Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation

Caterina Urban and **Antoine Miné**

14th January 2015

VMCAI 2015

Mumbai, India

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

$$\Box\varphi$$

- **guarantee properties**: *"something good happens at least once"* (e.g., termination)

$$\Diamond\varphi$$

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

$$\Box\Diamond\varphi$$

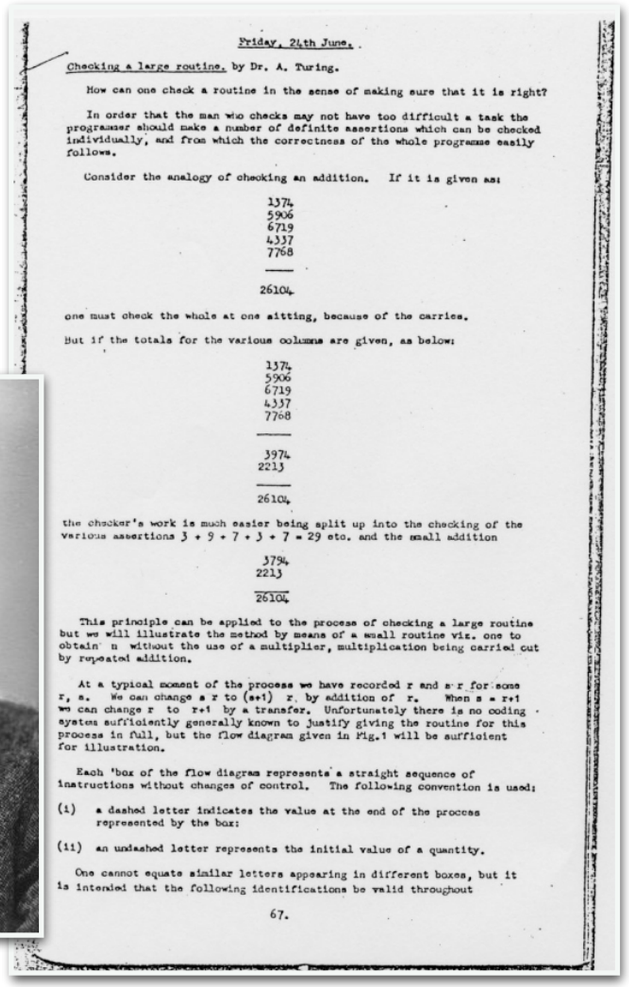
- **persistence properties**: *"something good eventually happens continuously"* (e.g., stabilization)

$$\Diamond\Box\varphi$$

Manna & Pnueli - *A Hierarchy of Temporal Properties* (PODC 1990)

Ranking Functions

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



Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

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

1374
5906
6719
4337
7768

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.
5906
6719
4337
7768

3974
2213

26104

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

379.
2213

26104

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 viz. one to obtain n 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. r for some r, s. We can change s r to (s+1) r, by addition of r. When s = r+1 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 in Fig.1 will be sufficient for illustration.

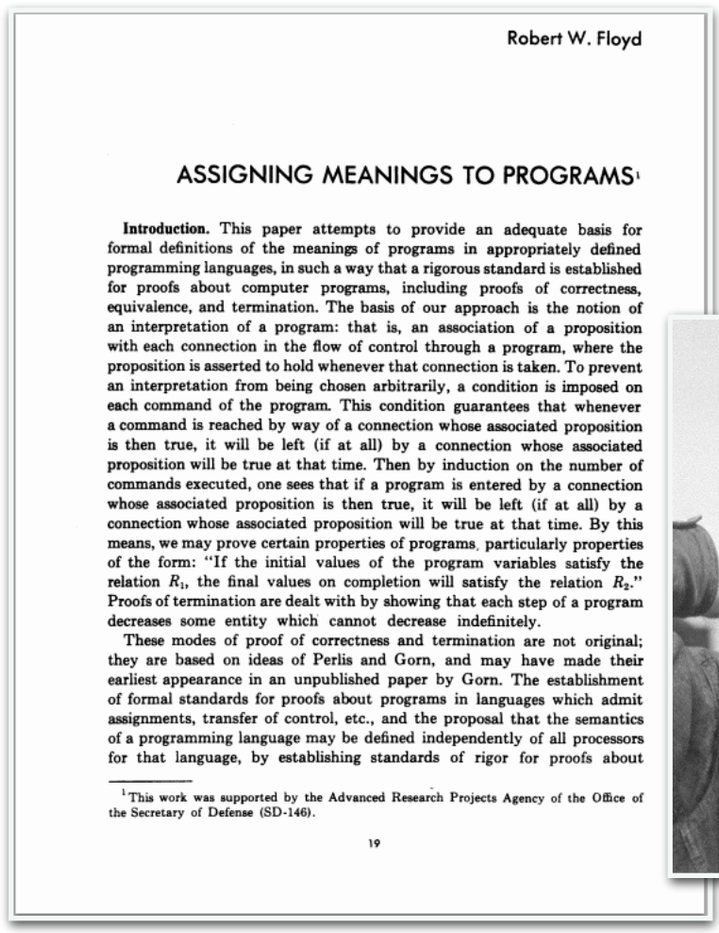
Each 'box' of the flow diagram represents a straight sequence of instructions without changes of control. The following convention is used:

(i) a dashed letter indicates the value at the end of the process represented by the box:

(ii) 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

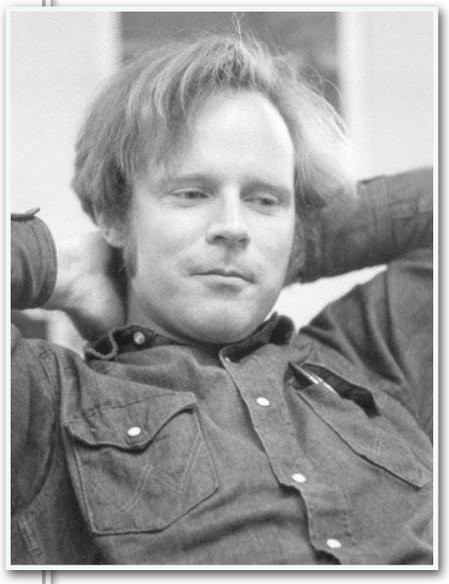
ASSIGNING MEANINGS TO PROGRAMS[1]

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 proposition 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_1, the final values on completion will satisfy the relation R_2." 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 Perlis 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

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

19



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

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

Cousot&Cousot - *An Abstract Interpretation Framework for Termination* (POPL 2012)

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

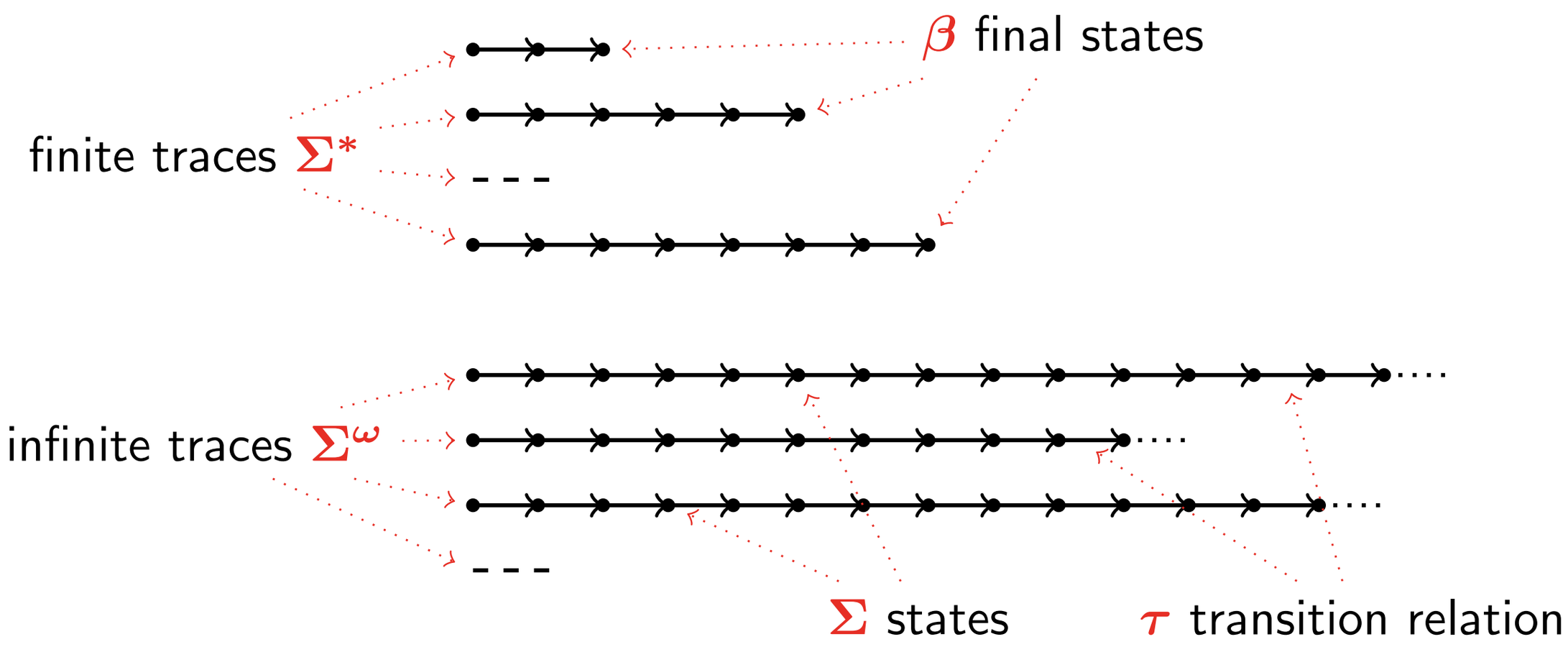
Cousot&Cousot - *An Abstract Interpretation Framework for Termination* (POPL 2012)

- **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**

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)

Termination

program $\mapsto$ **maximal trace semantics**



Termination Semantics

An Abstract Interpretation Framework for Termination

Patrick Cousot
CNRS, École Normale Supérieure, and INRIA, France
Courant Institute *, NYU, USA
cousot@ens.fr, pcousot@cims.nyu.edu

Radhia Cousot
CNRS, École Normale Supérieure, and INRIA, France
rcousot@ens.fr

Abstract

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

The abstract interpretation design principle is first illustrated for the design of new forward and backward proof, verification and analysis methods for *safety*. The safety collecting semantics defining the strongest safety property of programs is first expressed in a constructive fixpoint form. Safety proof and checking/verification methods then immediately follow by fixpoint induction. Static analysis of abstract safety properties such as invariance are constructively designed by fixpoint abstraction (or approximation) to (automatically) infer safety properties. So far, no such clear design principle did exist for termination so that the existing approaches are scattered and largely not comparable with each other.

