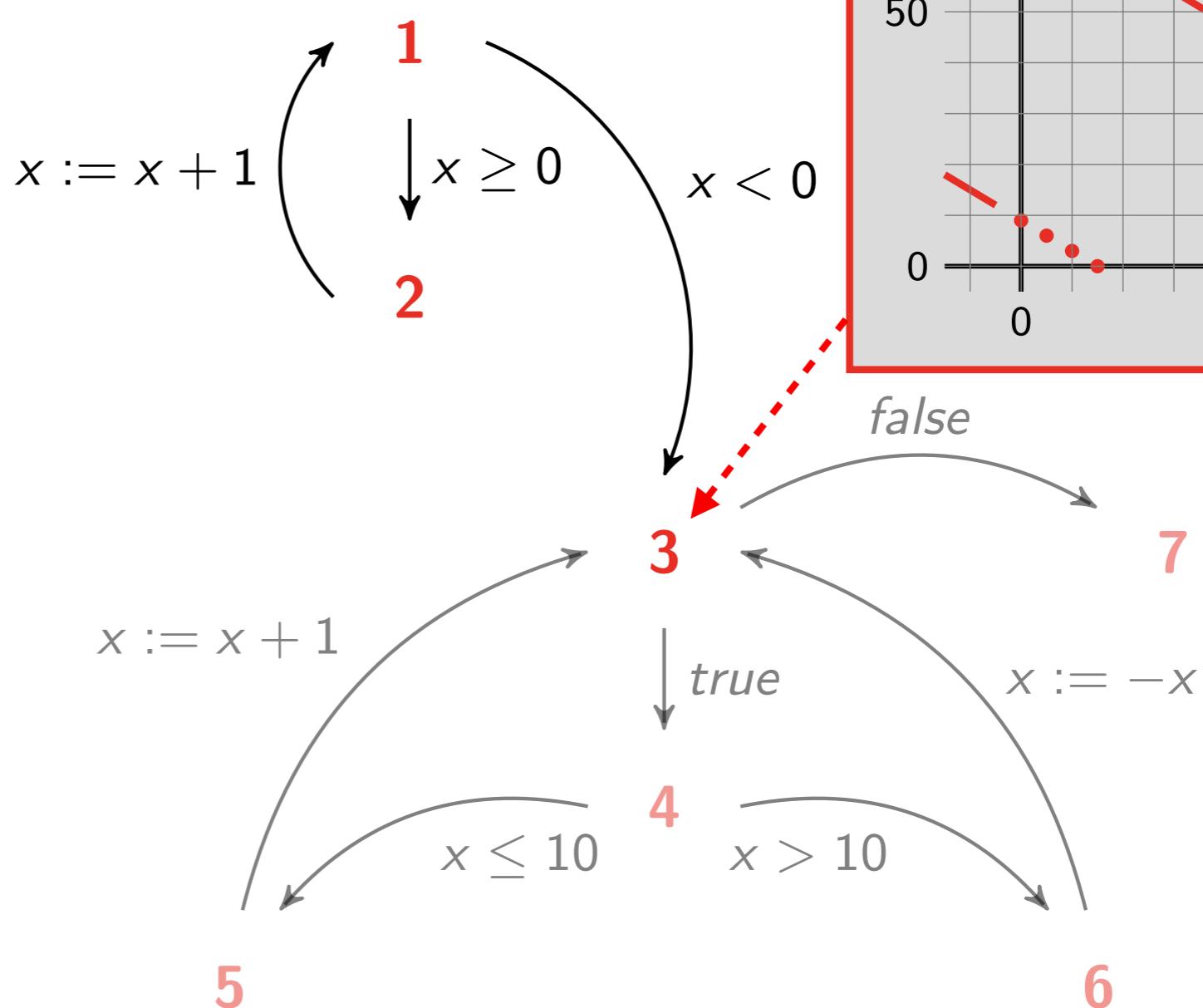


Example

```
int : x, y
while 1( $x \geq 0$ ) do
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7
```

Property

$$\diamondsuit x = 3$$



Example

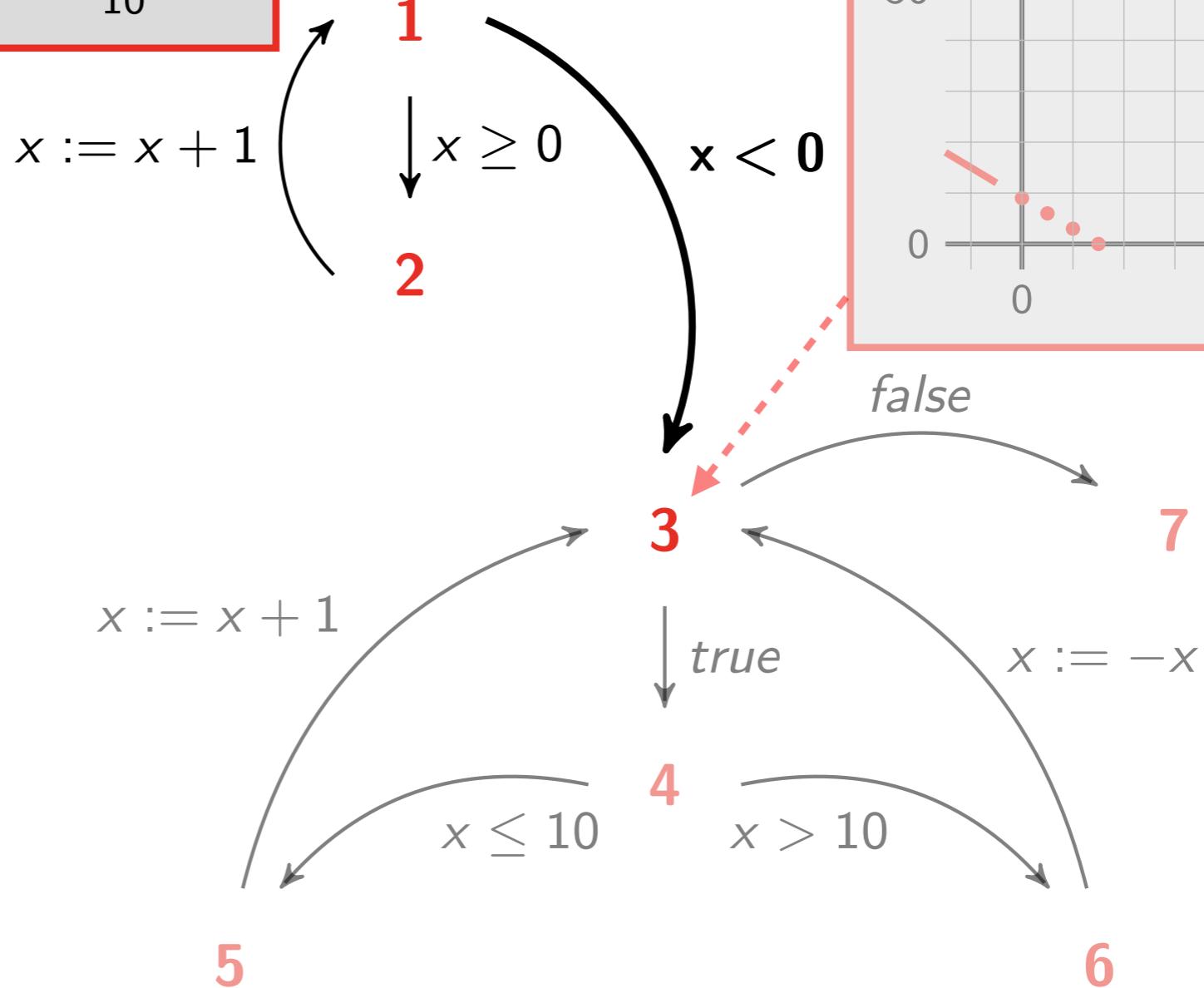
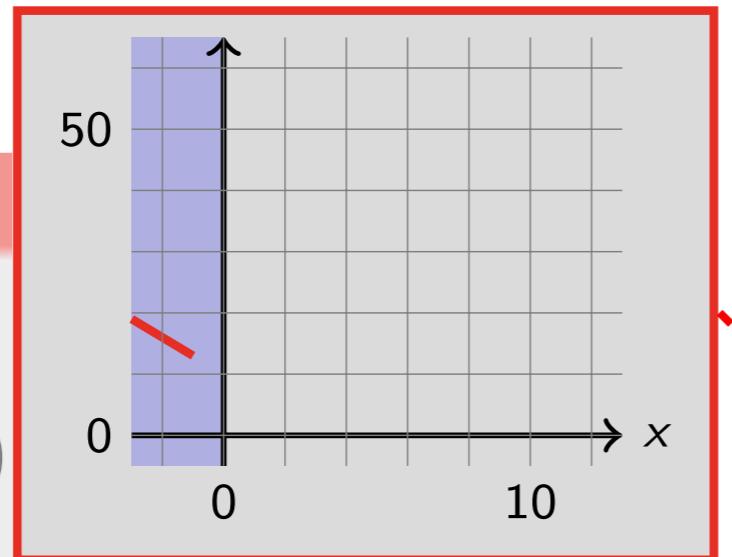
```

int : x, y
while 1(  $x \geq 0$  )
  2 x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 x := x + 1
  else
    6 x := -x
  od7

```

Property

$$\diamondsuit x = 3$$



5

Example

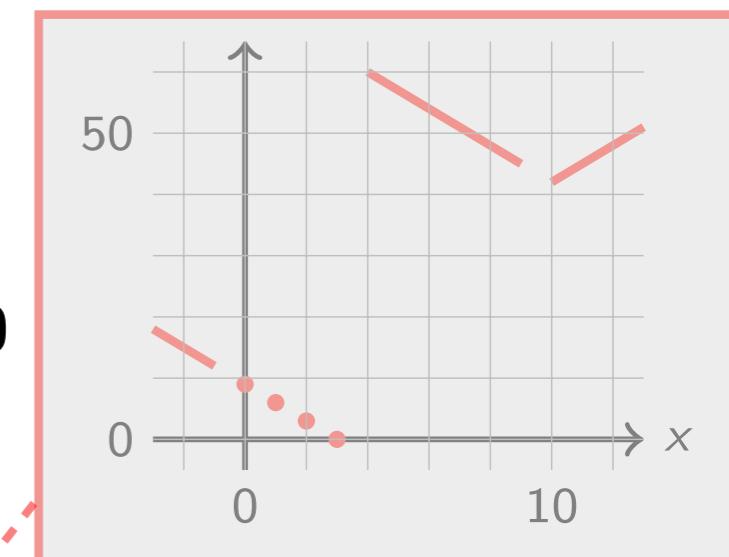
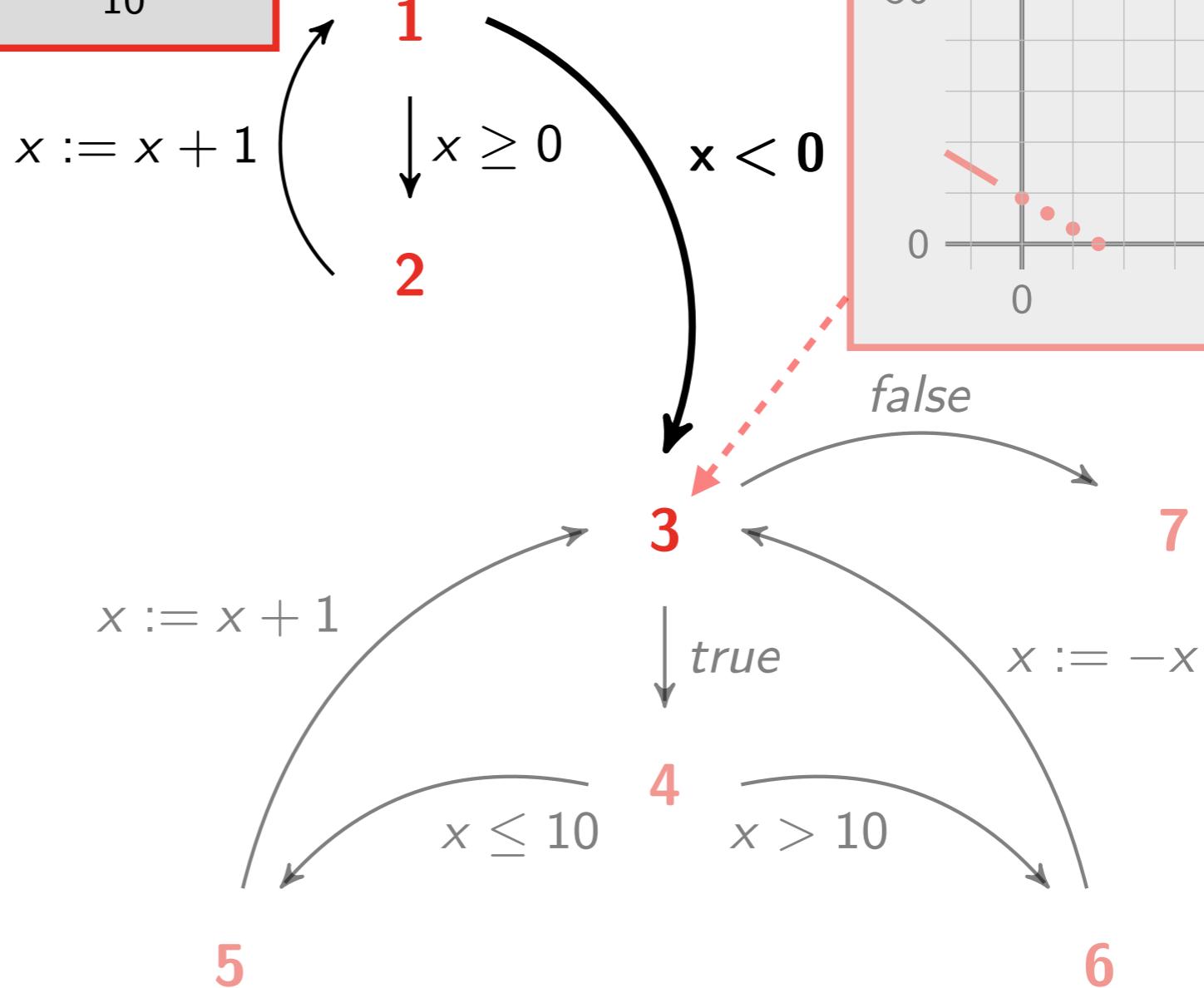
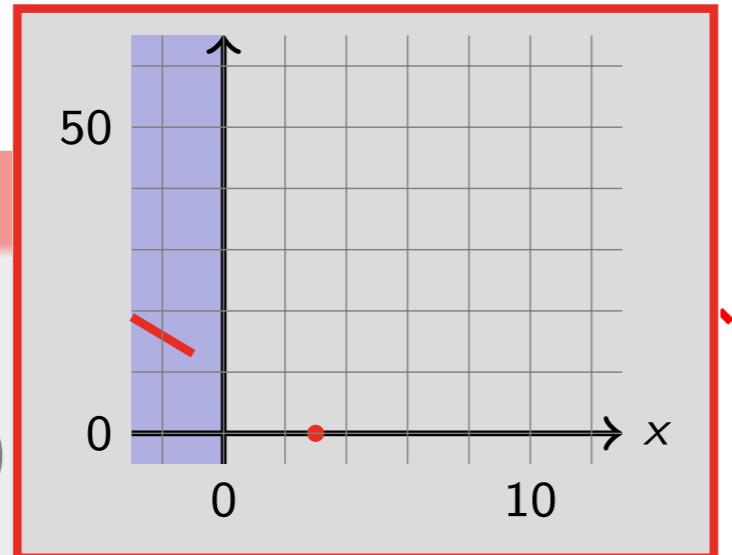
```

int : x, y
while 1(  $x \geq 0$  )
  2 x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 x := x + 1
  else
    6 x := -x
  od7

```

Property

$$\diamondsuit x = 3$$

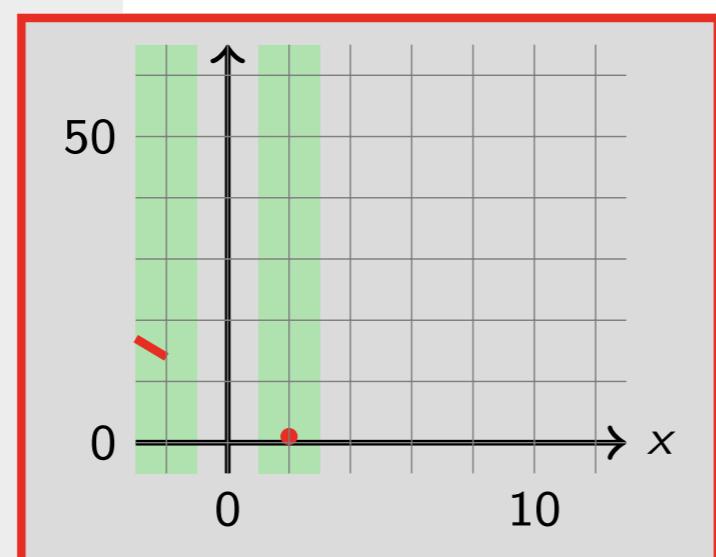
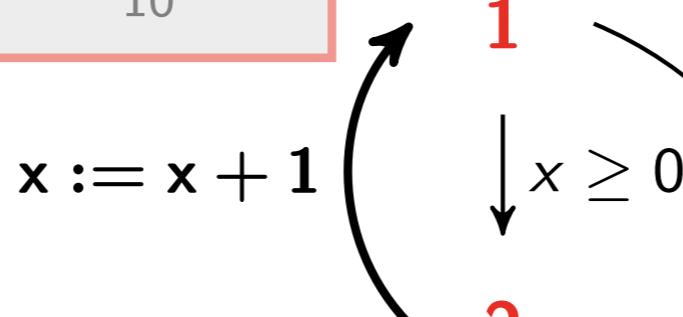
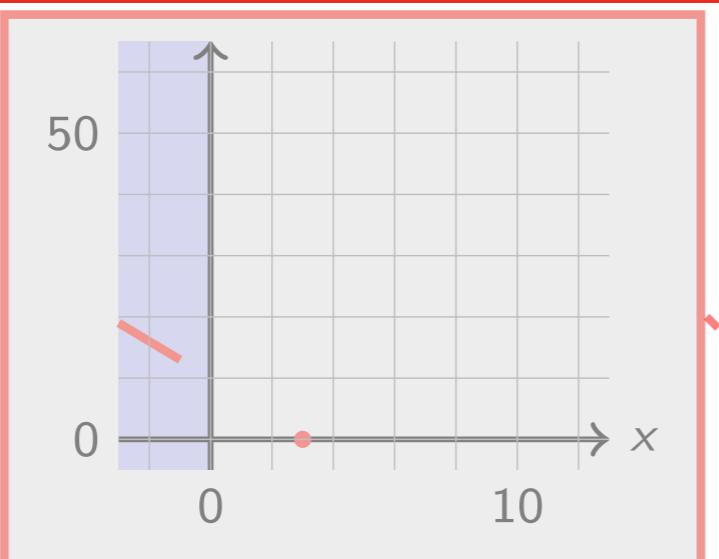


5

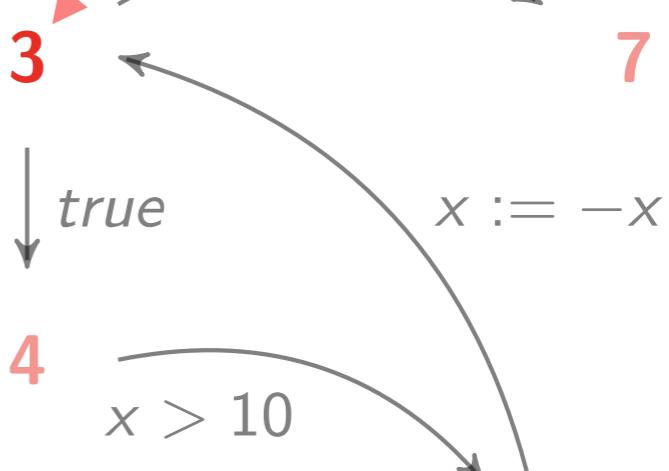
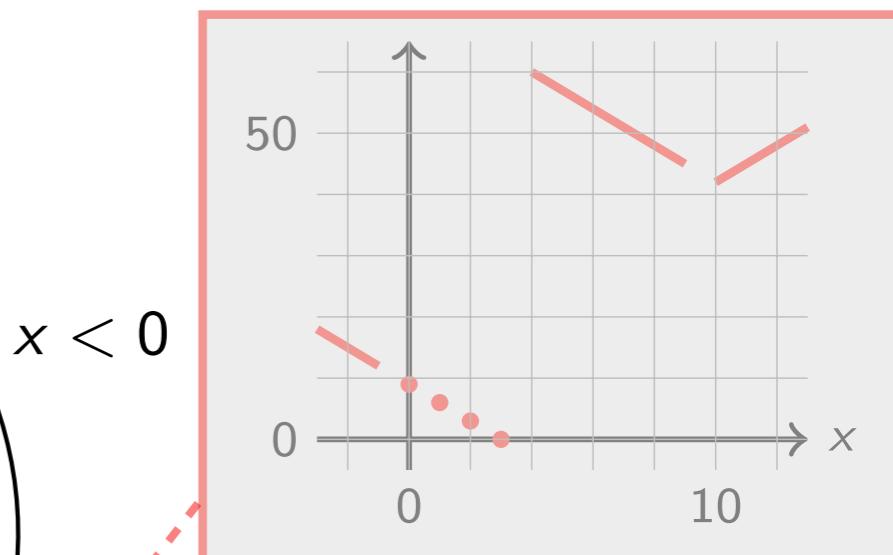
6

Example

```
int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7
```



5



4
6

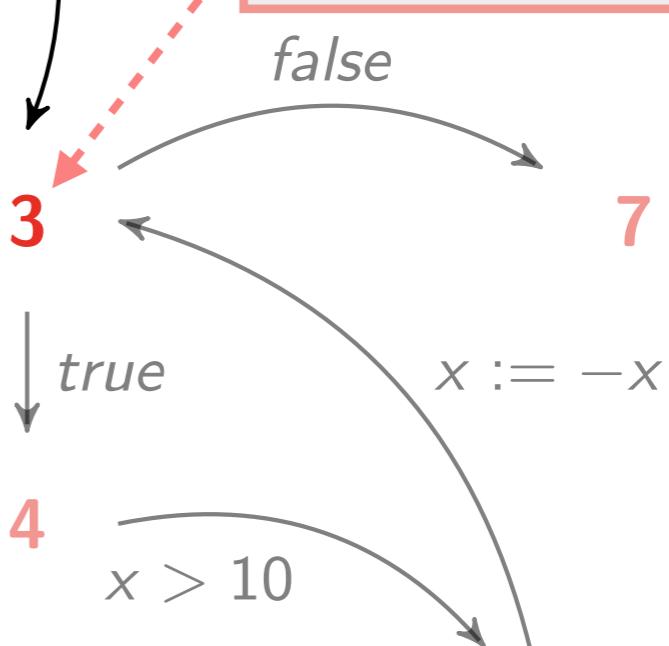
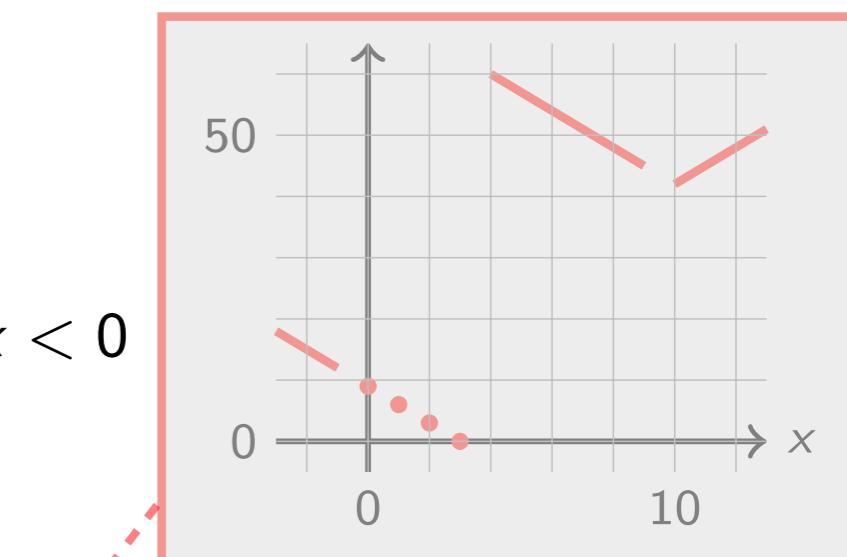
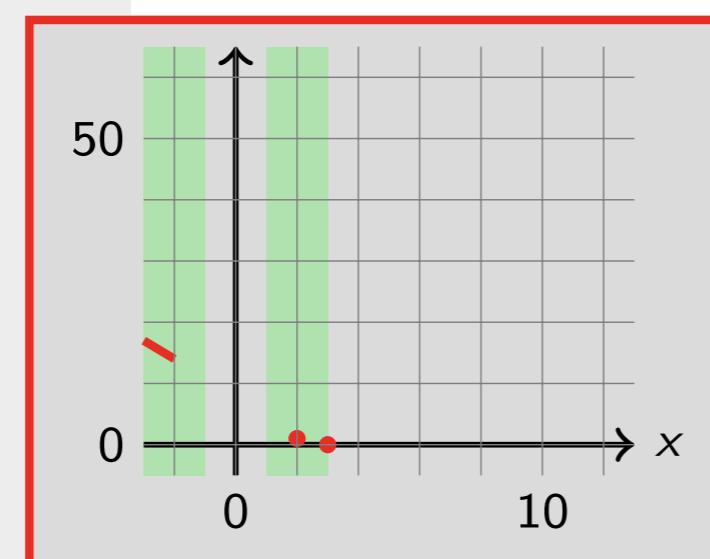
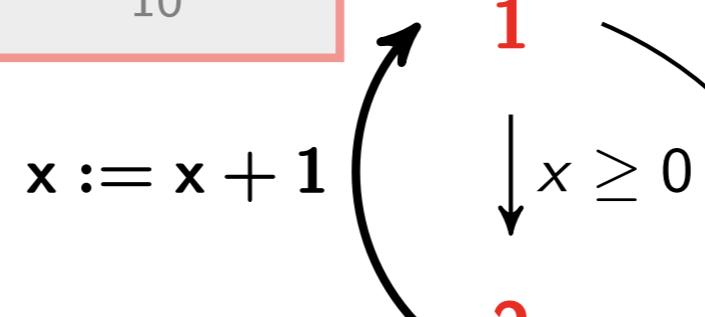
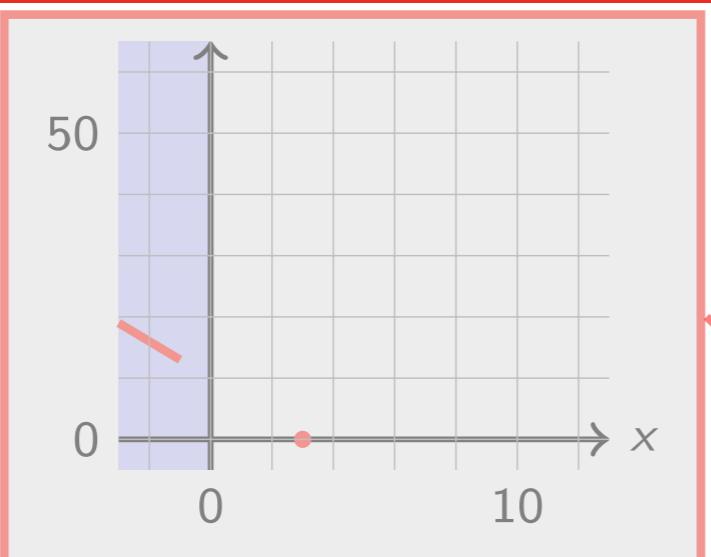
Property

$$\diamond x = 3$$

Example

```
int : x, y
while 1(x ≥ 0)
    2x := x + 1
od

while 3( true ) do
    if 4( x ≤ 10 )
        5x := x + 1
    else
        6x := -x
od 7
```



Property

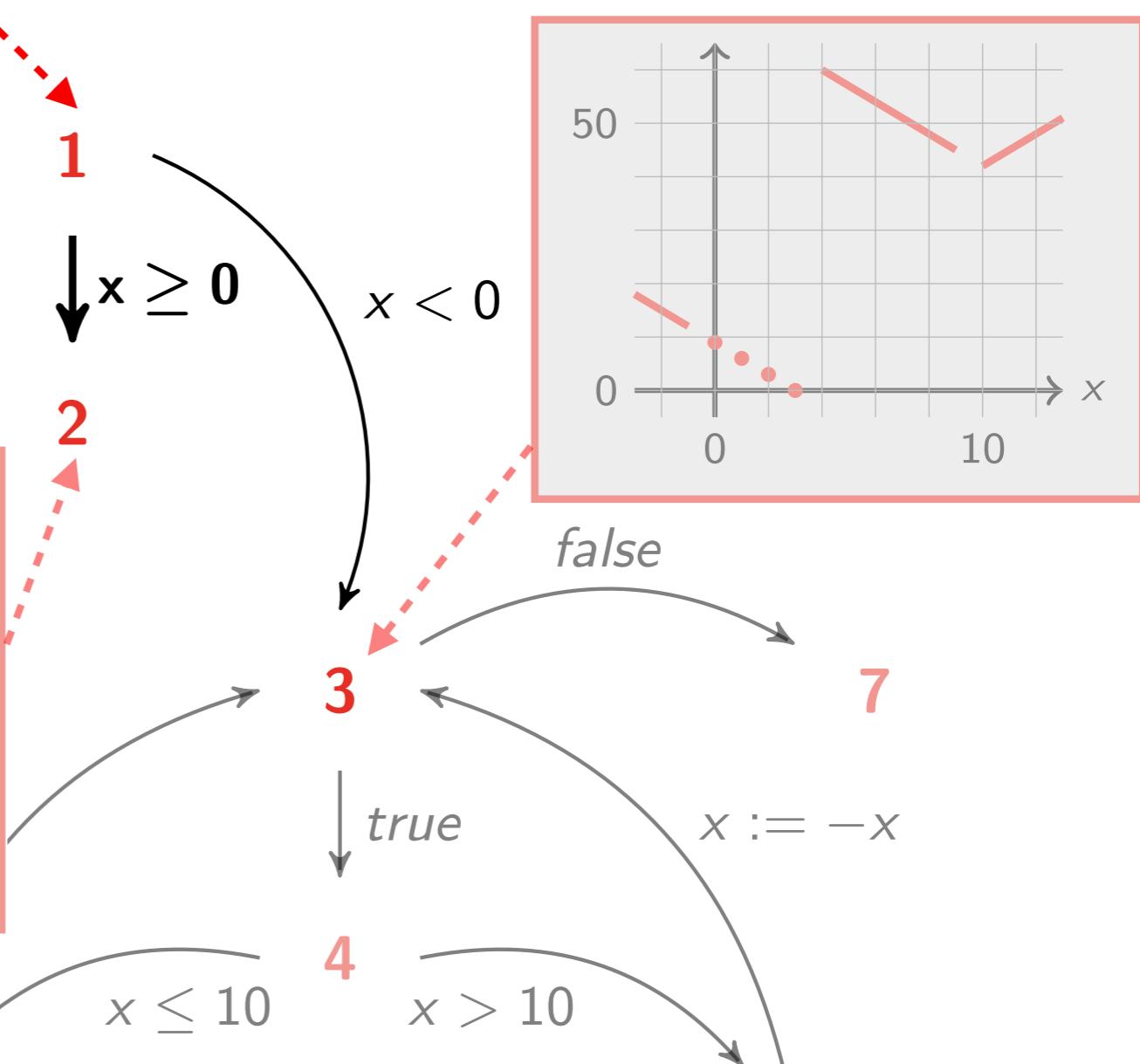
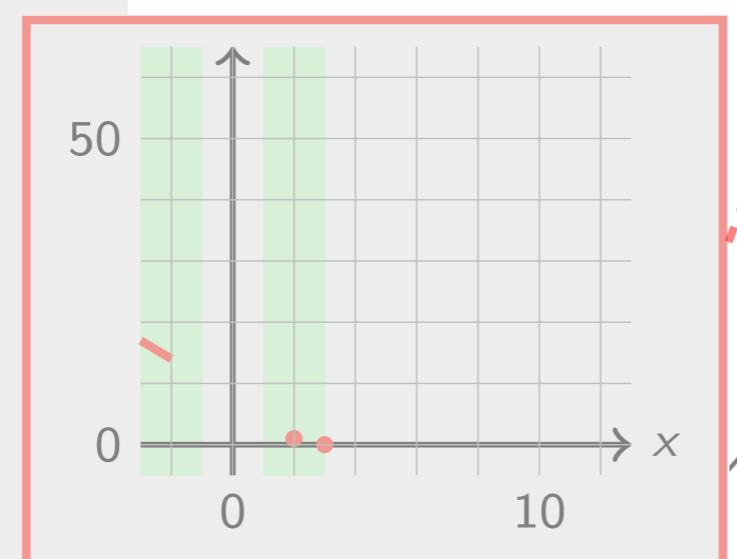
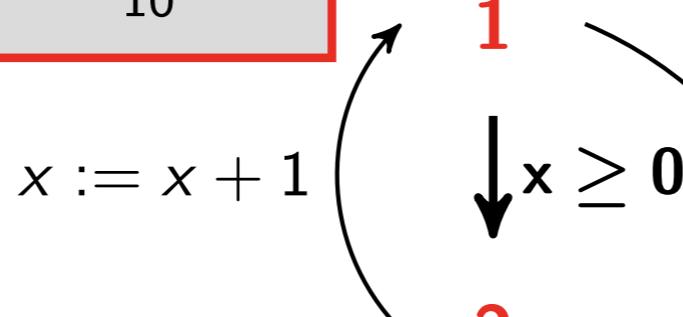
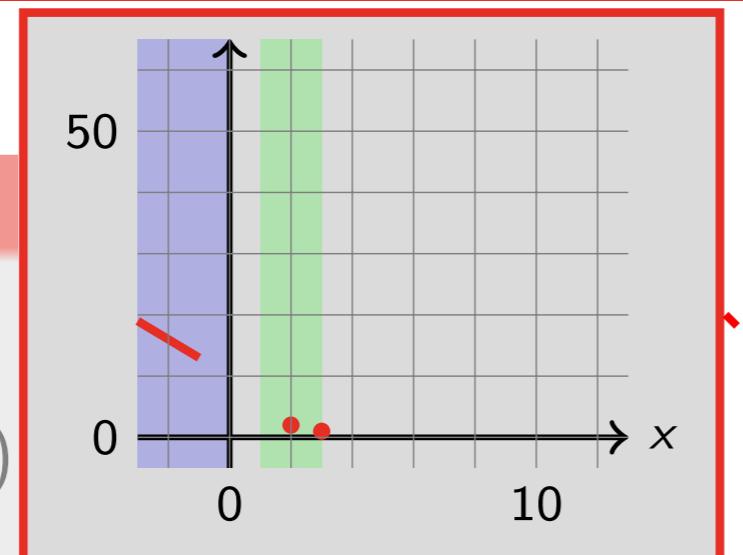
$$\diamond \quad x = 3$$