For (1), we show that this design principle applies equally well to *potential and definite termination*. The trace-based termination collecting semantics is given a fixpoint definition. Its abstraction yields a fixpoint definition of the best variant function. By further abstraction of this best variant function, we derive the Floyd/Turing termination proof method as well as new static analysis methods to effectively compute approximations of this best variant function.

For (2), we introduce a generalization of the syntactic notion of structural induction (as found in Hoare logic) into a *semantic structural induction* based on the new semantic concept of *inductive trace cover* covering execution traces by *segments*, a new basis for formulating program properties. Its abstractions allow for generalized recursive proof, verification and static analysis methods by induction on both program structure, control, and data. Examples of particular instances include Floyd's handling of loop cut-points as well as nested loops, Burstall's intermittent assertion total correctness proof method, and Podelski-Rybalchenko transition invariants.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]; D.3.1 [*Formal Definitions and Theory*]; F.3.1 [*Specifying and Verifying and Reasoning about Programs*].

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

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

1. Introduction

Floyd/Turing program proof methods for invariance and termination [24, 40, 59] have inspired most sound static analysis methods.

For static *invariance* analysis by abstract interpretation [19, 21], a key step is to express the strongest invariant as a fixpoint and next to approximate this strongest invariant to automatically infer an abstract inductive invariant using the constructive fixpoint approximation methods.

For static *termination* analysis, the discovery of variant functions is either decidable in limited cases [54] or else is based on the Floyd/Turing idea of variant functions into well-founded sets obtained by observing quantities that strictly decrease within loops while remaining lower-bounded, or dually. So most termination analysis methods indirectly reduce to a relational invariance analysis hence can reuse classical static analysis methods.

The abstract interpretation design principle is instantiated with suitable abstractions for *safety* and *termination* analysis, proof, and checking/verification (either potential termination or definite termination for nondeterministic systems).

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

The second main idea introduced in this paper both for safety and termination is that of *semantic structural induction*, including termination proofs, over *trace segment covers* and their abstractions. Trace segments are more powerful than binary relations between states which have been used traditionally in program termination proofs (for example, the transition invariants used in [53] are binary relation abstractions of the set of trace segments). Examples include structural induction on the program syntax (including loop invariants à la Floyd [40]), induction on data, à la Burstall [3], the covering of the transition relation closure by well-founded relations, à la Podelski-Rybalchenko [53], their combinations and generalizations.

2. Fixpoints, fixpoint induction, abstraction, and approximation

We express semantics as *fixpoints* of maps $f \in A \mapsto A$ i.e. elements $x \in A$ such that $x = f(x)$. We let $\mathrm{lfp}^{\sqsubseteq}_{a} f$ be the *least fixpoint* of $f \in A \mapsto A$ on the poset $\langle A, \sqsubseteq \rangle$ greater than or equal to $a \in A$, if any. The dual notion is that of *greatest fixpoint* $\mathrm{gfp}^{\sqsubseteq}_{a} f$. We write $\mathrm{lfp}^{\sqsubseteq} f$ if a is the infimum of A, and $\mathrm{lfp}\, f$ if the partial order $\sqsubseteq$ is clear from the context. By Tarski/Pataria's fixpoint theorem [50, 58], $\mathrm{lfp}^{\sqsubseteq}_{a} f = \bigsqcap \{P \in A \mid a \sqsubseteq P \wedge f(P) \sqsubseteq P\}$ exists for f increasing[1] on a complete lattice $\langle A, \sqsubseteq, a, \top, \sqcup, \sqcap \rangle$[2] or on a cpo $\langle A, \sqsubseteq, a, \sqcup \rangle$[3]. The *fixpoint iterates* are $f^0 \triangleq a$, $\forall n \in \mathbb{N} : f^{n+1} = f(f^n)$, $f^\omega \triangleq \bigsqcup_{n\in\mathbb{N}} f^n$ which is $\mathrm{lfp}^{\sqsubseteq}_{a} f$ when $a \sqsubseteq f(a)$ is a pre-fixpoint and f is continuous[4,5,6]. If f is increasing but not continuous, transfinite iterations may have to be used [22].

[1] $f \in A \mapsto A$ is *increasing* (also *monotone*, *isotone*, ...) on a poset $\langle A, \sqsubseteq \rangle$ if and only if $\forall x, y \in A : (x \sqsubseteq y) \implies (f(x) \sqsubseteq f(y))$ [36].

[2] A *complete lattice* $\langle A, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ is a poset s. t. any subset has a least upper bound (lub) $\sqcup$, hence a greatest lower bound (glb) $\sqcap$, $\bot = \sqcup\emptyset$, $\top = \sqcap\emptyset$.

[3] A *complete partial order* (cpo) $\langle A, \sqsubseteq, \bot, \sqcup \rangle$ is a poset $\langle A, \sqsubseteq \rangle$ such that any increasing chain $C \subseteq A$ such that $\forall x, y \in C : x \sqsubseteq y \vee y \sqsubseteq x$ has a least upper bound (lub) $\sqcup C$, hence has an infimum $\bot = \sqcup\emptyset$ for the empty chain.

[4] $f \in A \mapsto A$ is *continuous* on a poset $\langle A, \sqsubseteq, \sqcup \rangle$ if and only if for all increasing chains $C \in \wp(A)$ such that its lub $\sqcup C$ does exist then the lub $\sqcup f[C]$ exists and is such that $\sqcup f[C] = f(\sqcup C)$.

[5] $\wp(X)$ or 2^X is the powerset of X i.e. the set of all subsets of a set X.

[6] The *post-image* (or *image*) of $X \in \wp(A)$ by a map $f \in A \mapsto B$ is $f[X] \triangleq \{f(x) \mid x \in X\} \in \wp(B)$.

* Work supported in part by the CMACS NSF Expeditions in Computing award 0926166.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

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

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

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

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

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

Cousot&Cousot - *An Abstract Interpretation Framework for Termination* (POPL 2012)

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

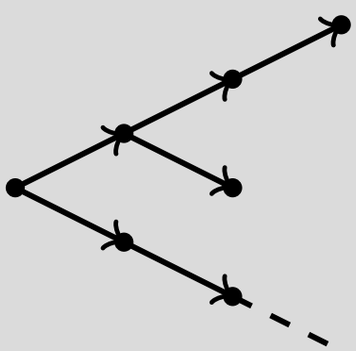
$$\mathcal{T}_t \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

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

Example



Cousot&Cousot - *An Abstract Interpretation Framework for Termination* (POPL 2012)

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

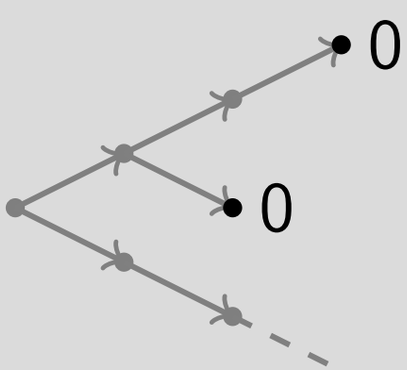
$$\mathcal{T}_t \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

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

Example



Cousot&Cousot - *An Abstract Interpretation Framework for Termination* (POPL 2012)

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

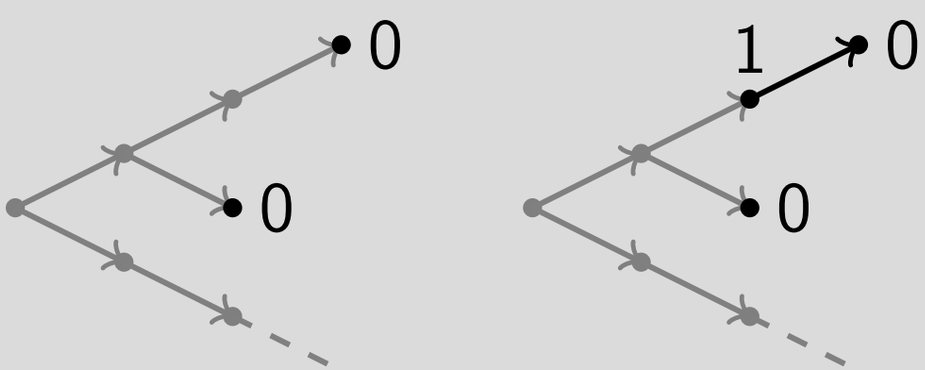
$$\mathcal{T}_t \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

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

Example



Cousot&Cousot - *An Abstract Interpretation Framework for Termination* (POPL 2012)