Example

```

int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7

```



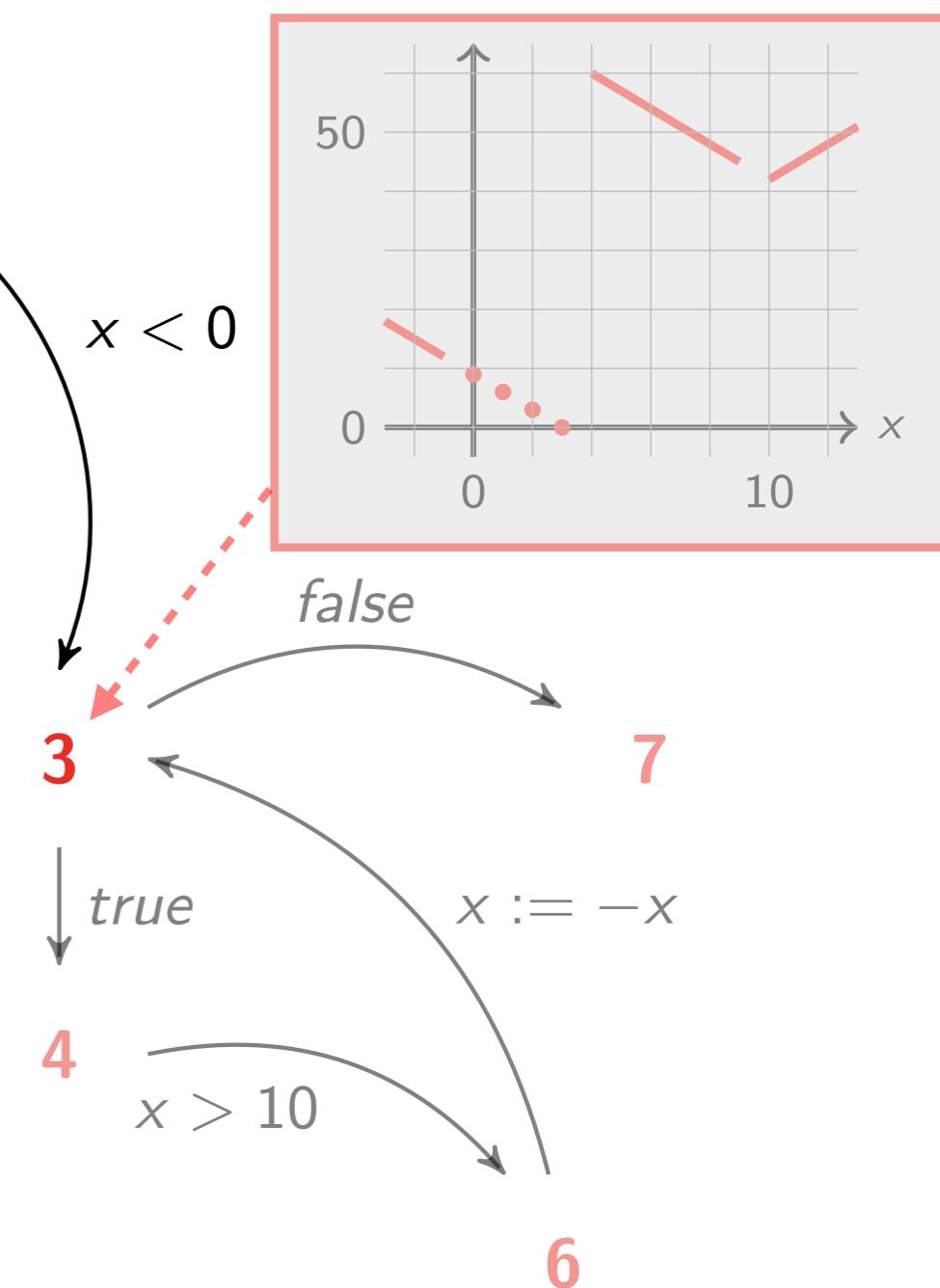
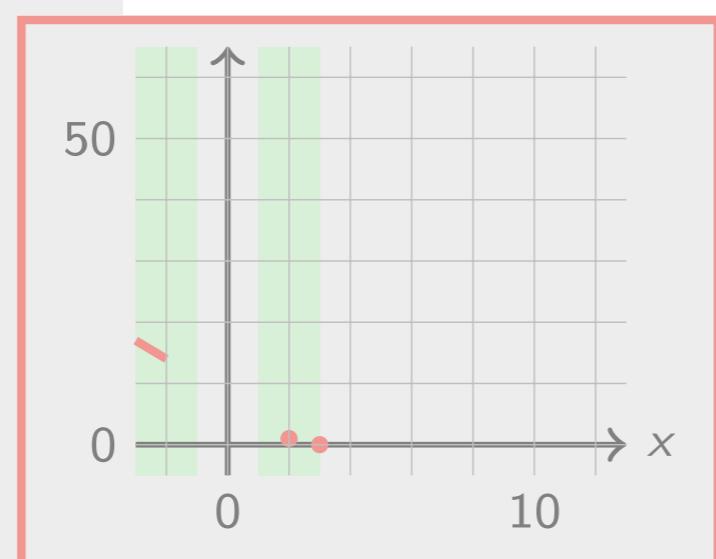
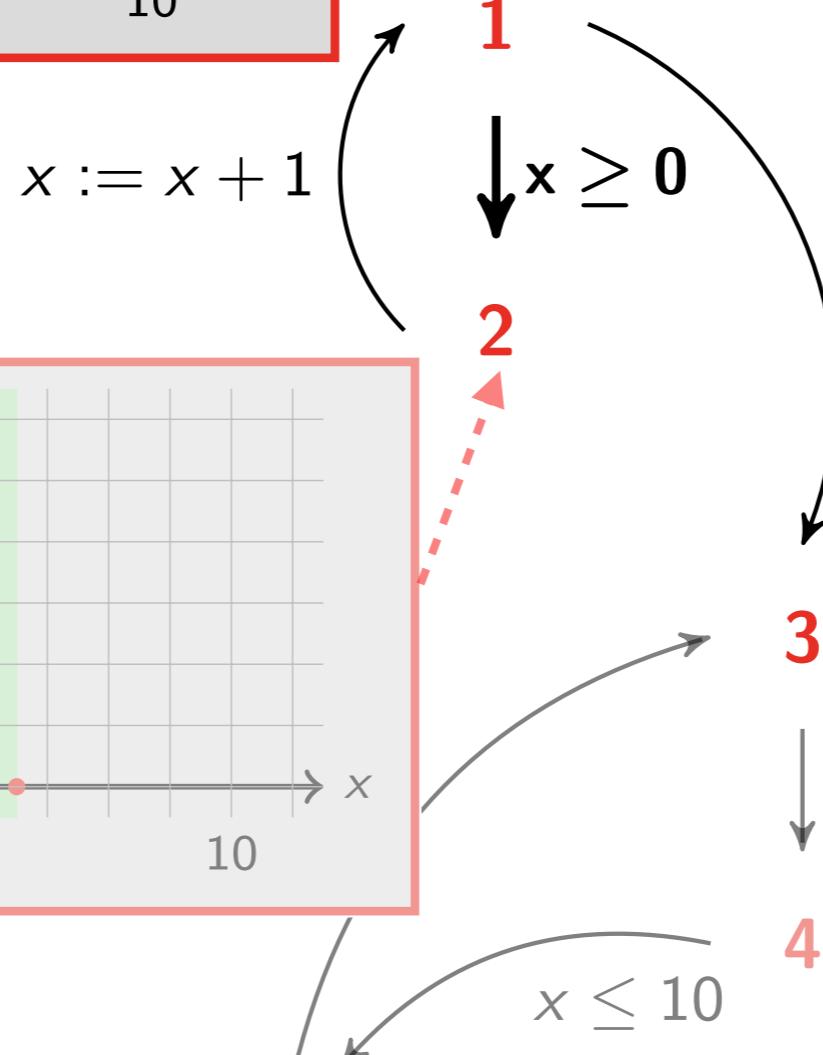
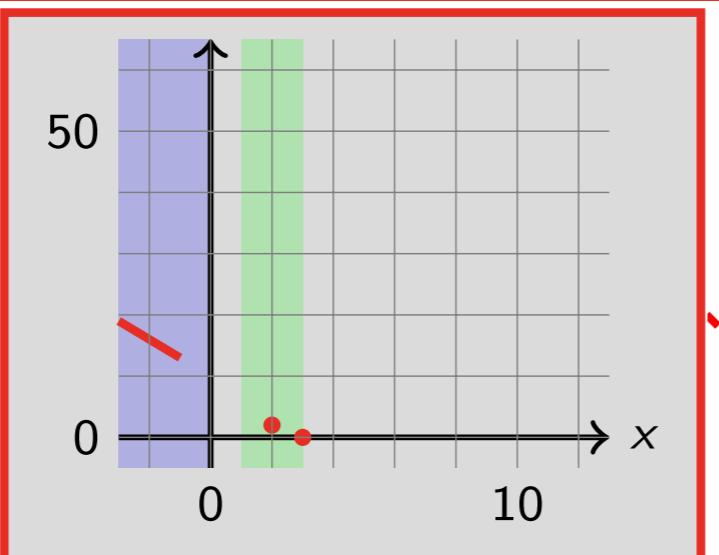
Property

$$\diamondsuit x = 3$$

5

Example

```
int : x, y
while 1(x ≥ 0)
2x := x + 1
od
while 3( true )
if 4( x ≤ 10
5x := x +
else
6x := -x
od 7
```



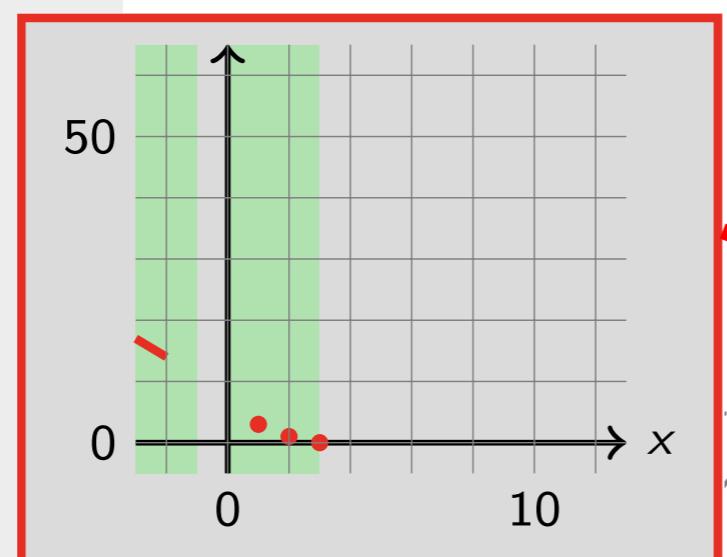
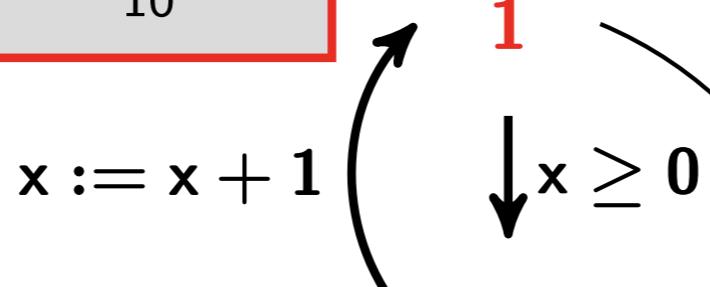
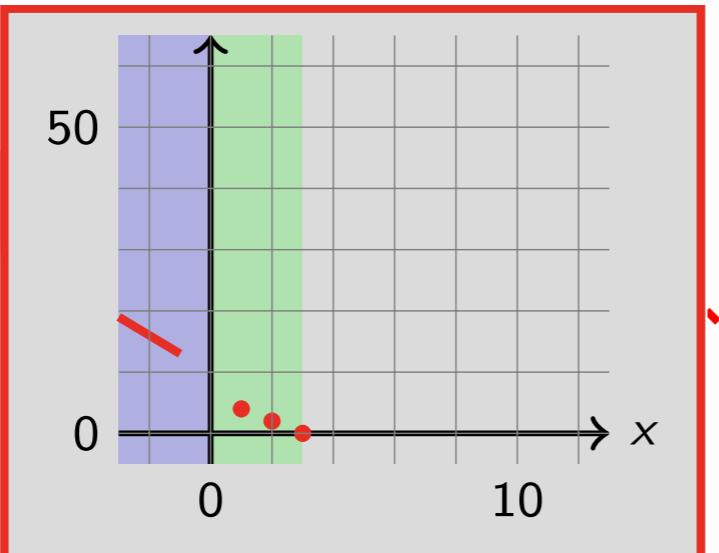
Property

$$\diamond \quad x = 3$$

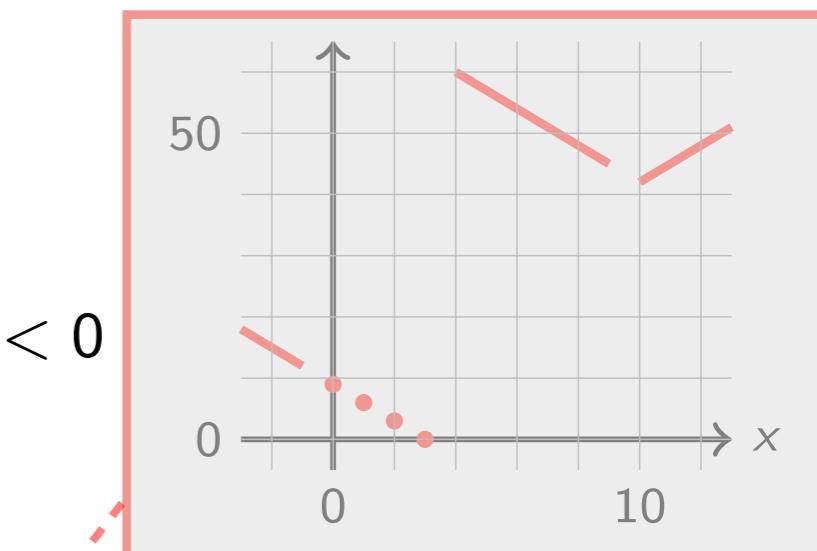
5

Example

```
int : x, y
while 1( $x \geq 0$ )
  2 x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 x := x + 1
  else
    6 x := -x
  od7
```



5



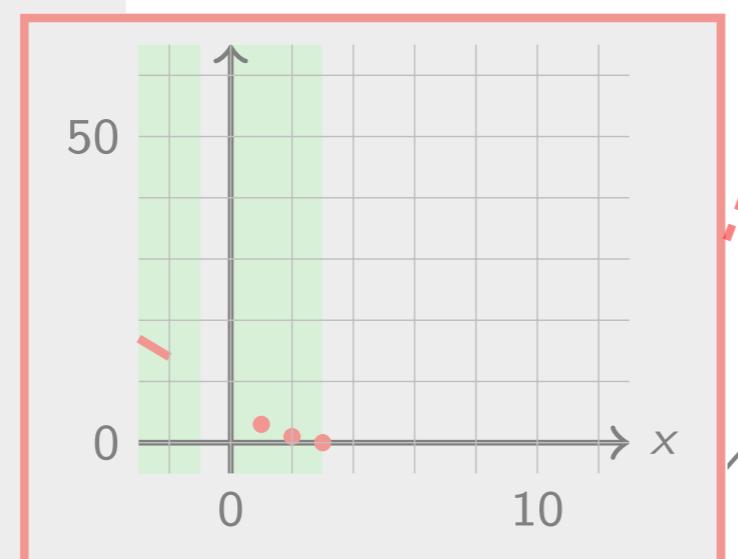
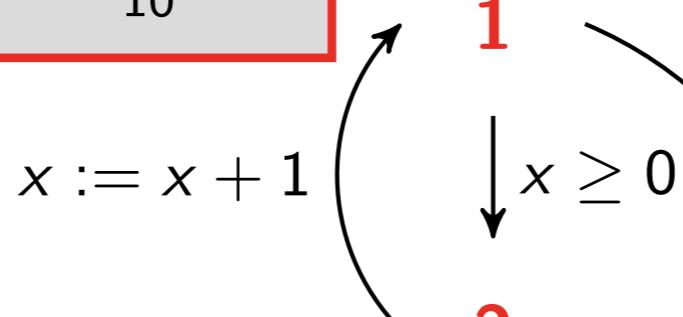
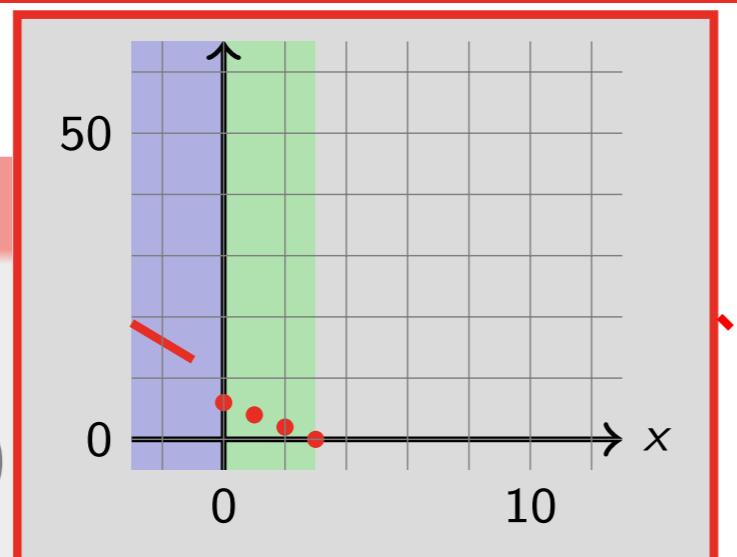
7
6
4
3
false
true
 $x := -x$

Property

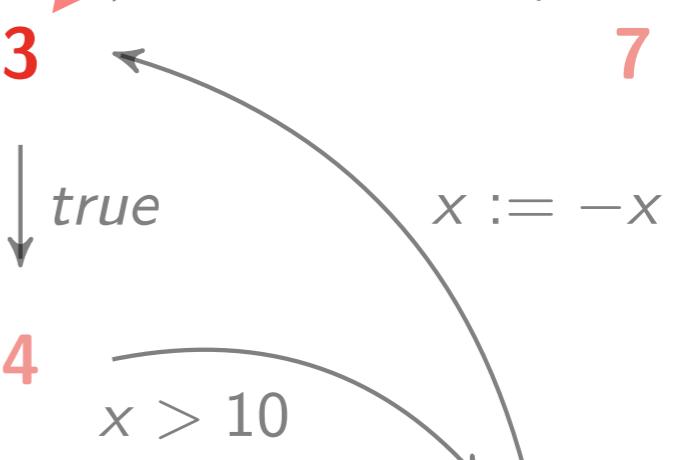
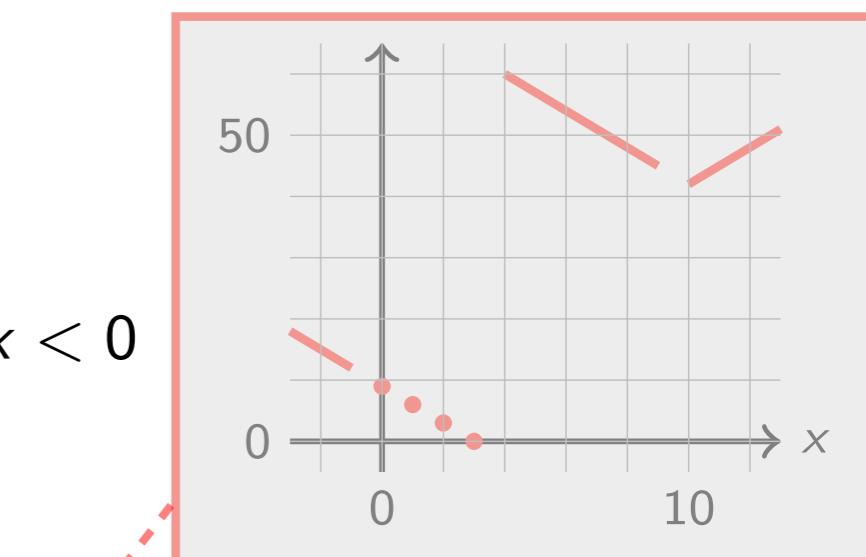
$$\diamondsuit x = 3$$

Example

```
int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7
```



5



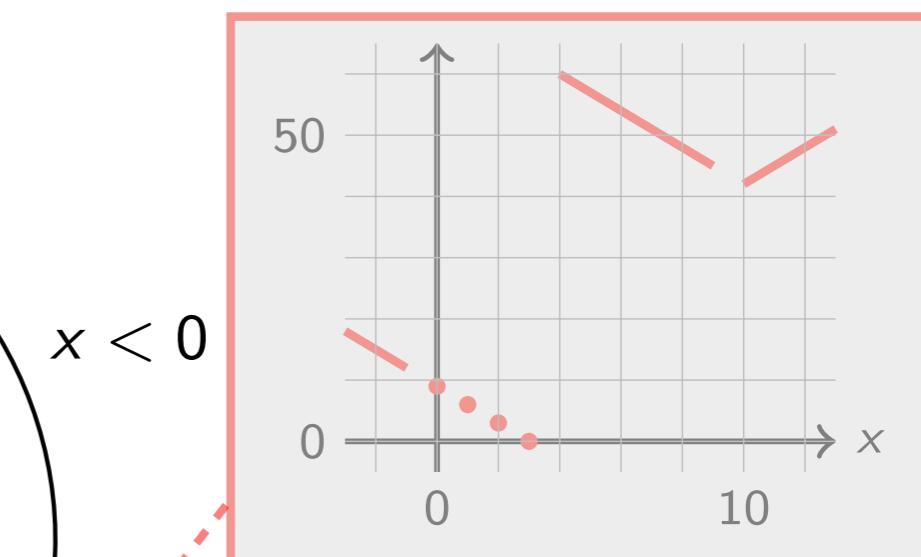
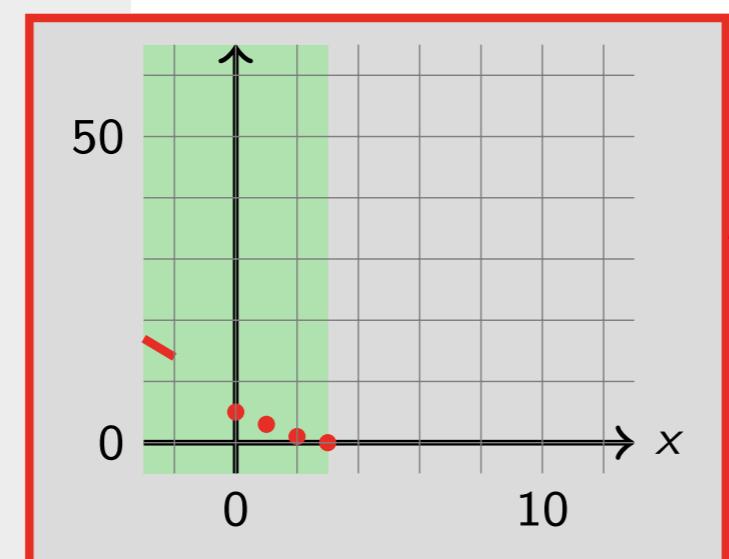
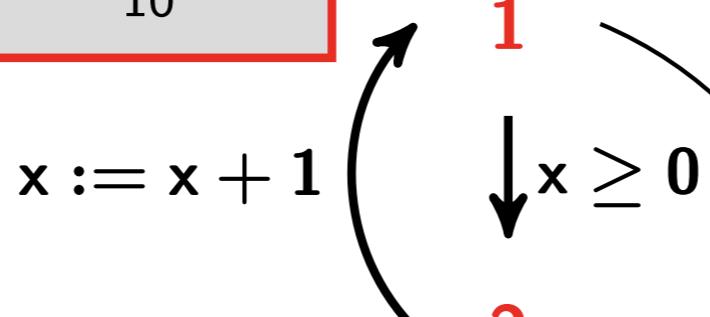
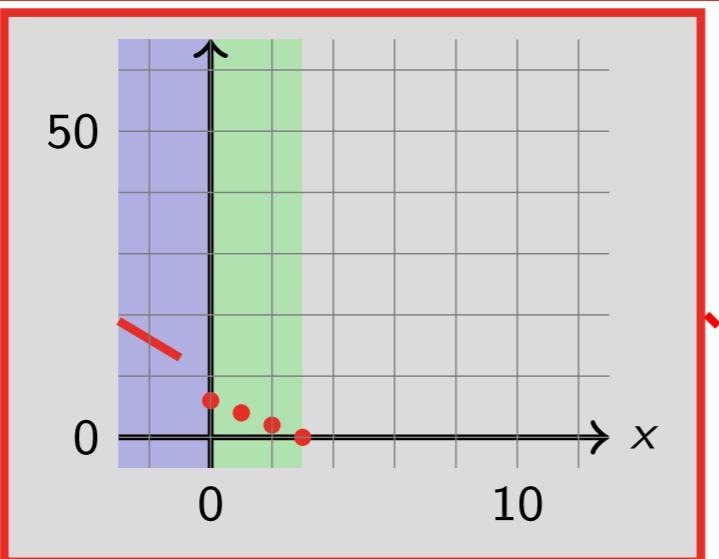
6

Property

$$\diamondsuit x = 3$$

Example

```
int : x, y
while 1( $x \geq 0$ )
  2 x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 x := x + 1
  else
    6 x := -x
  od7
```



Property

$$\diamondsuit x = 3$$

5

Example

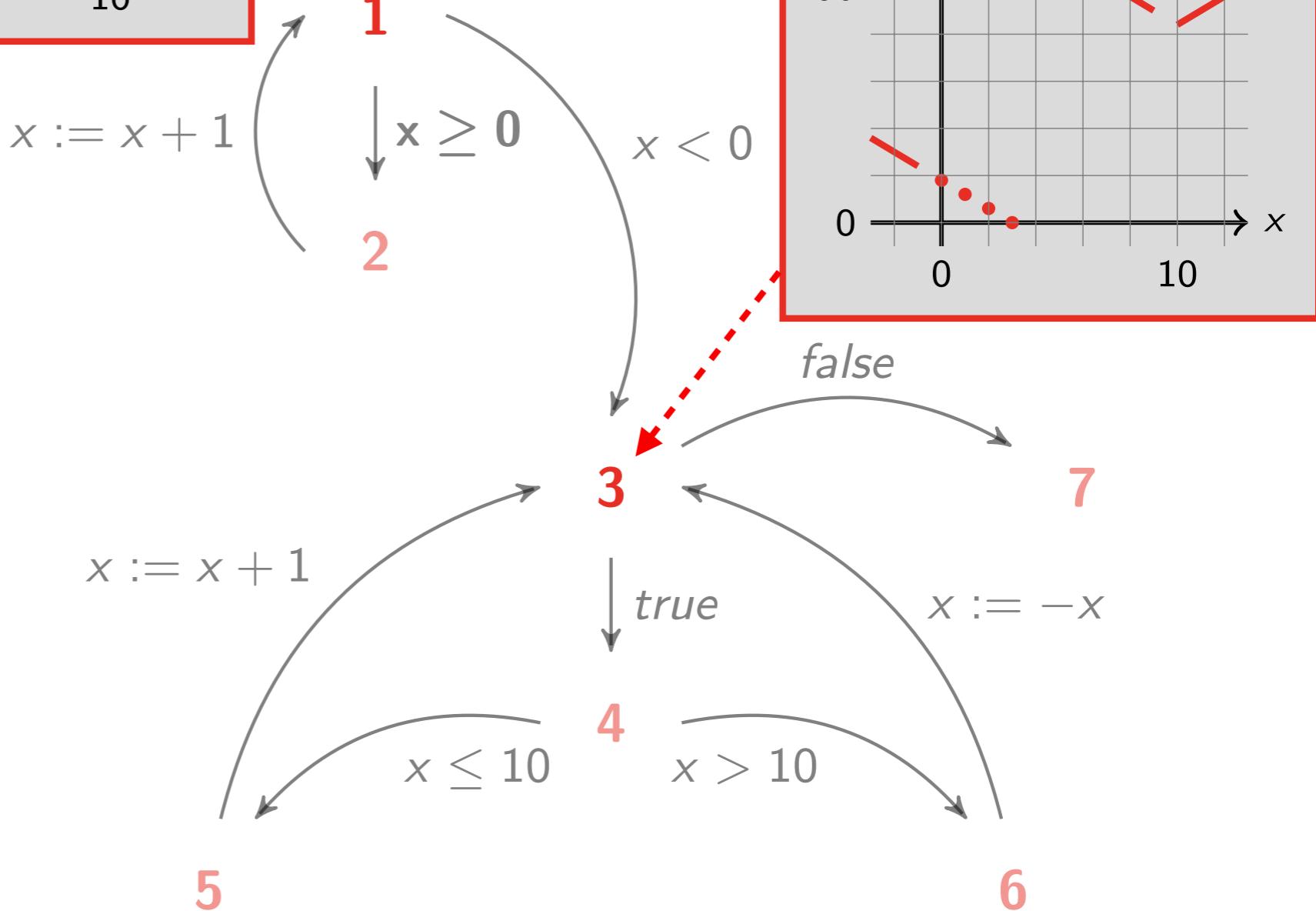
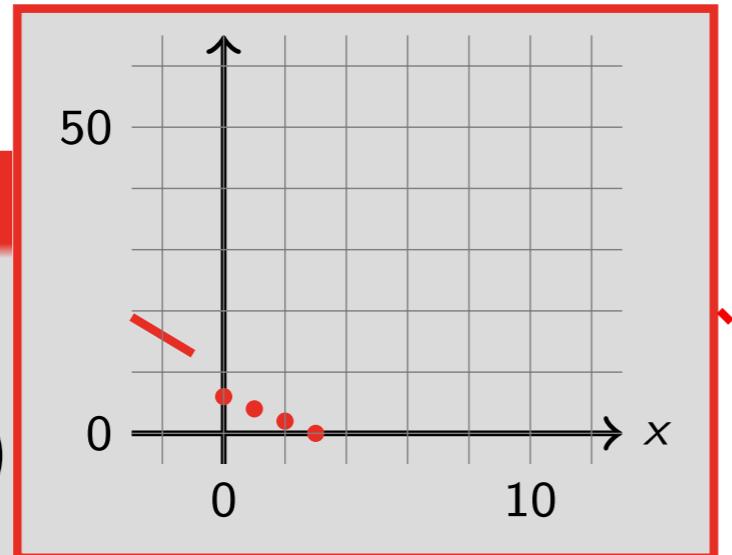
```

int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7