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

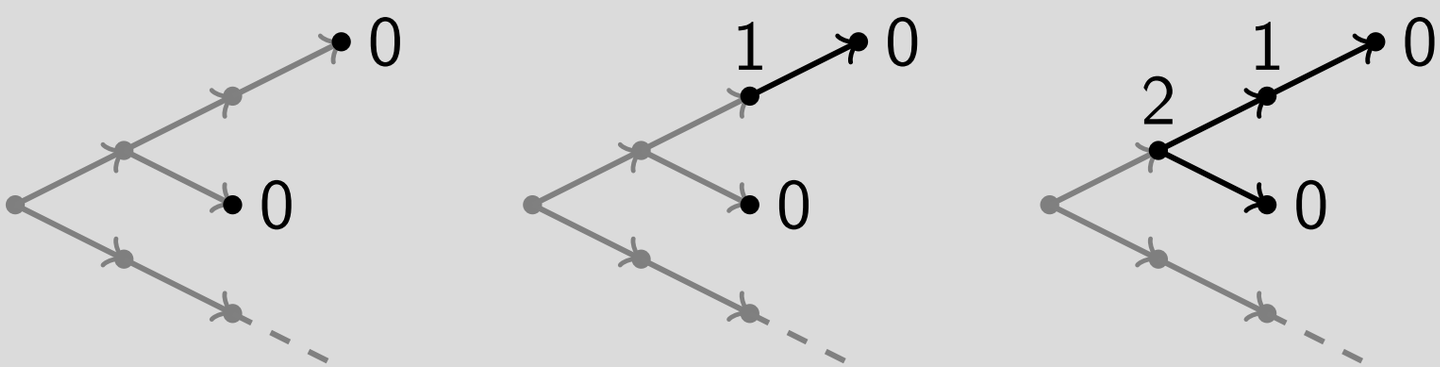
$$\mathcal{T}_t \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

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

Example



Cousot&Cousot - *An Abstract Interpretation Framework for Termination* (POPL 2012)

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

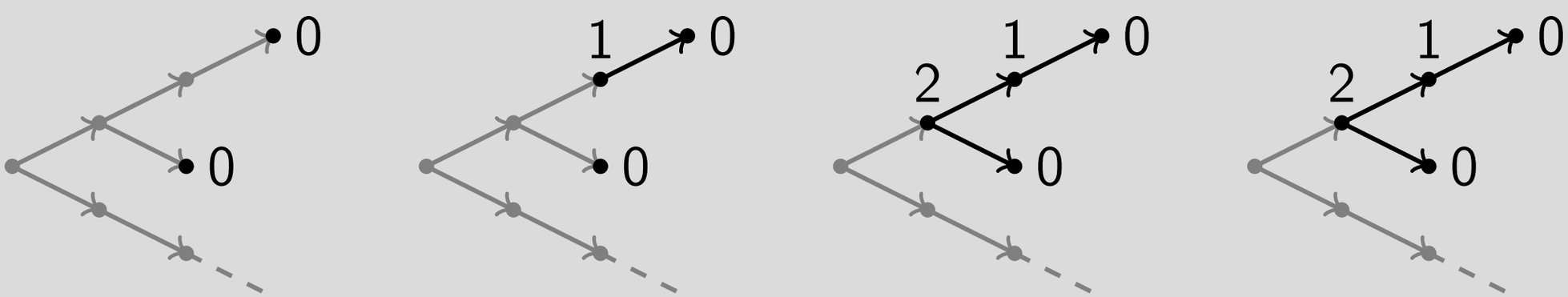
$$\mathcal{T}_t \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

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

Example



Cousot&Cousot - *An Abstract Interpretation Framework for Termination* (POPL 2012)

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

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

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

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

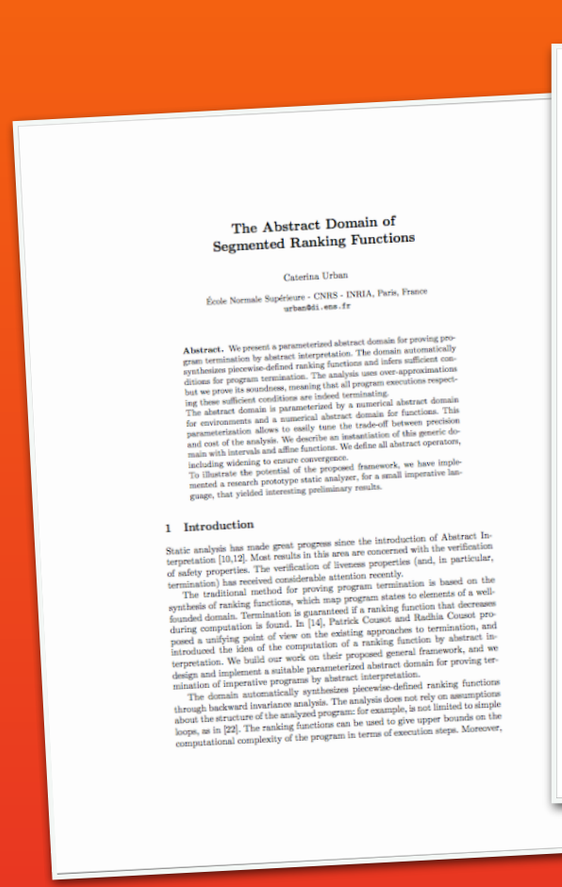
$$F_t(v)s \stackrel{\text{def}}{=} \begin{cases} \mathbf{0} & s \in \beta \\ \sup\{\ v(s') + 1 \mid \langle s, s' \rangle \in \tau\ \} & s \in \widetilde{\mathbf{pre}}(\mathbf{dom}(v)) \\ \mathbf{undefined} & \text{otherwise} \end{cases}$$

Theorem (Soundness and Completeness)

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

Cousot&Cousot - *An Abstract Interpretation Framework for Termination* (POPL 2012)

Piecewise-Defined Ranking Functions



The Abstract Domain of
Segmented Ranking Functions

Caterina Urban

École Normale Supérieure - CNRS - INRIA, Paris, France
urban@di.ens.fr

Abstract. We present a parameterized abstract domain for proving program termination by abstract interpretation. The domain automatically synthesizes piecewise-defined ranking functions and infers sufficient conditions for program termination. The analysis uses over-approximations but we prove its soundness, meaning that all program executions respecting these sufficient conditions are indeed terminating.
The abstract domain is parameterized by a numerical abstract domain for environments and a numerical abstract domain for functions. This parameterization allows to easily tune the trade-off between precision and cost of the analysis. We describe an instantiation of this generic domain with intervals and affine functions. We define all abstract operators, including widening to ensure convergence.
To illustrate the potential of the proposed framework, we have implemented a research prototype static analyzer, for a small imperative language, that yielded interesting preliminary results.

1 Introduction

Static analysis has made great progress since the introduction of Abstract Interpretation [10,12]. Most results in this area are concerned with the verification of safety properties. The verification of liveness properties (and, in particular, termination) has received considerable attention recently.
The traditional method for proving program termination is based on the synthesis of ranking functions, which map program states to elements of a well-founded domain. Termination is guaranteed if a ranking function that decreases during computation is found. In [14], Patrick Cousot and Radhia Cousot proposed a unifying point of view on the existing approaches to termination, and introduced the idea of the computation of a ranking function by abstract interpretation. We build our work on their proposed general framework, and we design and implement a suitable parameterized abstract domain for proving termination of imperative programs by abstract interpretation.
The domain automatically synthesizes piecewise-defined ranking functions through backward invariance analysis. The analysis does not rely on assumptions about the structure of the analyzed program: for example, is not limited to simple loops, as in [22]. The ranking functions can be used to give upper bounds on the computational complexity of the program in terms of execution steps. Moreover,

An Abstract Domain to Infer
Ordinal-Valued Ranking Functions*

Caterina Urban and Antoine Miné

ÉNS & CNRS & INRIA, Paris, France
urban@di.ens.fr, mine@di.ens.fr

Abstract. The traditional method for proving program termination consists in inferring a ranking function. In many cases (i.e. programs with unbounded non-determinism), a single ranking function over natural numbers is not sufficient. Hence, we propose a new abstract domain to automatically infer ranking functions over ordinals.
We extend an existing domain for piecewise-defined natural-valued ranking functions to polynomials in ω, where the polynomial coefficients are natural-valued functions of the program variables. The abstract domain is parametric in the choice of the maximum degree of the polynomial, and the types of functions used as polynomial coefficients.
We have implemented a prototype static analyzer for a while-language by instantiating our domain using affine functions as polynomial coefficients. We successfully analyzed small but intricate examples that are out of the reach of existing methods.
To our knowledge this is the first abstract domain able to reason about ordinals. Handling ordinals leads to a powerful approach for proving termination of imperative programs, which in particular subsumes existing techniques based on lexicographic ranking functions.

1 Introduction

The traditional method for proving program termination [12] consists in inferring ranking functions, namely mappings from program states to elements of a well-ordered set (e.g. ordinals) whose value decreases during program execution.
Intuitively, we can define a partial ranking function from the states of a program to ordinal numbers in an incremental way: we start from the program final states, where the function has value 0 (and is undefined elsewhere); then, we add states to the domain of the function, retracing the program backwards and counting the maximum number of performed program steps as value of the function. In [10], this intuition is formalized into a most precise ranking function that can be expressed in fixpoint form by abstract interpretation [8] of the program maximal trace semantics.