```

Property

$$\diamondsuit x = 3$$



5

Example

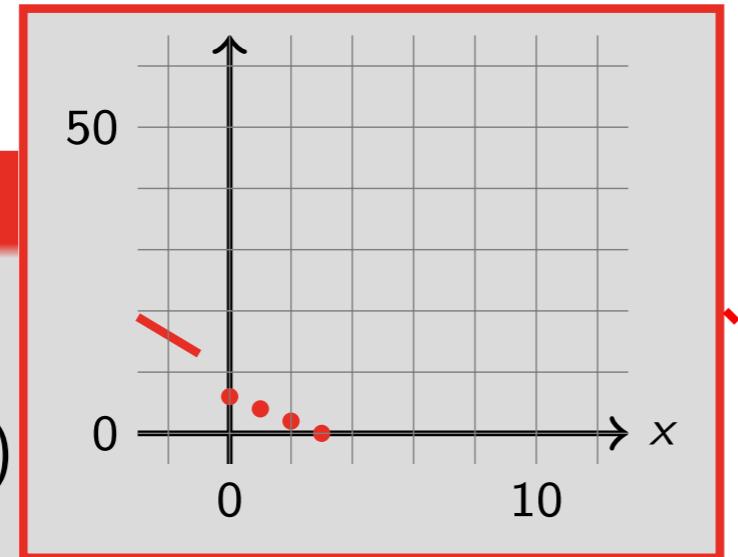
```

int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7

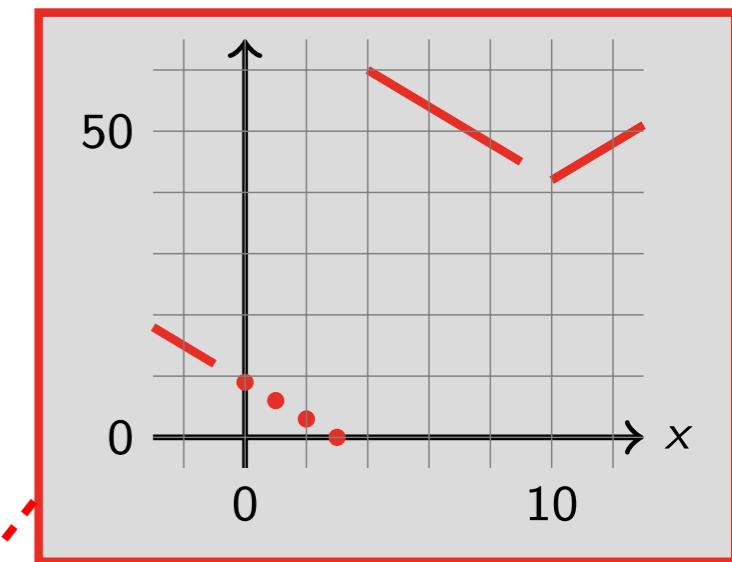
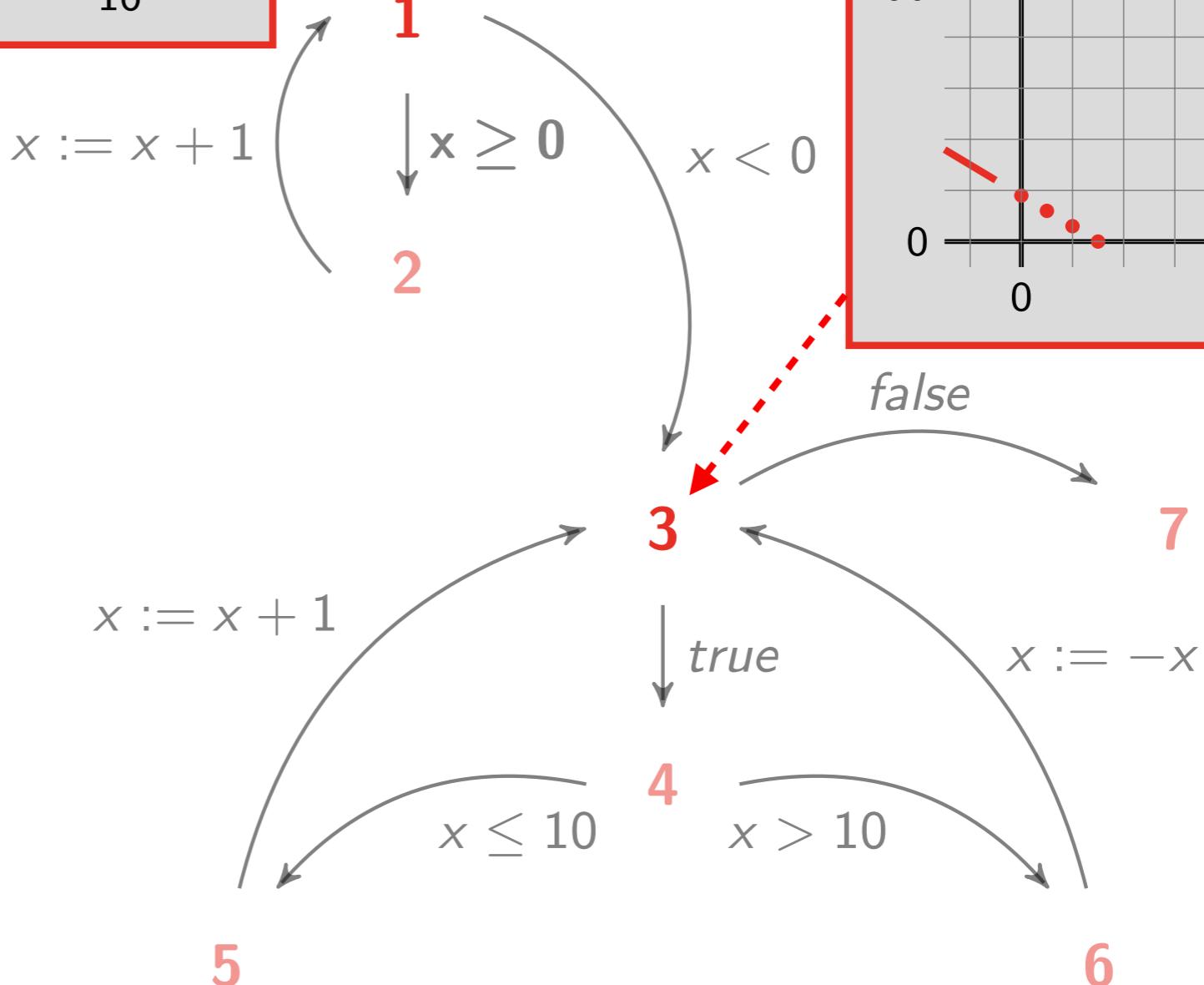
```

Property

$$\diamondsuit x = 3$$



the analysis gives $x \leq 3$ as
sufficient precondition



Recurrence Properties

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

$$\mathcal{T}_r^\varphi \in \Sigma \rightarrow \emptyset$$

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi$$

$$F_r^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} v(s) & s \in \text{dom}(\mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(v))}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

$$\mathcal{T}_r^\varphi \in \Sigma \multimap \mathbb{O}$$

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi = \bigcap_{i \in \mathbb{N}} (F_r^\varphi)^i$$

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

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

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi = \bigcap_{i \in \mathbb{N}} (F_r^\varphi)^i$$

$$(F_r^\varphi)^0 = \mathcal{T}_g^\varphi$$

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

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

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi = \bigcap_{i \in \mathbb{N}} (F_r^\varphi)^i$$

$$(F_r^\varphi)^0 = \mathcal{T}_g^\varphi$$

$$(F_r^\varphi)^1 = \mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(\mathcal{T}_g^\varphi))}$$

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

$$\mathcal{T}_r^\varphi \in \Sigma \multimap \mathbb{O}$$

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi = \bigcap_{i \in \mathbb{N}} (F_r^\varphi)^i$$

$$(F_r^\varphi)^0 = \mathcal{T}_g^\varphi$$

$$(F_r^\varphi)^1 = \mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(\mathcal{T}_g^\varphi))}$$

$$(F_r^\varphi)^2 = \mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(\mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(\mathcal{T}_g^\varphi))}))}$$

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

$$\mathcal{T}_r^\varphi \in \Sigma \multimap \mathbb{O}$$

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi = \bigcap_{i \in \mathbb{N}} (F_r^\varphi)^i$$

$$(F_r^\varphi)^0 = \mathcal{T}_g^\varphi$$

$$(F_r^\varphi)^1 = \mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(\mathcal{T}_g^\varphi))}$$

$$(F_r^\varphi)^2 = \mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(\mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(\mathcal{T}_g^\varphi))}))}$$

⋮

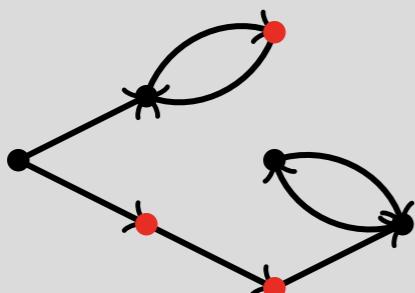
program \mapsto maximal trace semantics $\rightarrow \varphi$ -recurrence semantics

$$\mathcal{T}_r^\varphi \in \Sigma \rightarrow \emptyset$$

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi$$

$$F_r^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} v(s) & s \in \text{dom}(\mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(v))}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



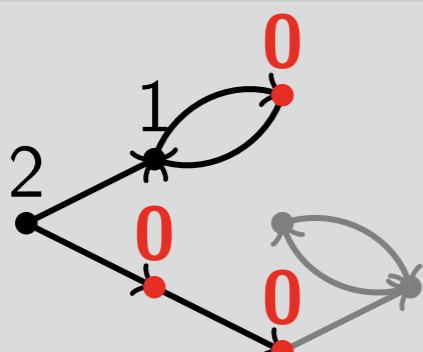
program \mapsto maximal trace semantics $\rightarrow \varphi$ -recurrence semantics

$$\mathcal{T}_r^\varphi \in \Sigma \rightarrow \emptyset$$

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi$$

$$F_r^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} v(s) & s \in \text{dom}(\mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(v))}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



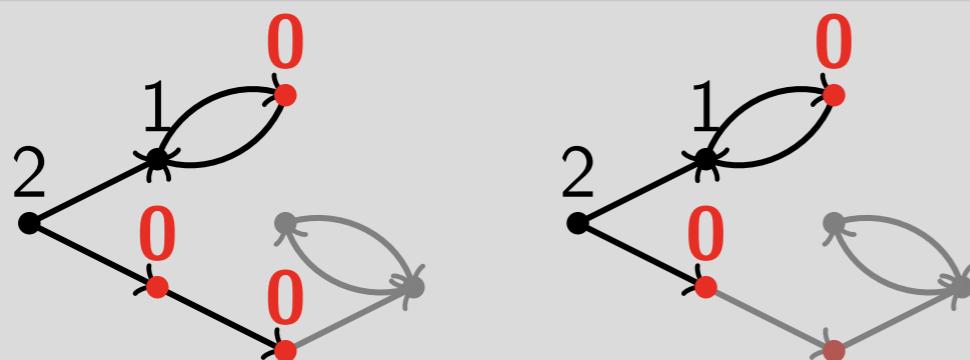
program \mapsto maximal trace semantics $\rightarrow \varphi$ -recurrence semantics

$$\mathcal{T}_r^\varphi \in \Sigma \rightarrow \emptyset$$

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi$$

$$F_r^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} v(s) & s \in \text{dom}(\mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(v))}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



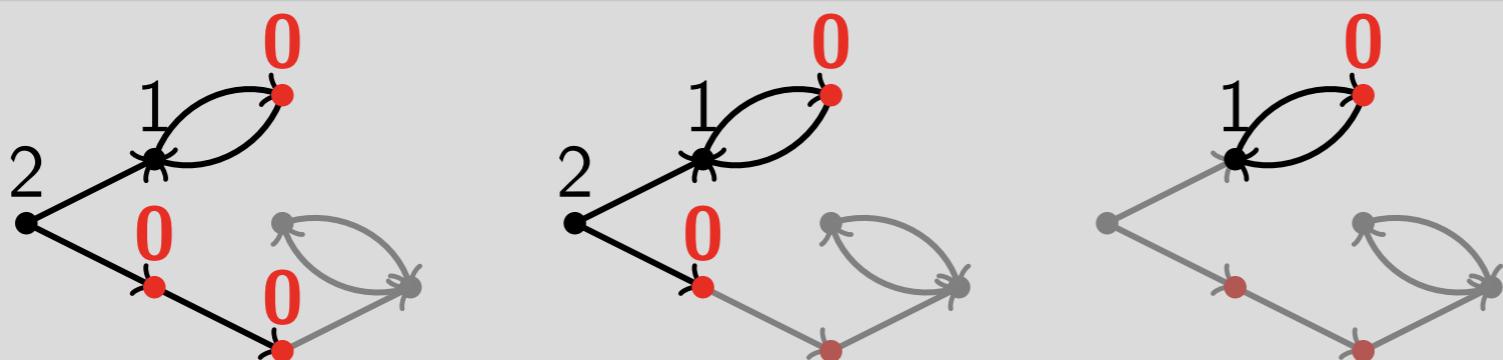
program \mapsto maximal trace semantics $\rightarrow \varphi$ -recurrence semantics

$$\mathcal{T}_r^\varphi \in \Sigma \rightarrow \emptyset$$

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi$$

$$F_r^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} v(s) & s \in \text{dom}(\mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(v))}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example



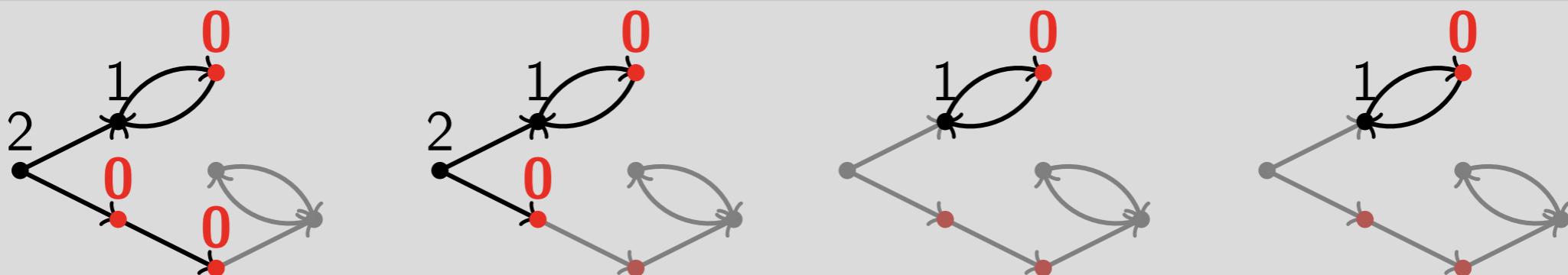
program \mapsto maximal trace semantics $\rightarrow \varphi$ -recurrence semantics

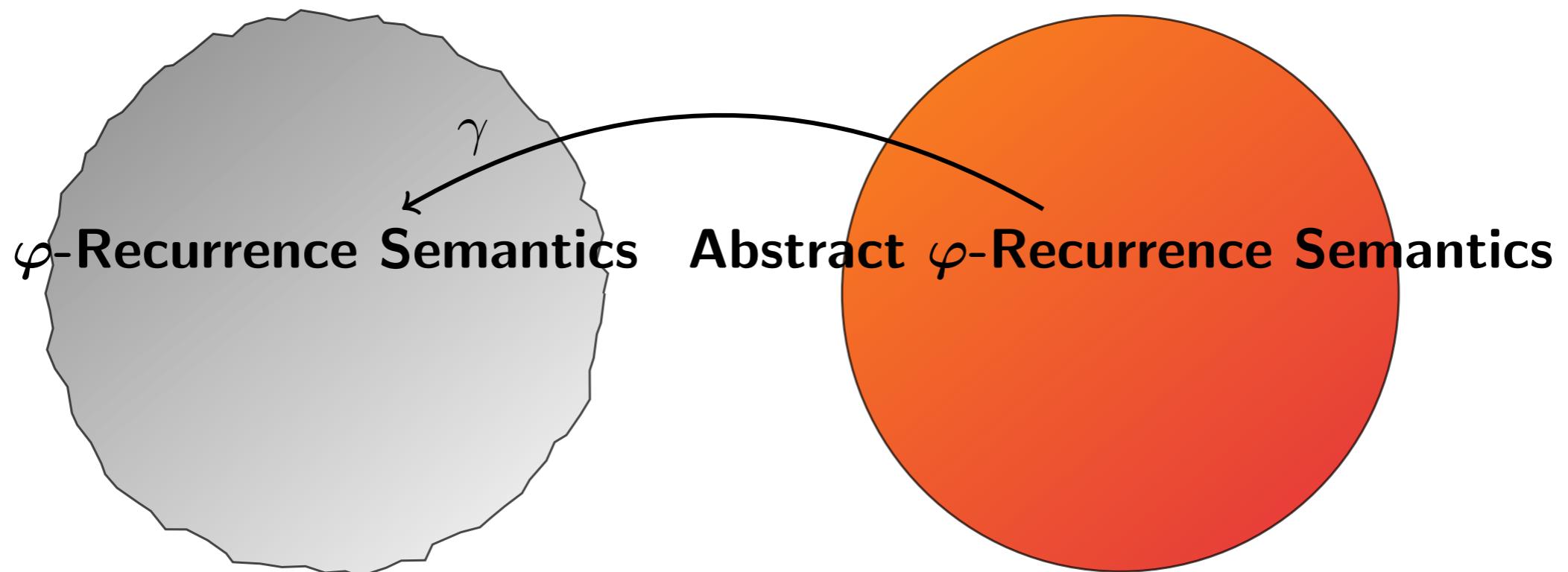
$$\mathcal{T}_r^\varphi \in \Sigma \rightarrow \emptyset$$

$$\mathcal{T}_r^\varphi \stackrel{\text{def}}{=} \text{gfp}_{\mathcal{T}_g^\varphi} F_r^\varphi$$

$$F_r^\varphi(v)s \stackrel{\text{def}}{=} \begin{cases} v(s) & s \in \text{dom}(\mathcal{T}_g^{\varphi \cap \widetilde{\text{pre}}(\text{dom}(v))}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example