* The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 269335 (ARTEMIS project MBAT) (see Article II.9. of the JU Grant Agreement)

A Decision Tree Abstract Domain
for Proving Conditional Termination*

Caterina Urban and Antoine Miné

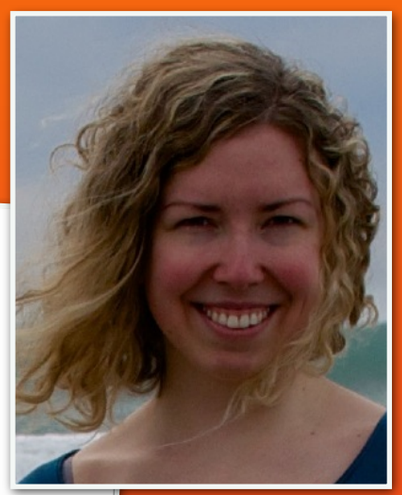
ÉNS & CNRS & INRIA, France
{urban,mine}@di.ens.fr

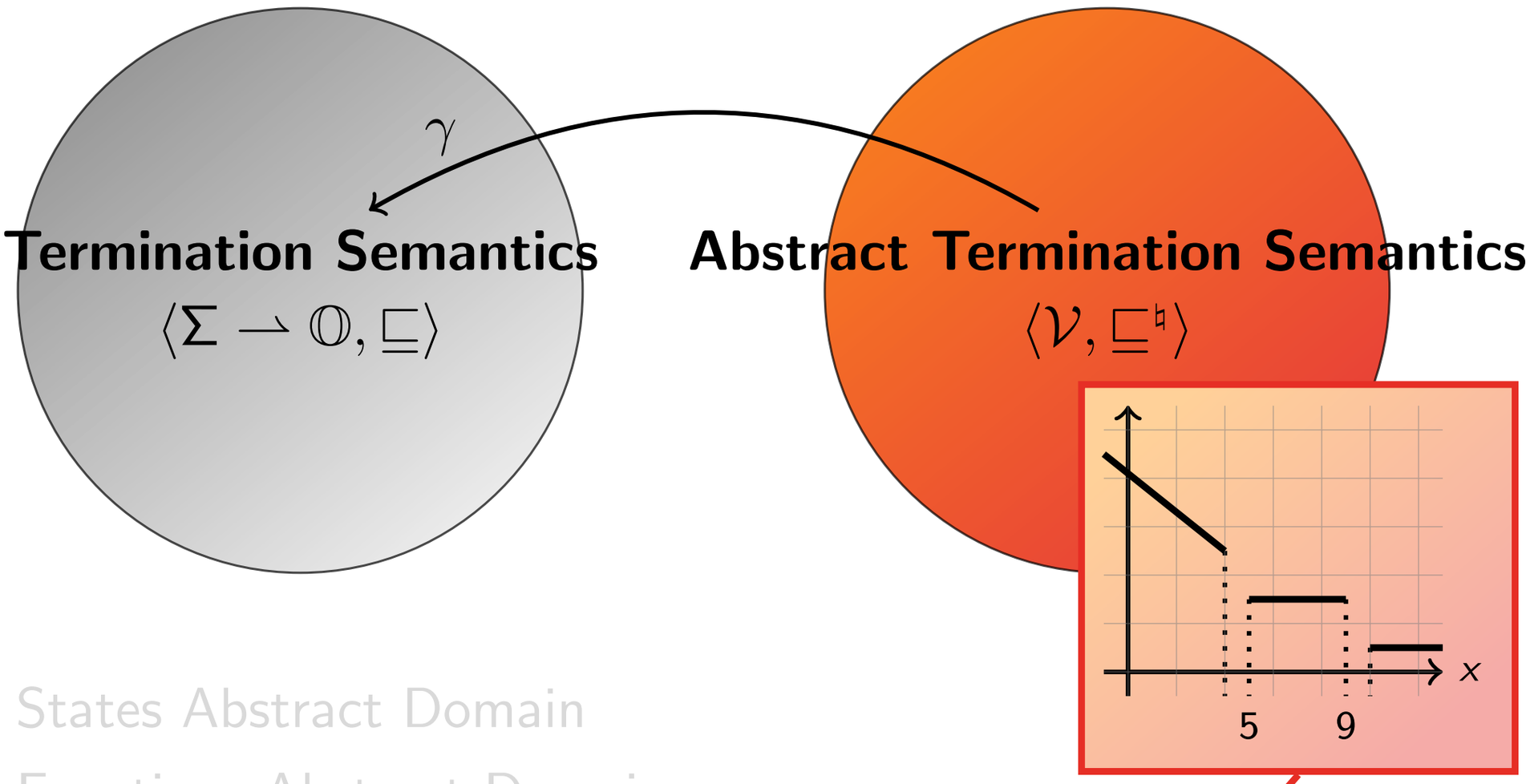
Abstract. We present a new parameterized abstract domain able to refine existing numerical abstract domains with finite disjunctions. The elements of the abstract domain are decision trees where the decision nodes are labeled with linear constraints, and the leaf nodes belong to a numerical abstract domain.
The abstract domain is parametric in the choice between the expressivity and the cost of the linear constraints for the decision nodes (e.g., polyhedral or octagonal constraints), and the choice of the abstract domain for the leaf nodes. We describe an instance of this domain based on piecewise-defined ranking functions for the automatic inference of sufficient preconditions for program termination.
We have implemented a static analyzer for proving conditional termination of programs written in (a subset of) C and, using experimental evidence, we show that it performs well on a wide variety of benchmarks, it is competitive with the state of the art and is able to analyze programs that are out of the reach of existing methods.

1 Introduction

Numerical abstract domains are widely used in static program analysis and verification to maintain information about the set of possible values of the program variables along with the possible numerical relationships between them. The most common abstract domains — intervals [10], octagons [27] and convex polyhedra [14] — maintain this information using convex sets consisting of conjunctions of linear constraints. The convexity of these abstract domains makes the analysis scalable. On the other hand, it might lead to too harsh approximations and imprecisions in the analysis, ultimately yielding false alarms and a failure of the analyzer to prove the desired program property.
The key for an adequate cost versus precision trade-off is the handling of disjunctions arising during the analysis (e.g., from program tests and loops). In practice, numerical abstract domains are usually refined by adding weak forms of disjunctions to increase the expressivity while minimizing the cost of the analysis [13, 18, 20, 29, etc.].
In this paper, we propose a novel parameterized abstract domain for the disjunctive refinement of numerical abstract domains which is particularly well-suited for proving conditional termination of imperative programs.
The elements of the abstract domain are inspired by the space partitioning trees [16] developed in the context of 3D computer graphics and the use of decision trees

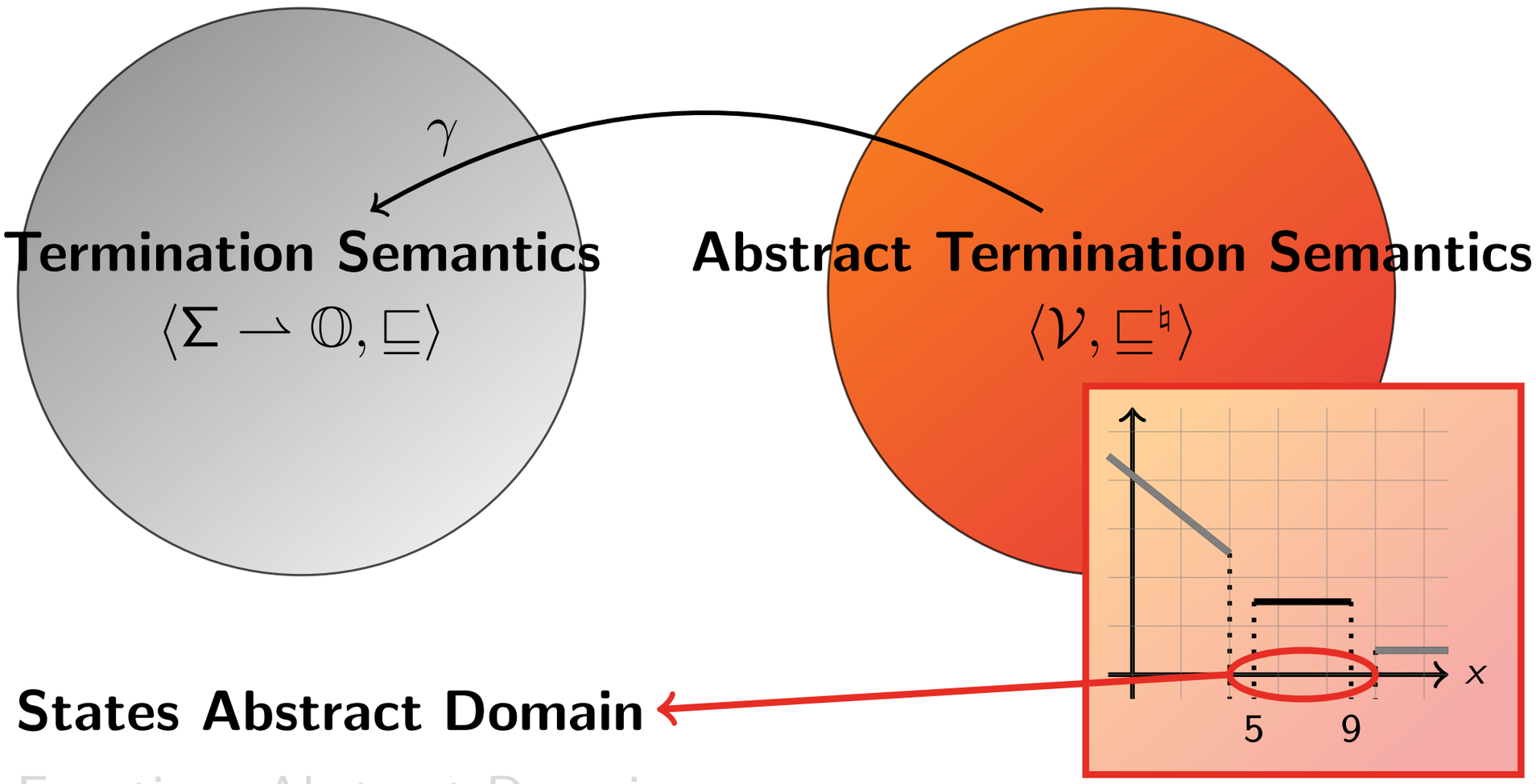
* The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 269335 (ARTEMIS project MBAT) (see Article II.9. of the JU Grant Agreement)