Example

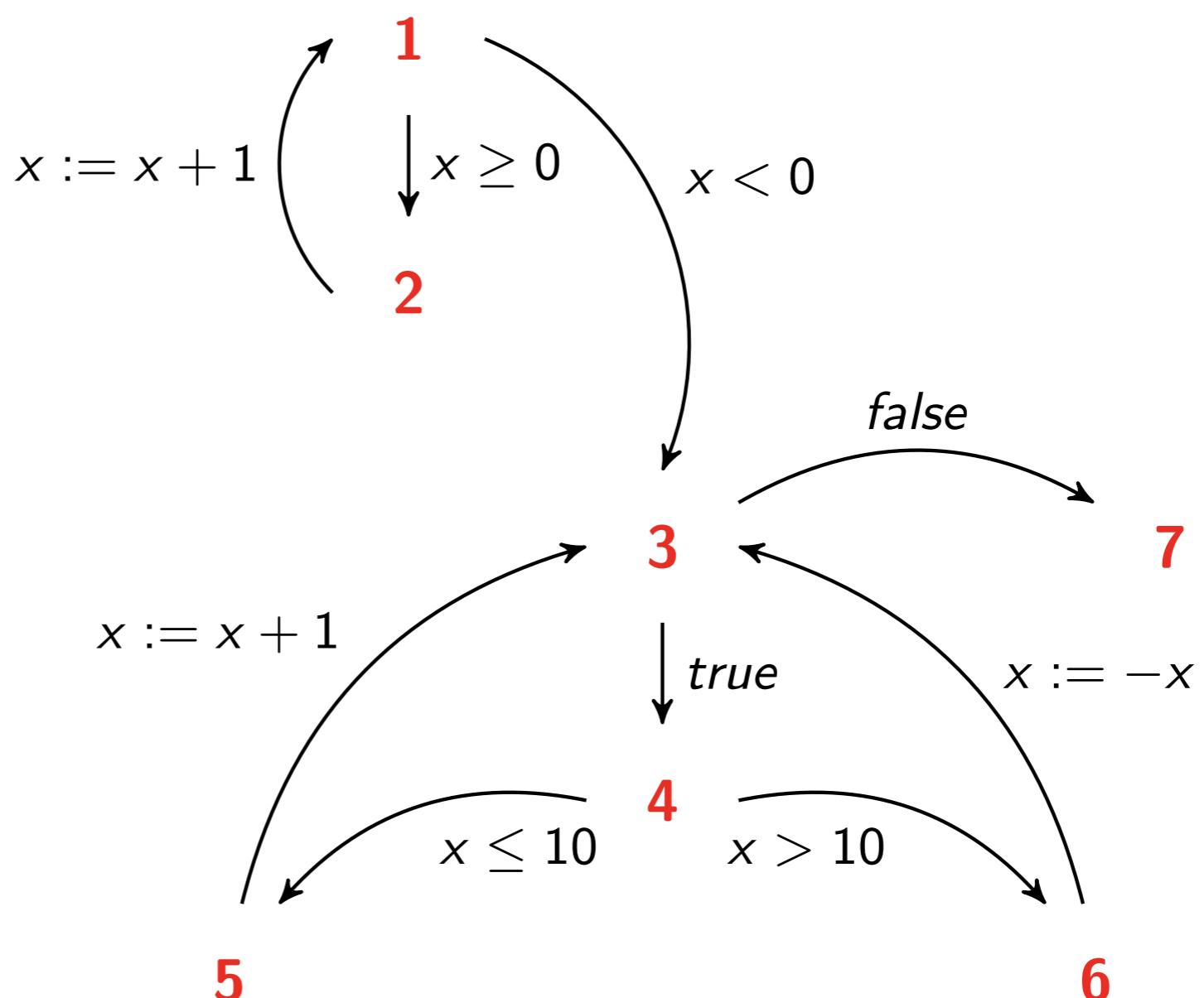
```

int : x, y
while 1( $x \geq 0$ ) do
  2 $x := x + 1$ 
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 $x := x + 1$ 
  else
    6 $x := -x$ 
od7

```

Property

$$\square \diamond x = 3$$



Example

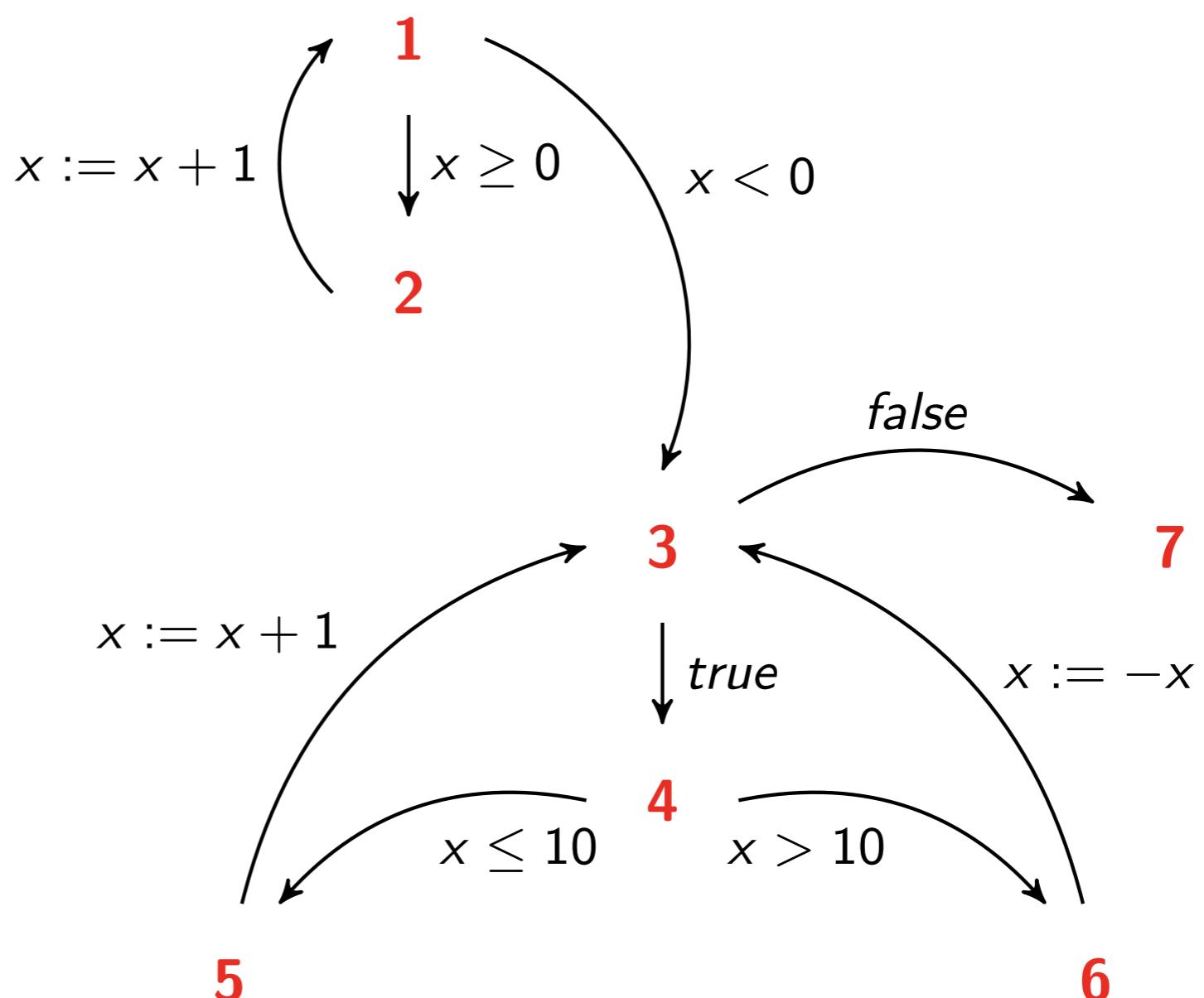
```

int : x, y
while 1( $x \geq 0$ ) do
  2 $x := x + 1$ 
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5 $x := x + 1$ 
  else
    6 $x := -x$ 
od7

```

Property

$$\square \diamond x = 3$$



Example

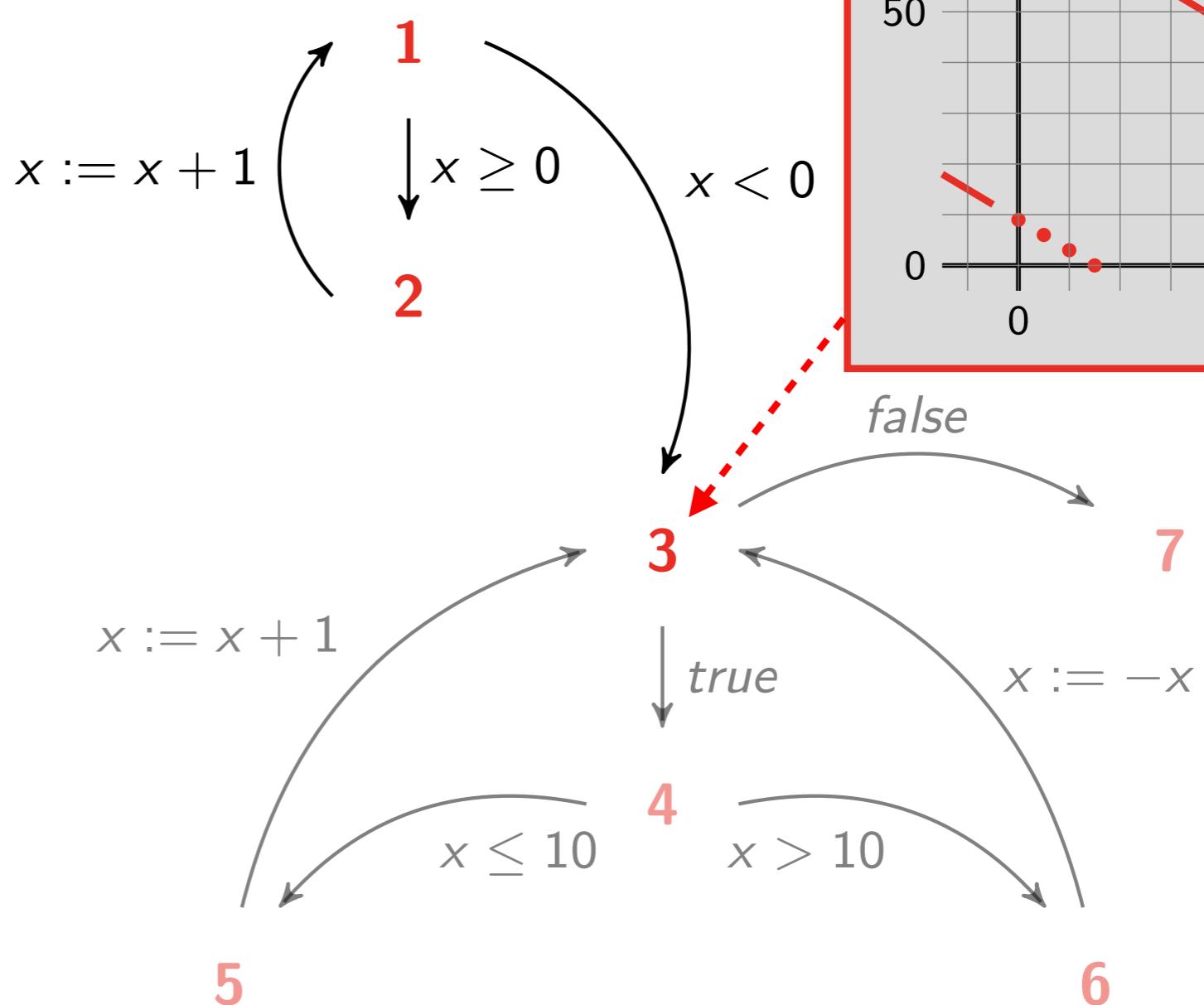
```

int : x, y
while 1( $x \geq 0$ ) do
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7

```

Property

$$\square \diamond x = 3$$

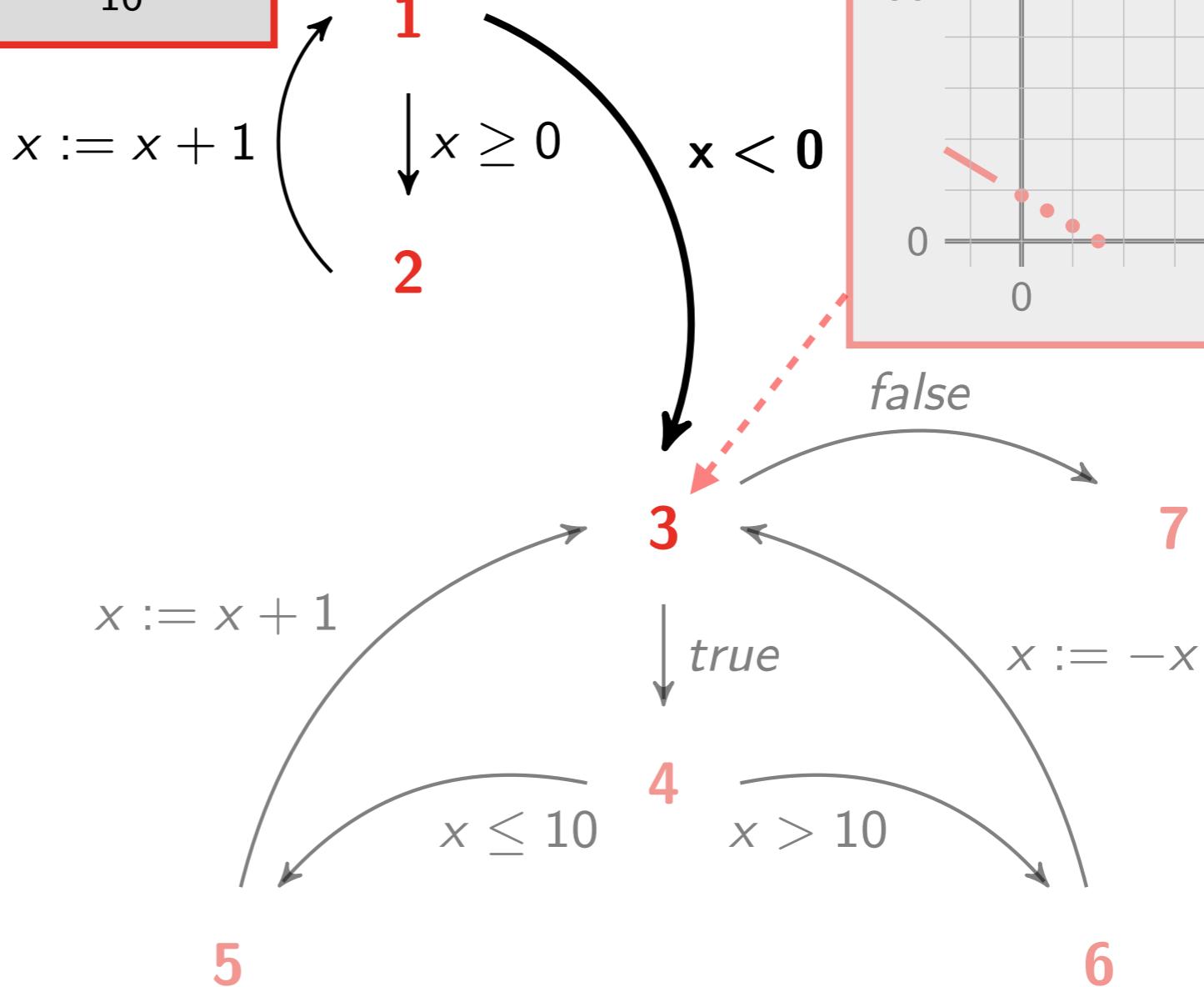
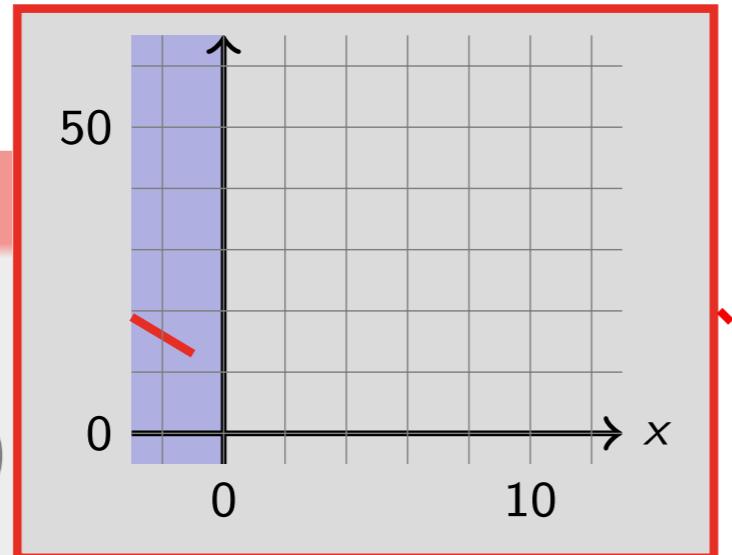


Example

```
int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7
```

Property

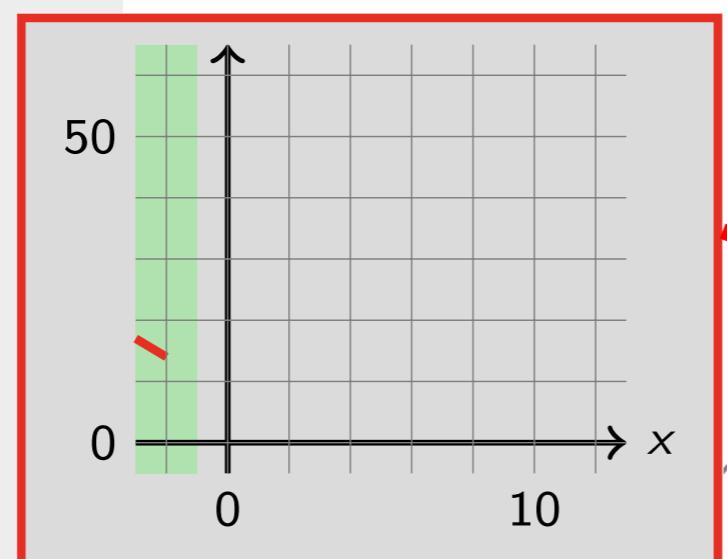
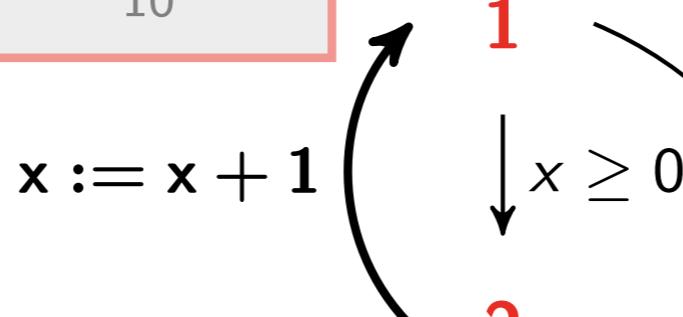
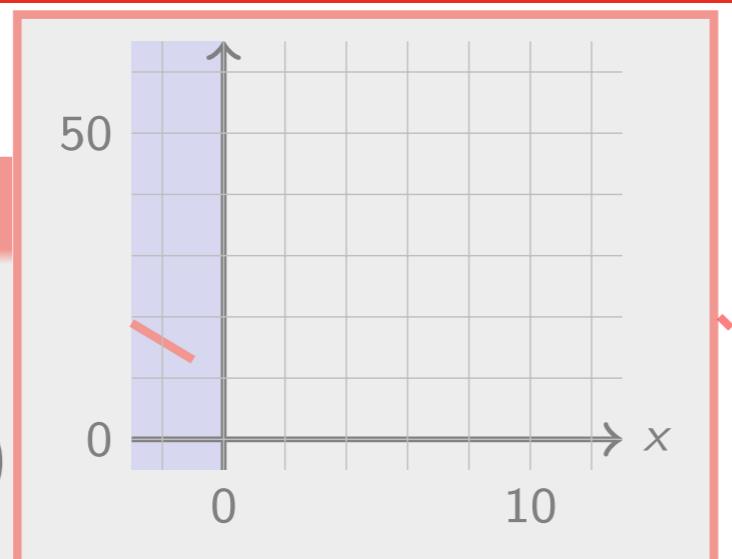
$$\square \diamond x = 3$$