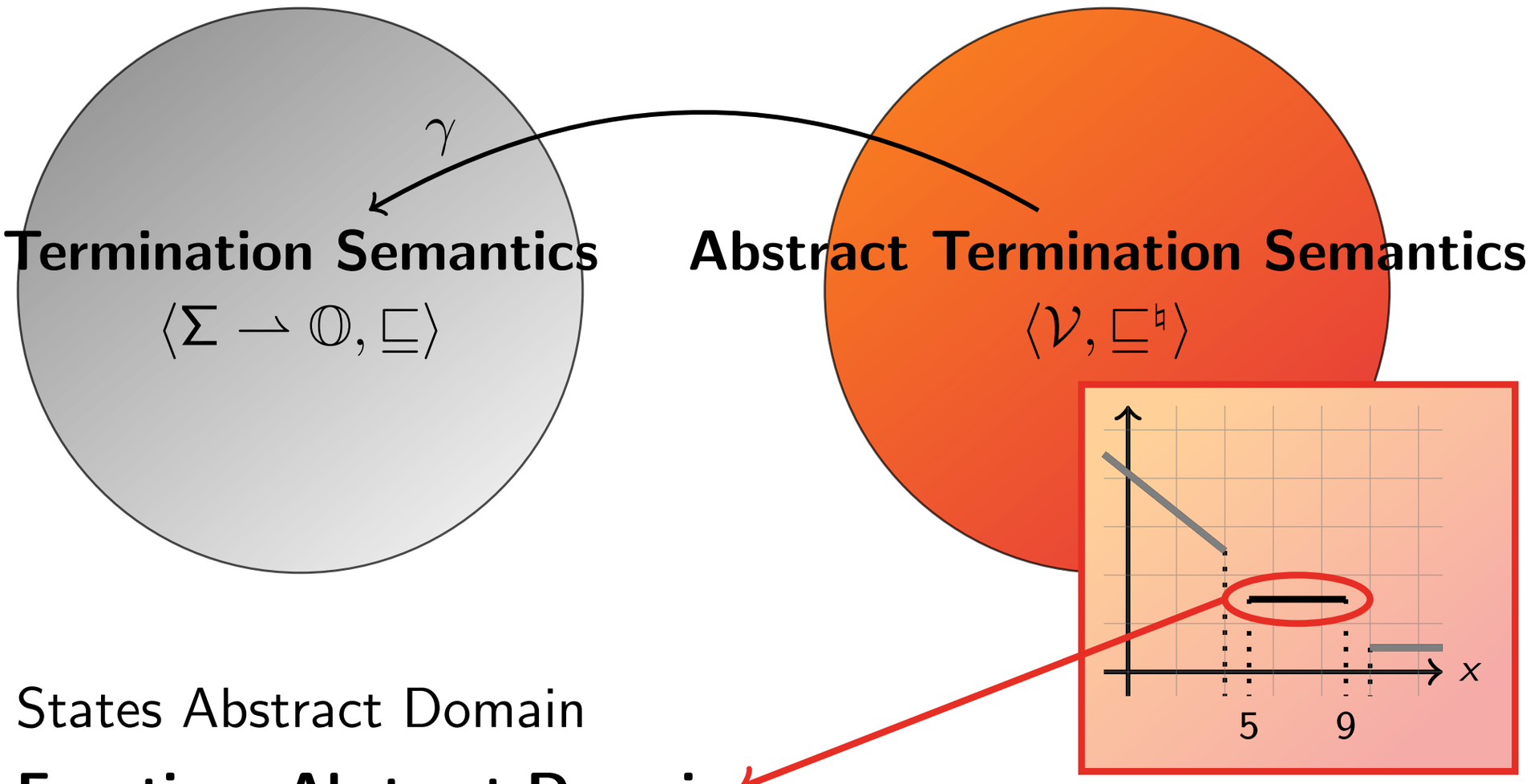
- States Abstract Domain S
- Functions Abstract Domain F
- **Piecewise-Defined Ranking Functions Abstract Domain** V(S, F)