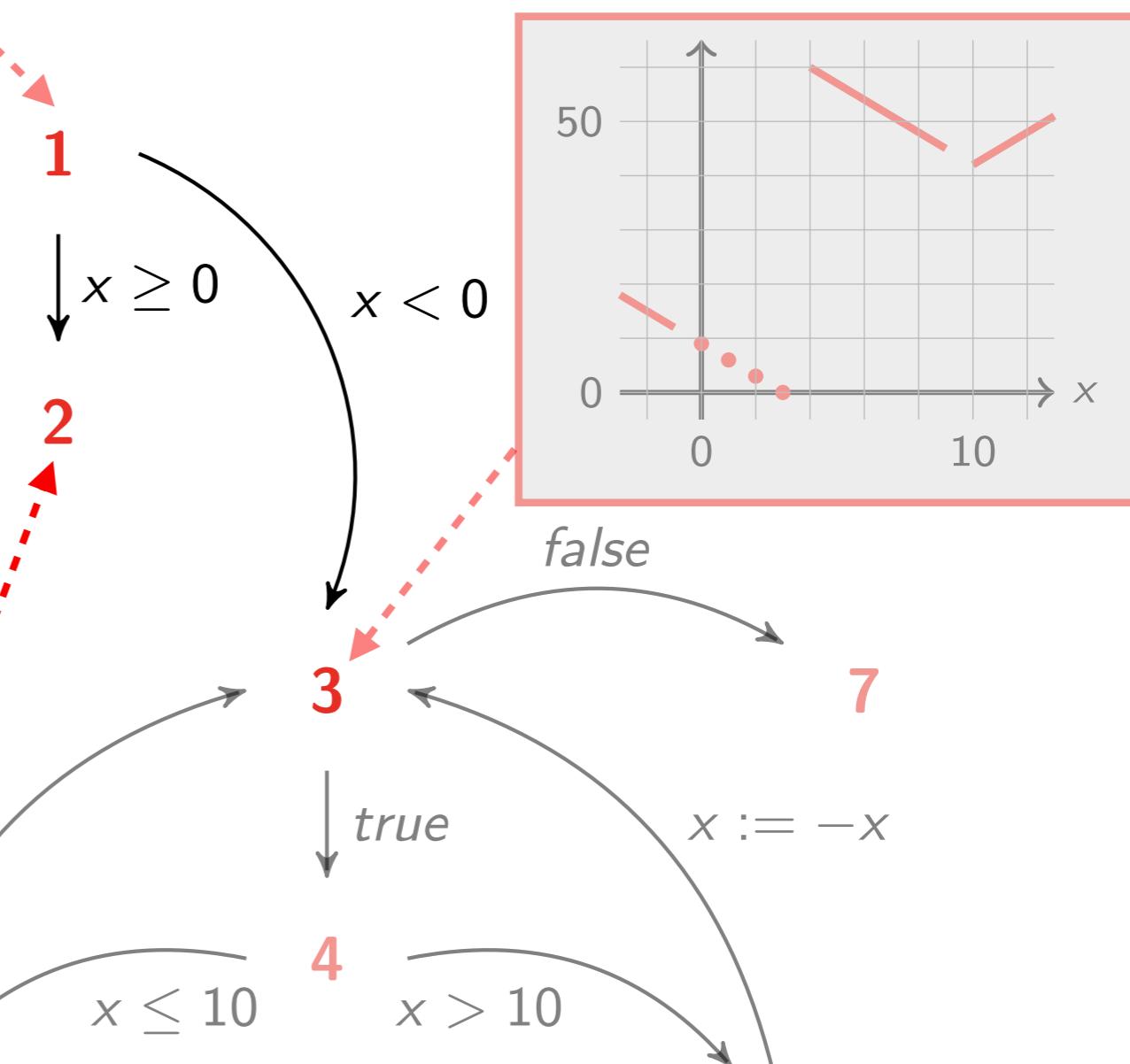
5

Example

```
int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7
```



5



Property

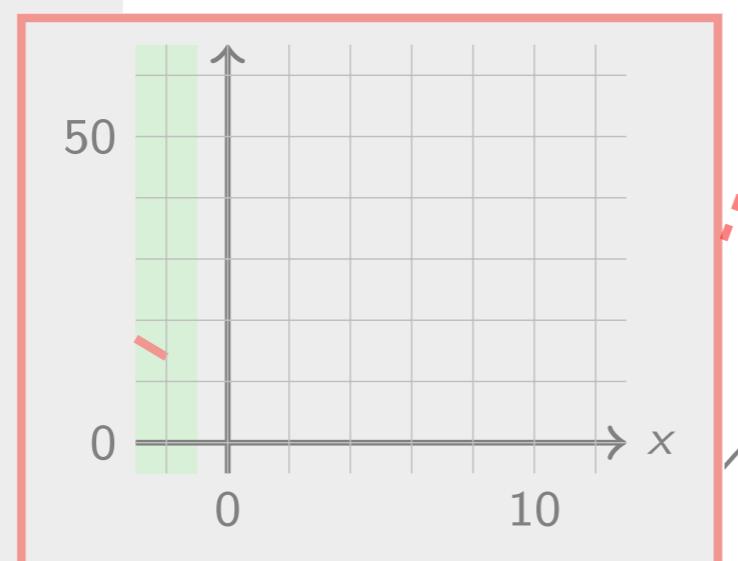
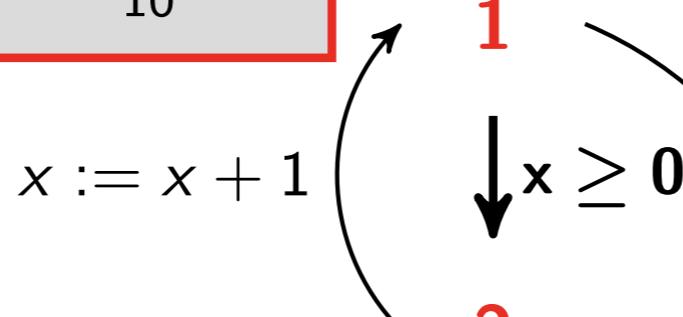
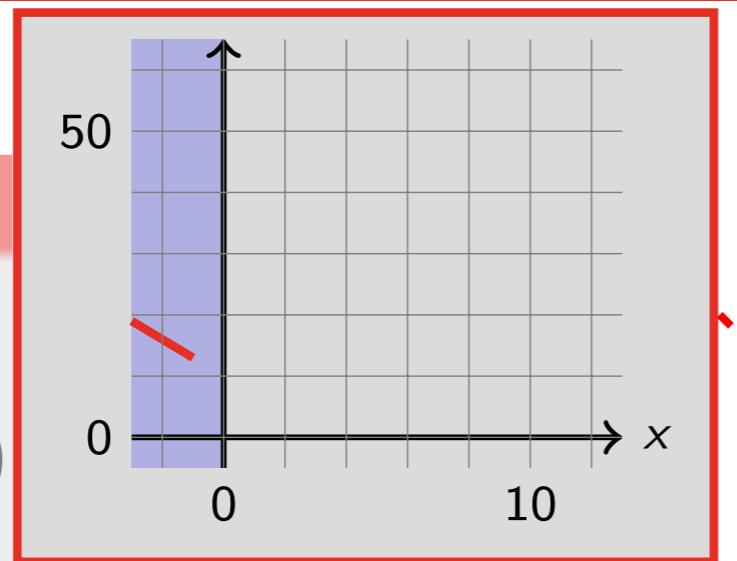
$$\square \diamond x = 3$$

Example

```

int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7

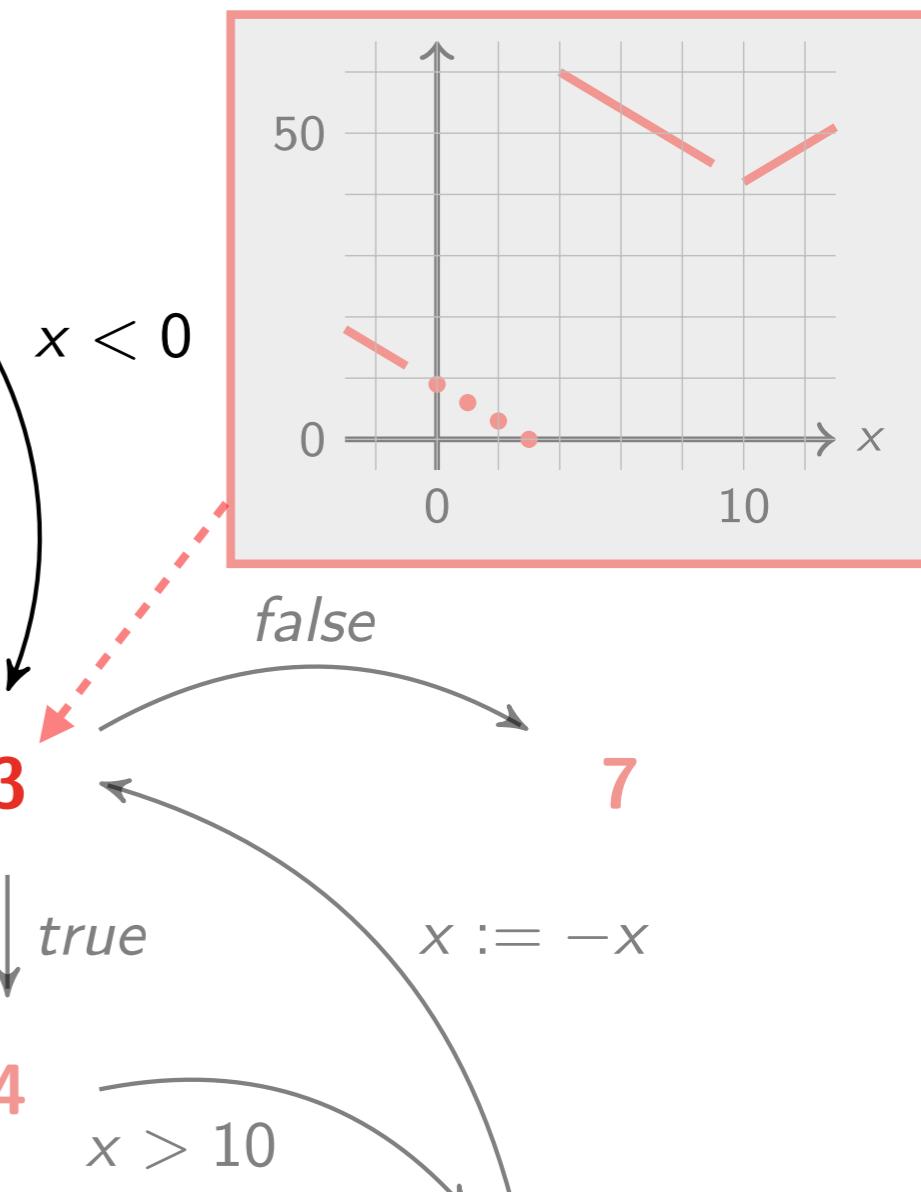
```



5

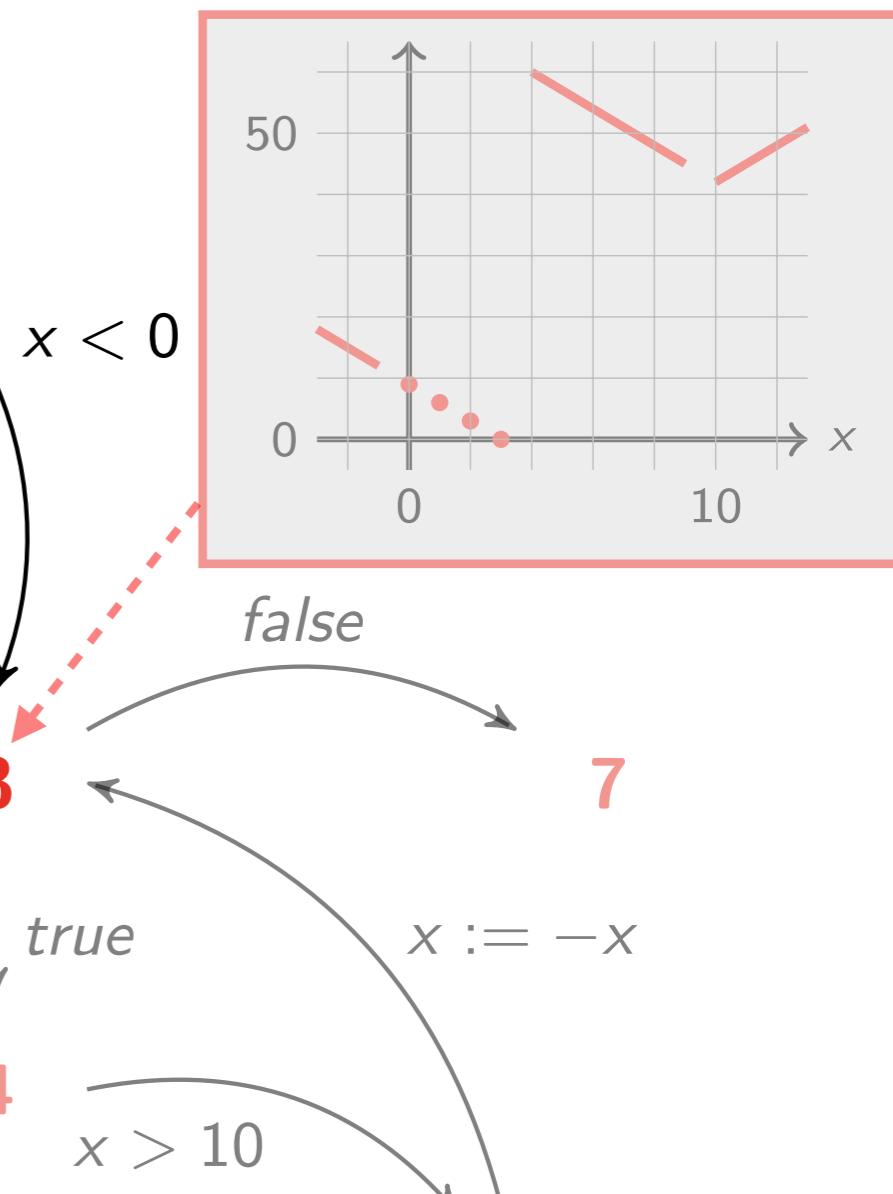
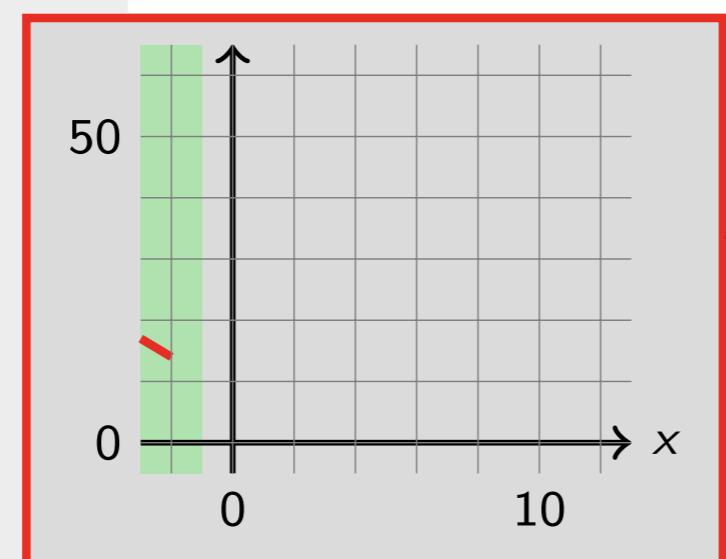
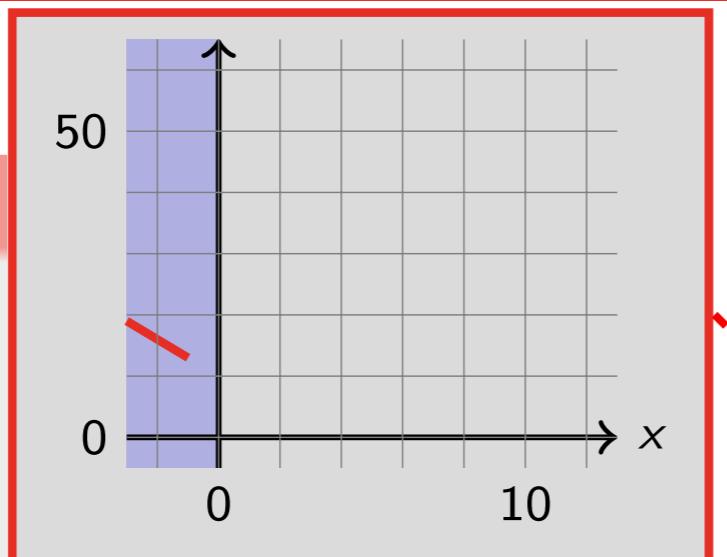
Property

$$\square \diamond x = 3$$



Example

```
int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7
```



5

Property

$$\Box \Diamond x = 3$$

Example

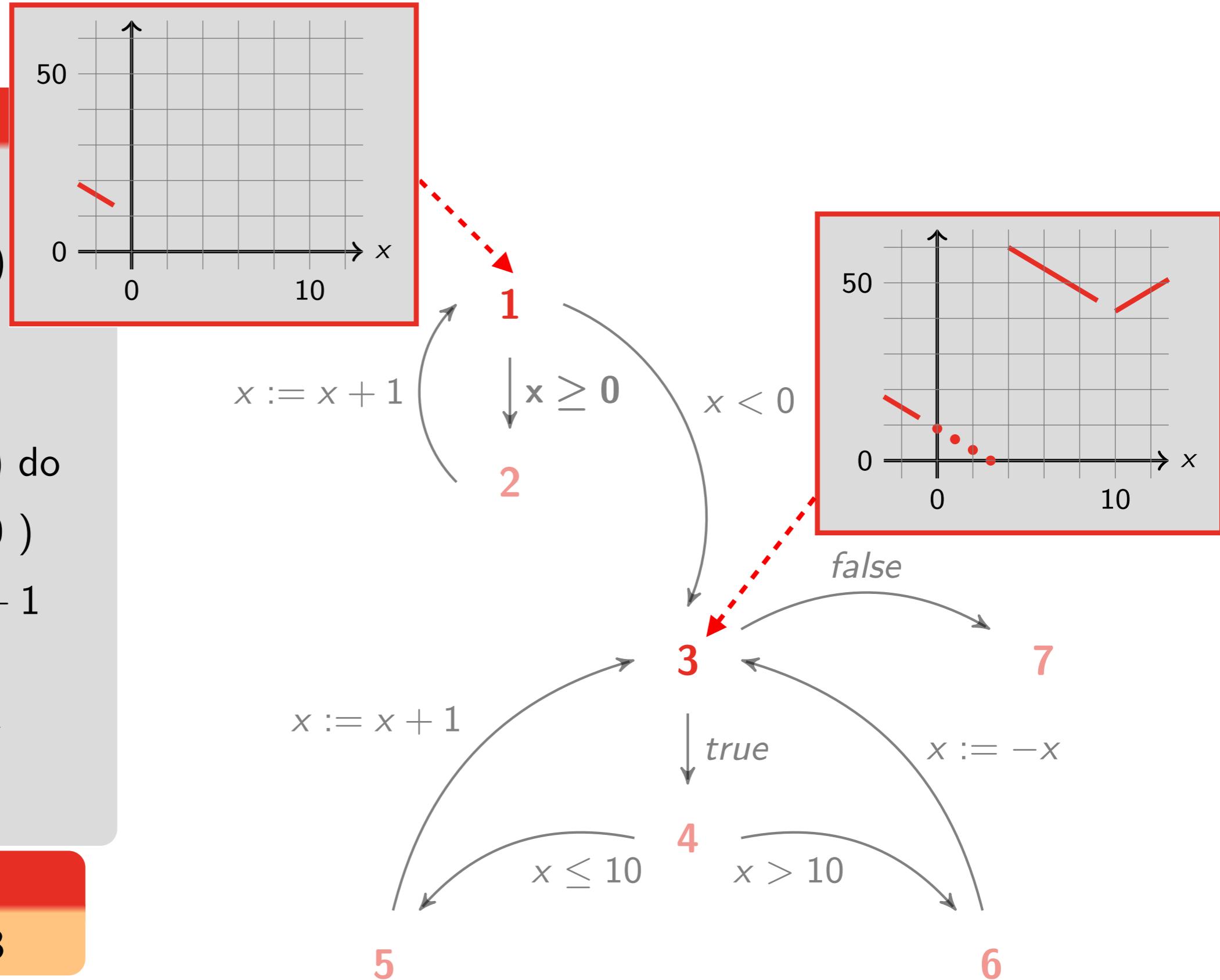
```

int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
  od7

```

Property

$$\square \diamond x = 3$$



5

Example

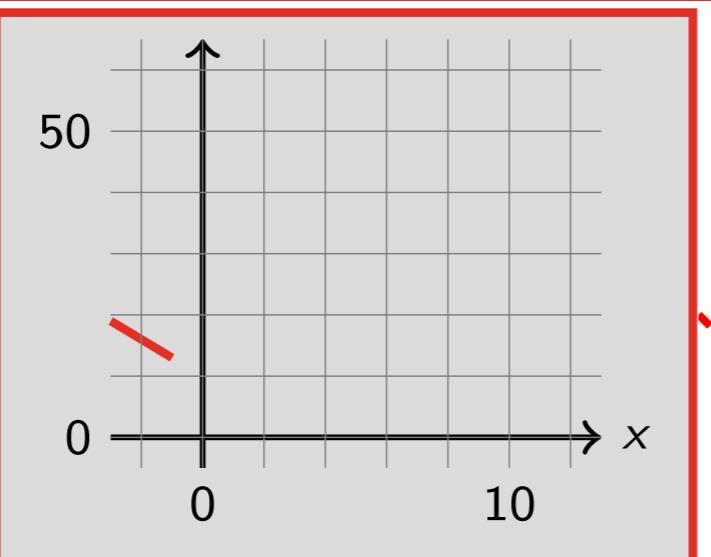
```

int : x, y
while 1( $x \geq 0$ )
  2x := x + 1
od
while 3( true ) do
  if 4(  $x \leq 10$  )
    5x := x + 1
  else
    6x := -x
od7

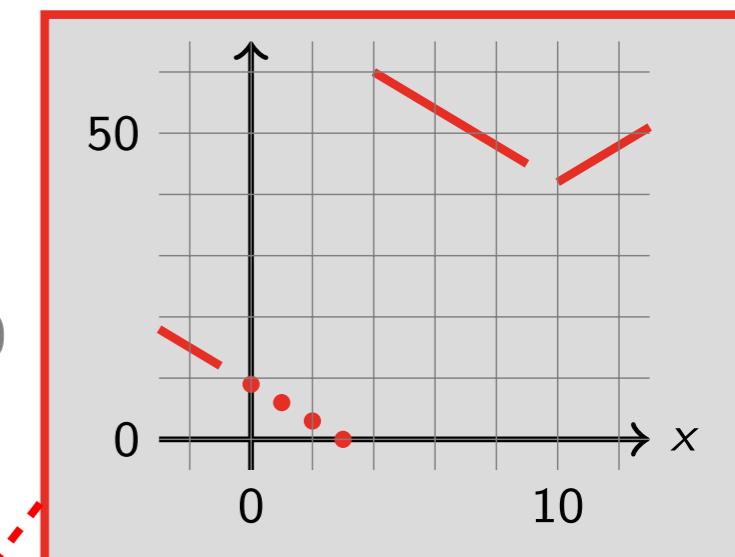
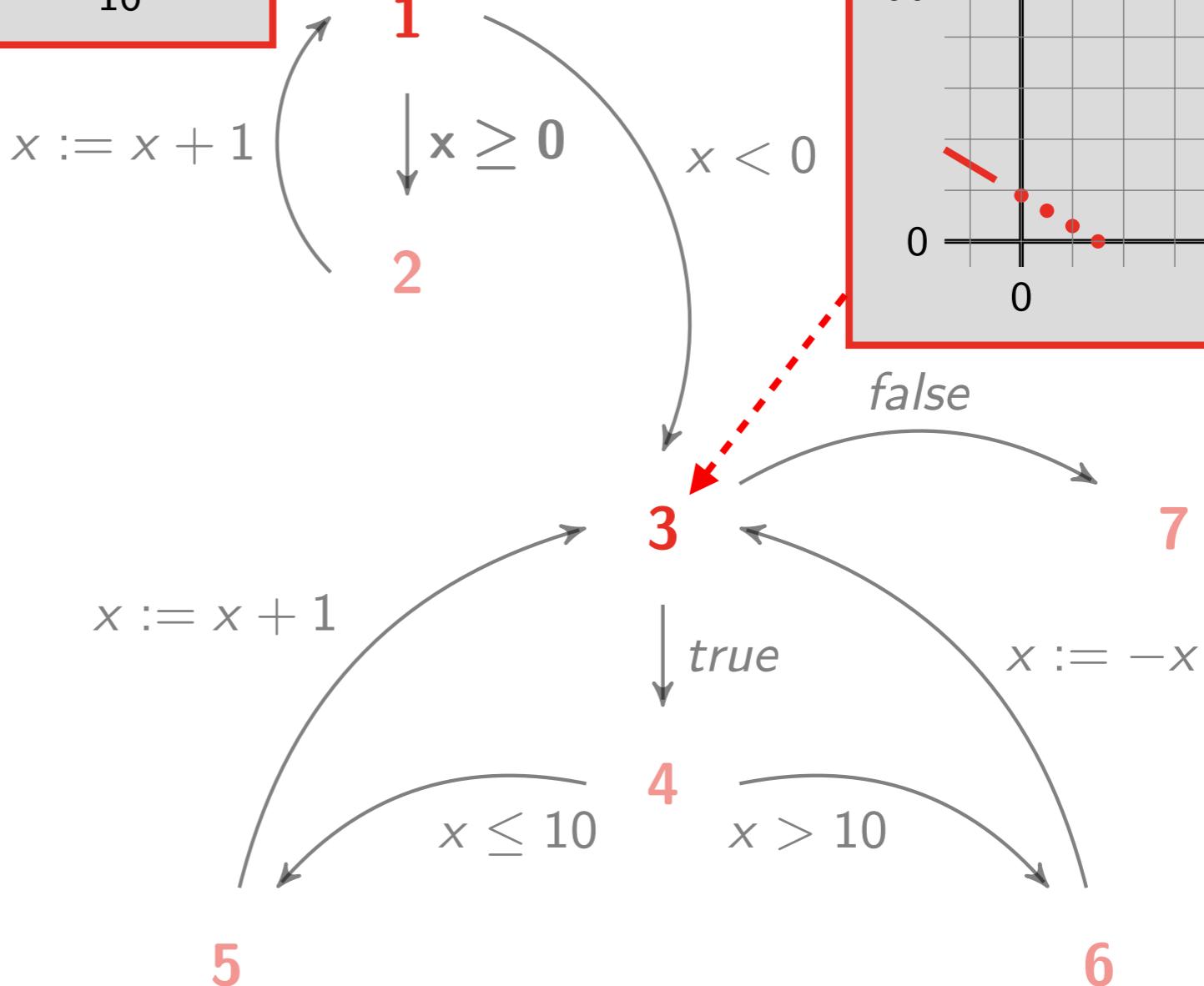
```

Property

$$\square \diamond x = 3$$



the analysis gives $x < 0$ as
sufficient precondition



5

6

Peterson's Algorithm

Example

$flag_1 := 0; flag_2 := 0;$

```
[while 1( true ) do
  2flag1 := 1
  3turn := 2
  await 4( flag2 = 0 ∨ turn = 1 )
  5CRITICAL_SECTION
  6flag1 := 0]
```

```
[while 1( true ) do
  2flag2 := 1
  3turn := 1
  await 4( flag1 = 0 ∨ turn = 2 )
  5CRITICAL_SECTION
  6flag2 := 0]
```

Property

$\square \diamond 5 : \text{true}$

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:

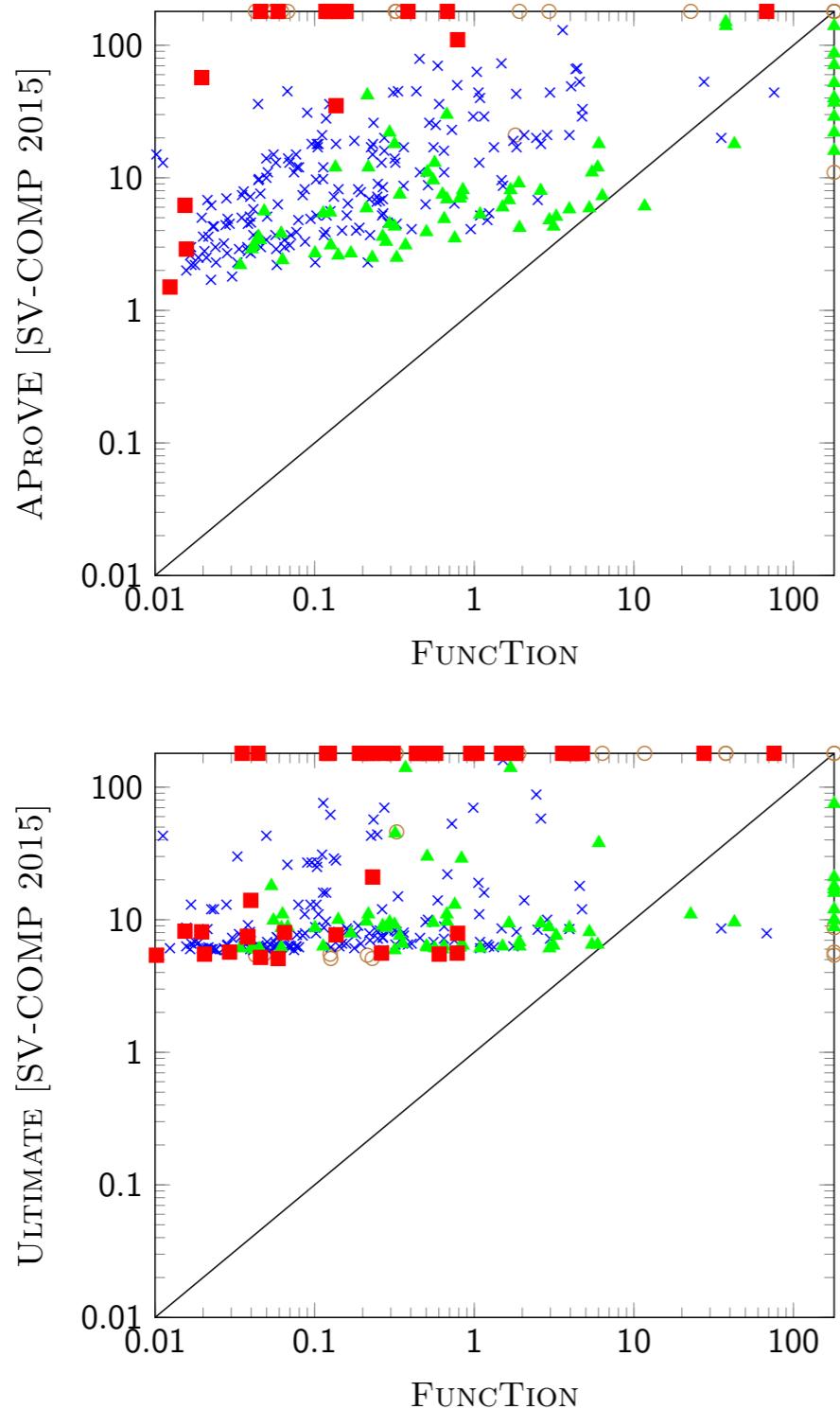
Forward option(s):

- Widening delay:

Backward option(s):

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

- accepts programs written in a small **subset of C**
- the only basic data type are **mathematical integers**
- implemented in **OCaml**
- abstract domains implemented on top of the **APRON library**
- participating to **SV-COMP 2014** (demo) and **SV-COMP 2015**



- 288 terminating programs
- 1% only FUNCTION (■)
- 2.7% only AProVE (▲)
- 1% only HIPTNT+ (▲)
- 1.7% only ULTIMATE (▲)
- 0.7% none (○)

Conclusions

To Infinity...

- family of **abstract domains** for liveness properties
 - piecewise-defined ranking functions
 - sufficient preconditions for liveness properties
- instances based on affine and **ordinal-valued functions**

Future Work

... and Beyond!

- **potential** termination and **non-termination**
- more **abstract domains**
 - non-linear ranking functions
 - better widening
 - machine integers and floats
- **fair termination** and other **liveness** properties
- heap-manipulating and **concurrent** programs

Thank You!



- **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)
- **D'Silva & Urban** - *Conflict-Driven Conditional Termination* (CAV 2015)

IN CONCLUSION,

AAAAAAA!!!



THE BEST THESIS DEFENSE IS A GOOD THESIS OFFENSE.