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



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

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



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

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

Example

int : x

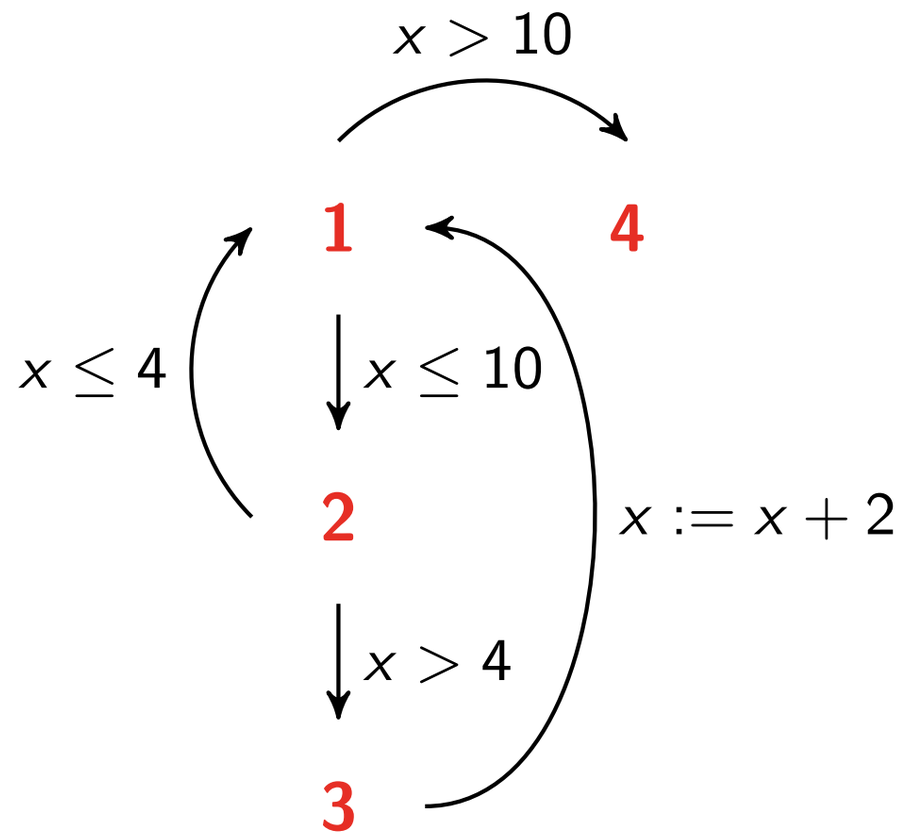
while $^{1}(x \leq 10)$ do

  if $^{2}(x > 4)$ then

    $^{3}x := x + 2$

  fi

od^{4}



Example

int : x

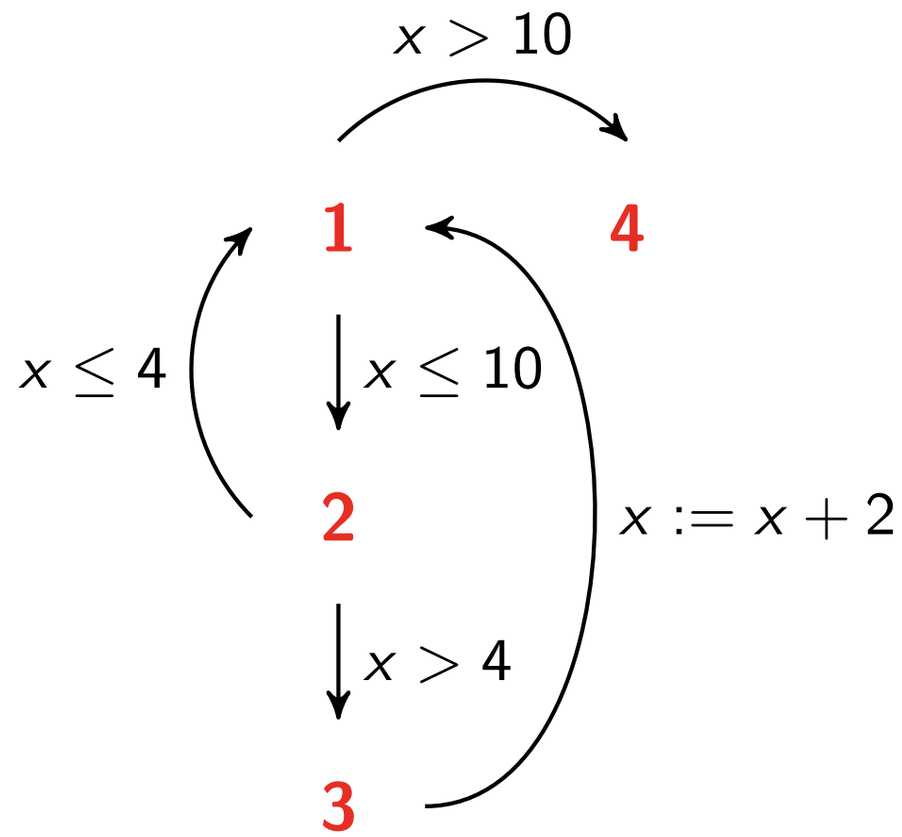
while $^{1}(x \leq 10)$ do

  if $^{2}(x > 4)$ then

    $^{3}x := x + 2$

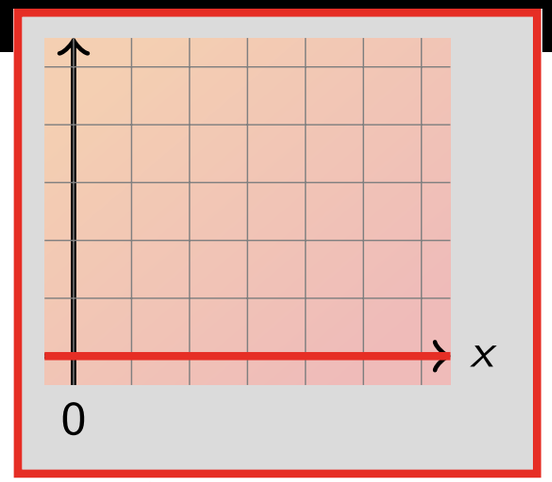
  fi

od^{4}



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

we start at the end
with 0 steps
to termination



Example

int : x

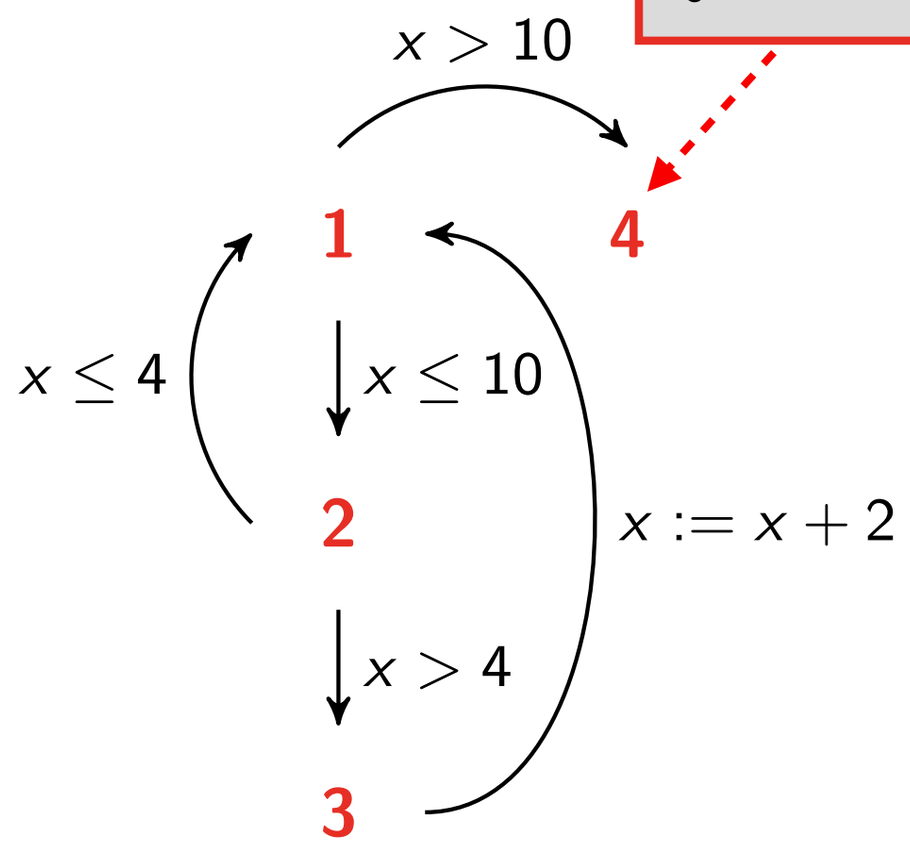
while $^{1}(x \leq 10)$ do

  if $^{2}(x > 4)$ then

    $^{3}x := x + 2$

  fi

od^{4}



Example

int : x

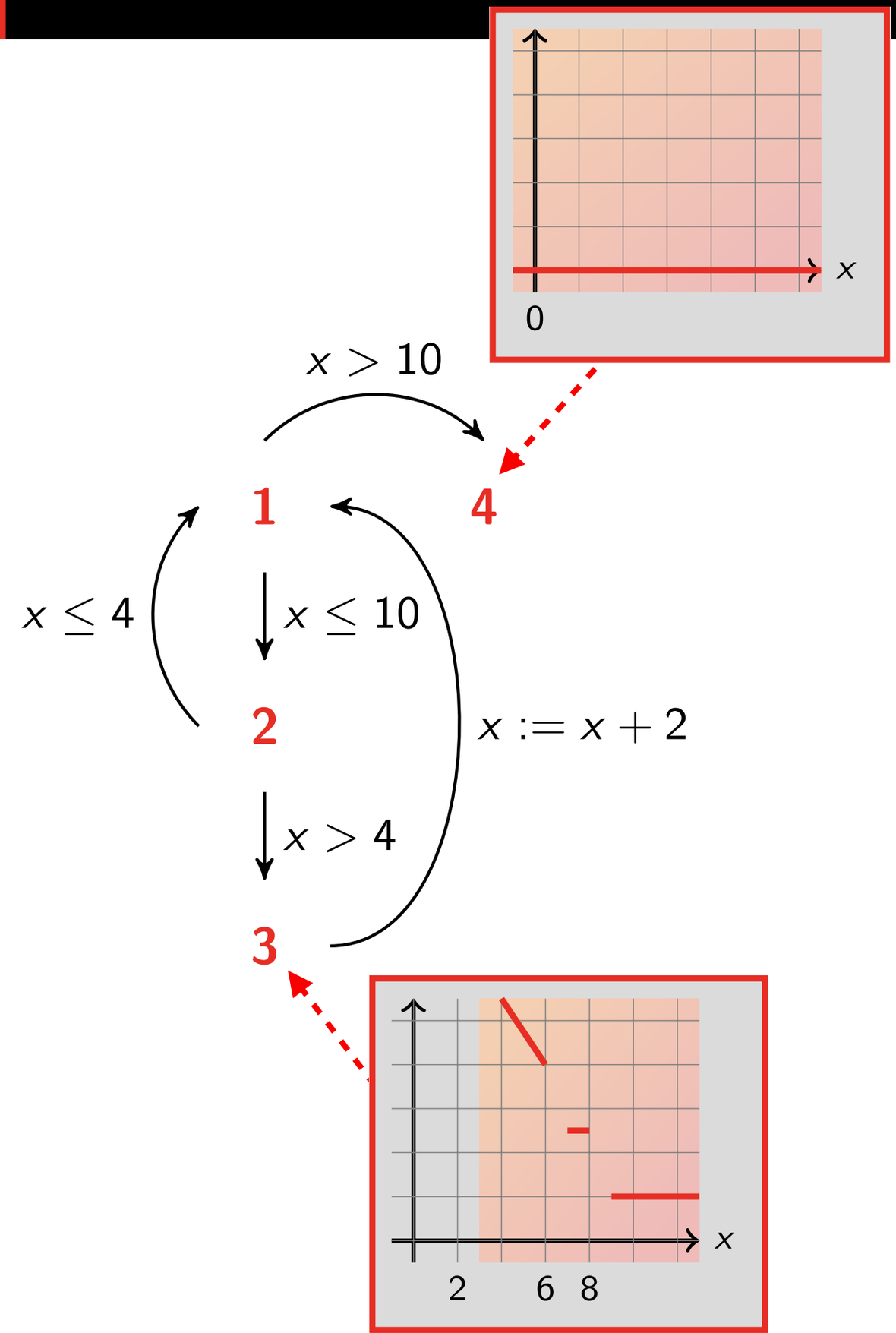
while $^{1}(x \leq 10)$ do

  if $^{2}(x > 4)$ then

    $^{3}x := x + 2$

  fi

od^{4}



Example

int : x

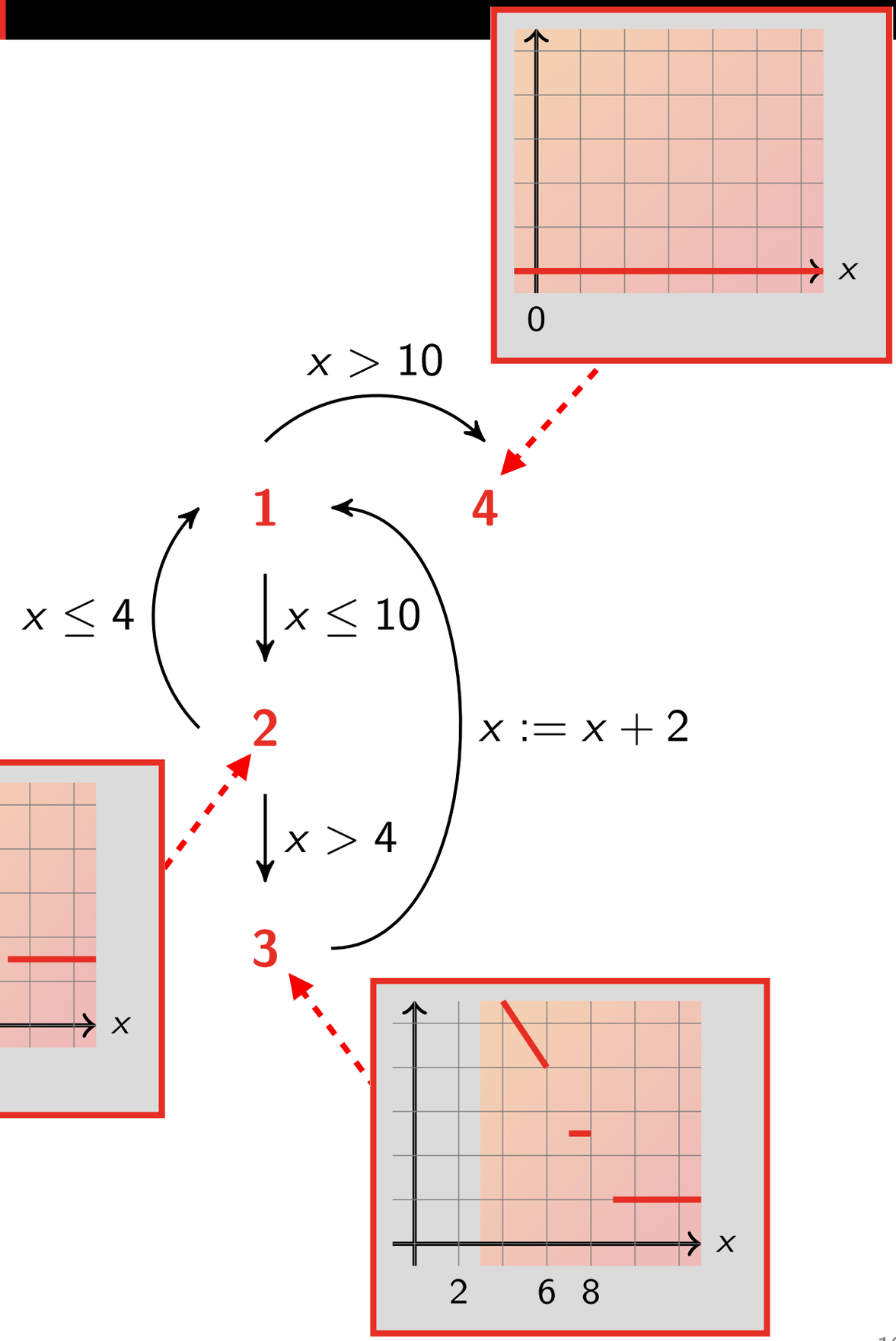
while $^{1}(x \leq 10)$ do

  if $^{2}(x > 4)$ then

    $^{3}x := x + 2$

  fi

od^{4}



the analysis gives $x \geq 5$
as **sufficient precondition**

Example

int : x

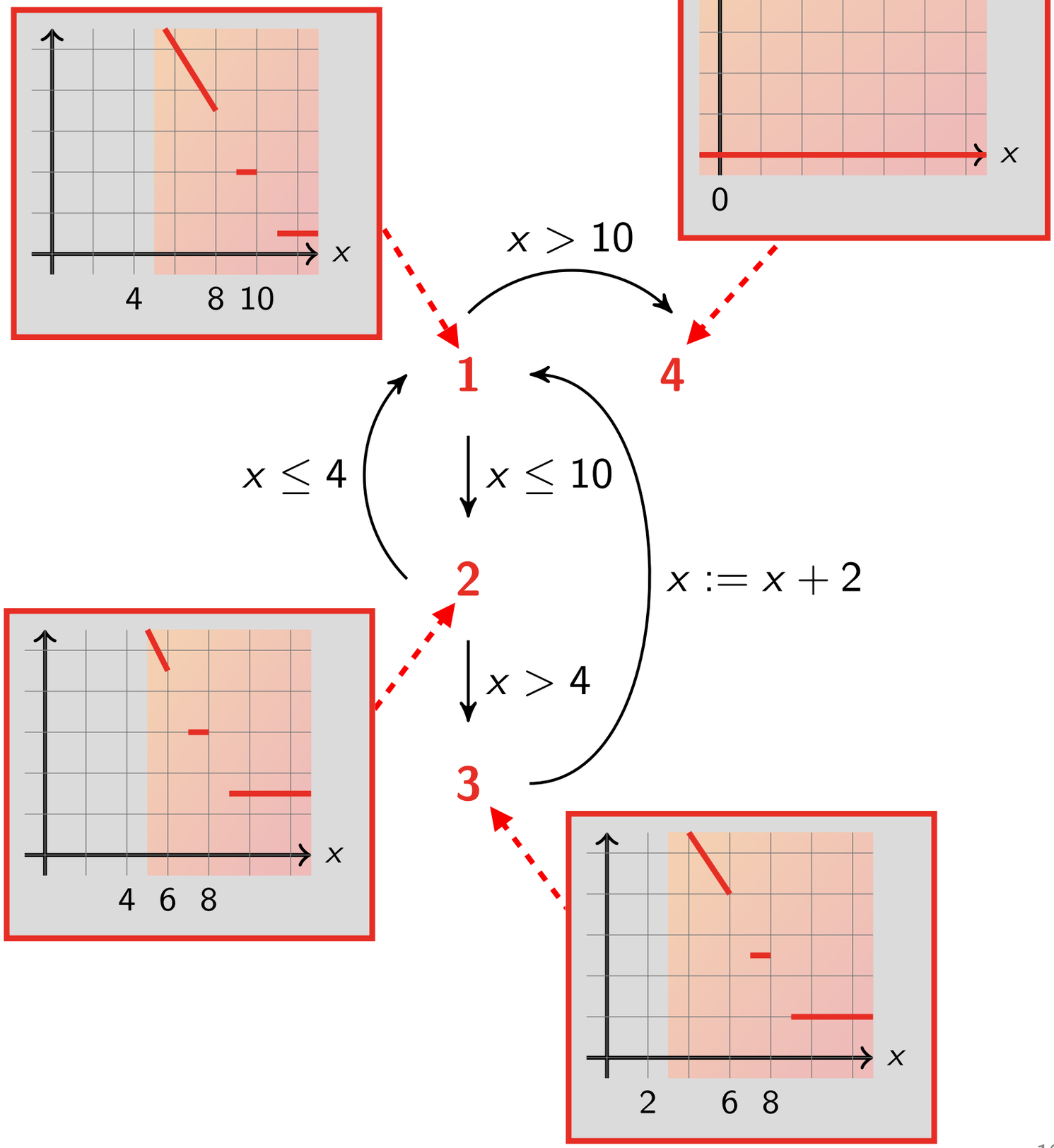
while $^{1}(x \leq 10)$ do

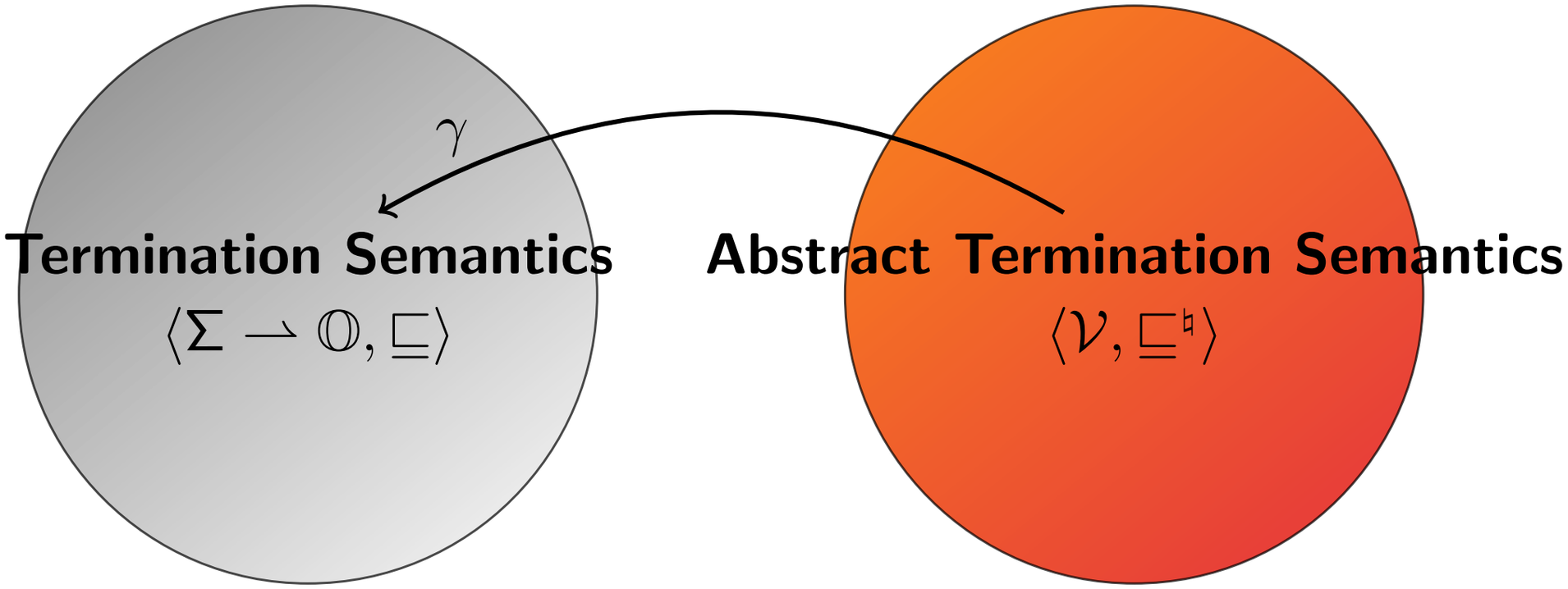
  if $^{2}(x > 4)$ then

    $^{3}x := x + 2$

  fi

od^{4}





Theorem (Soundness)

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

Guarantee and Recurrence Properties

Example

int : x, y

while [1] $(x \geq 0)$ do

  [2] $x := x + 1$

od

while [3] (true) do

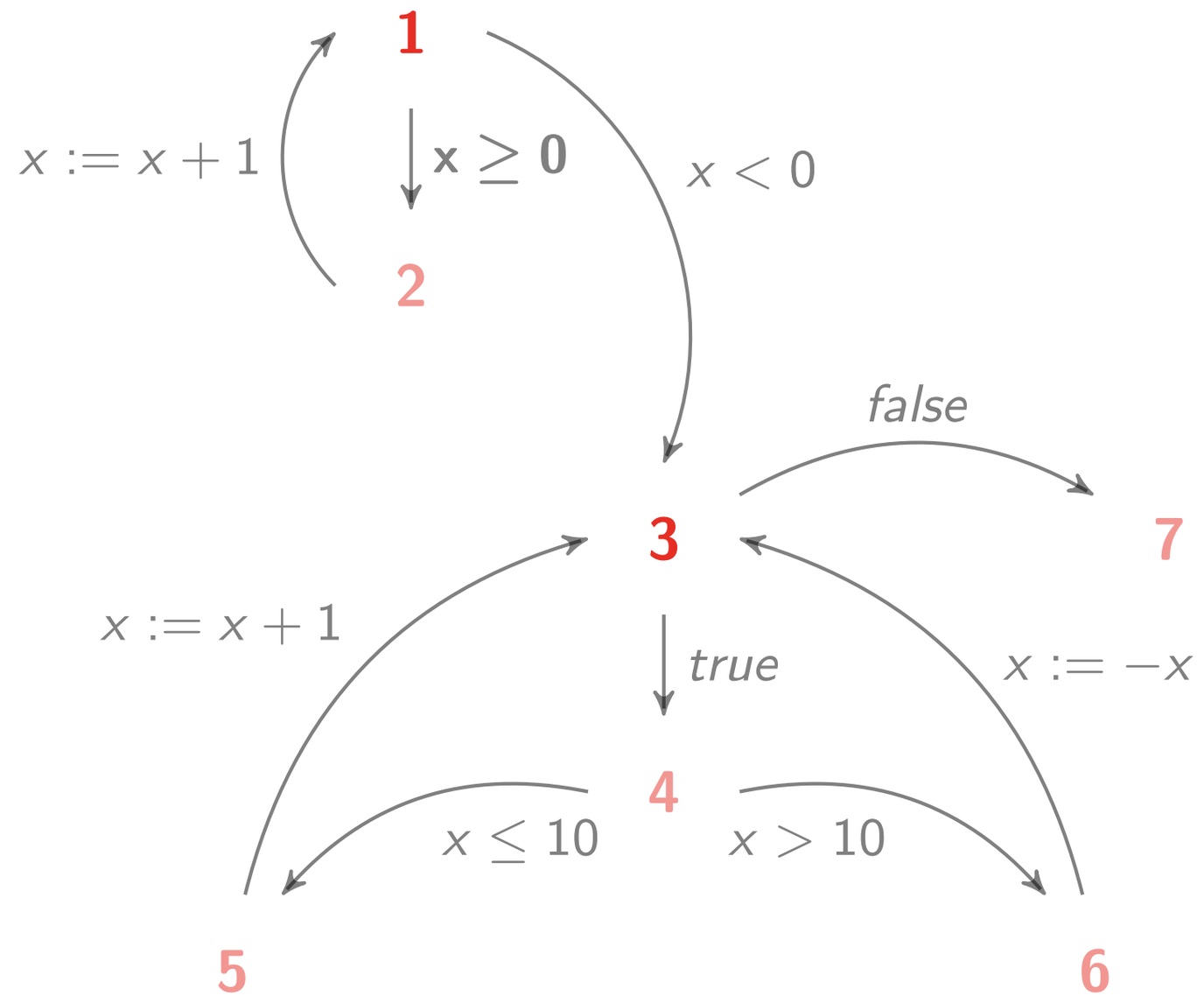
  if [4] ($x \leq 10$)

    [5] $x := x + 1$

  else

    [6] $x := -x$

od[7]



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] $x := x + 1$

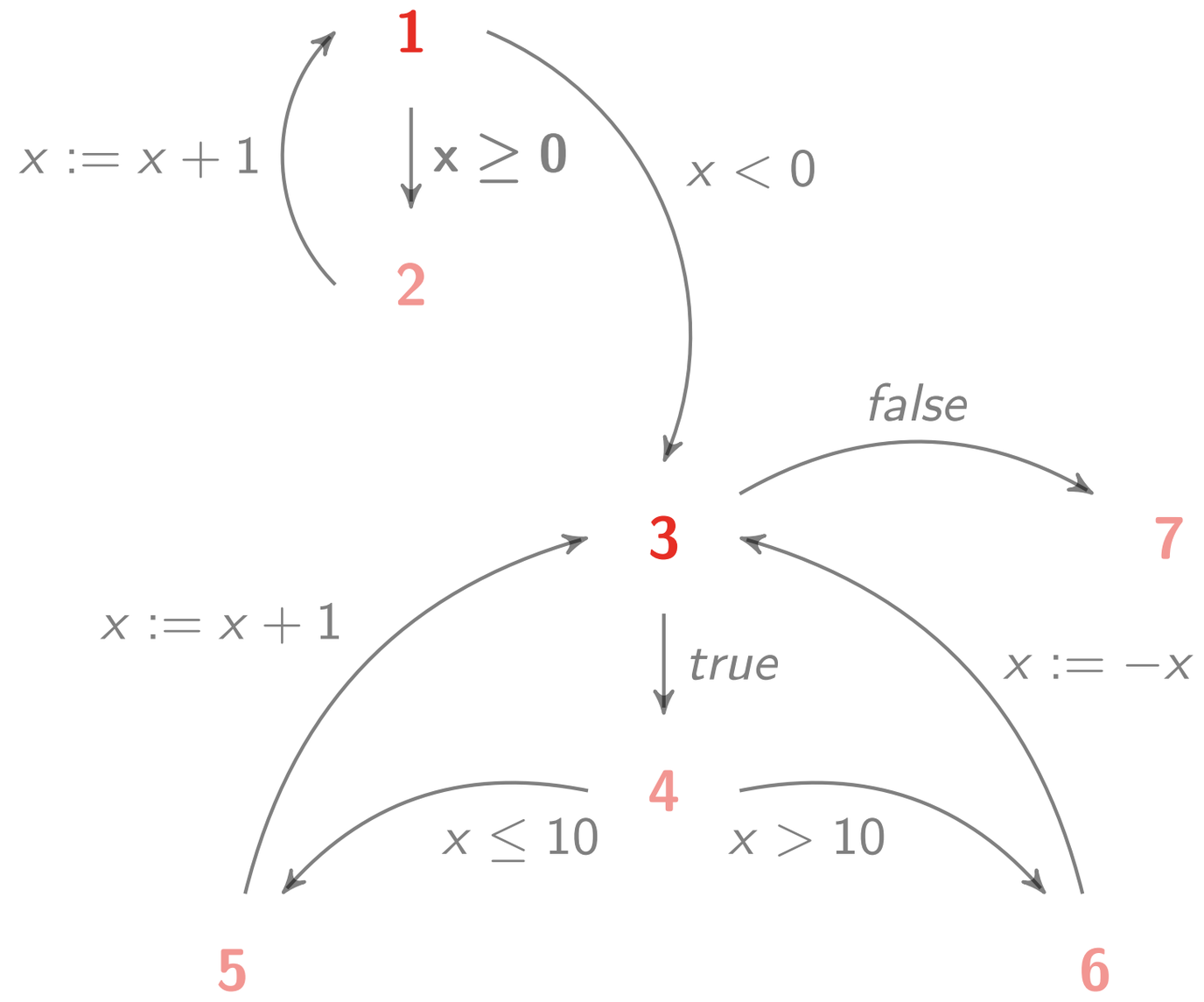
  else

    [6] $x := -x$

od [7]

Guarantee Property

$\Diamond\ x = 3$



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] $x := x + 1$

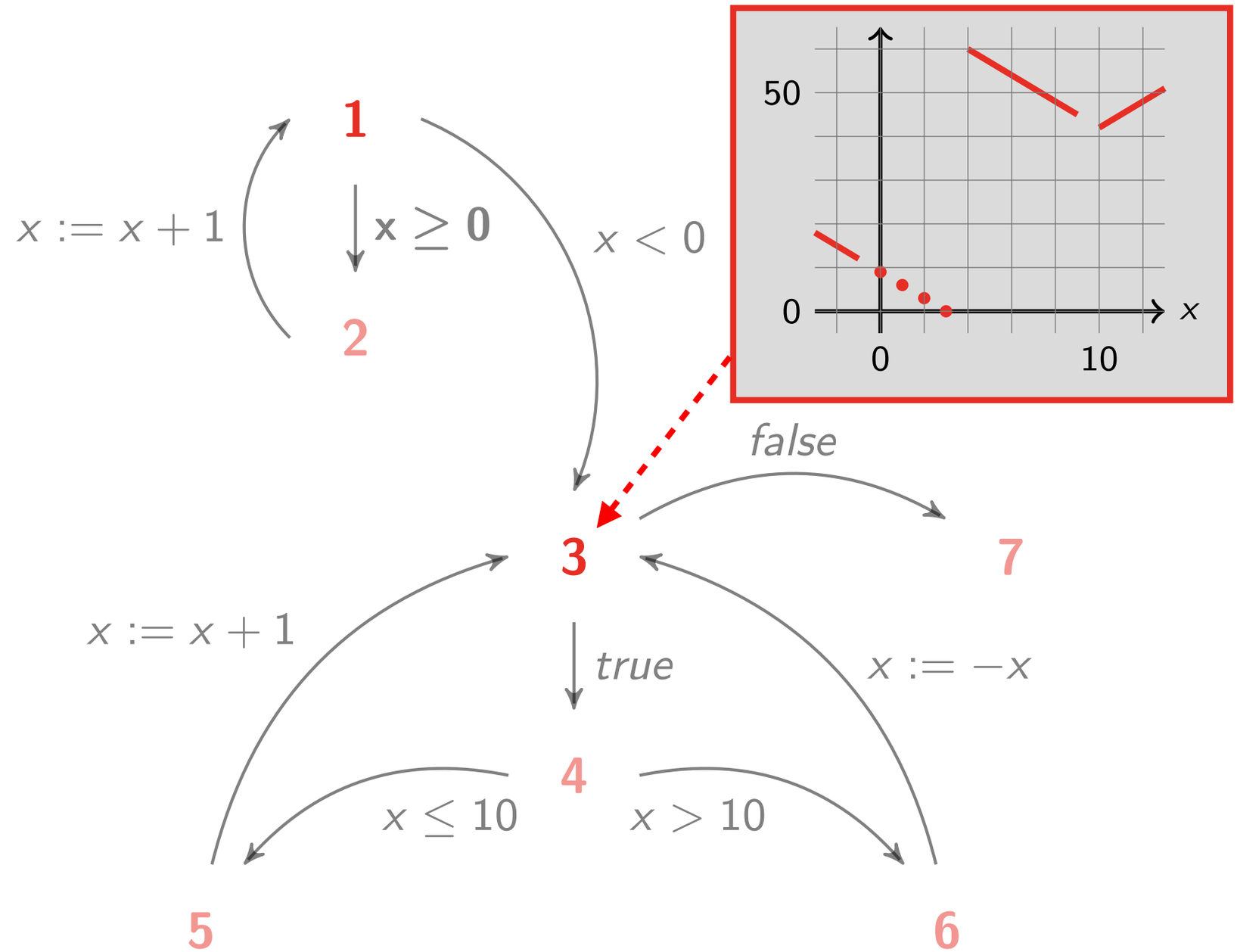
  else

    [6] $x := -x$

od [7]

Guarantee Property

$\Diamond\ x = 3$



Example

int : x, y

while [1]$(x \geq 0)$

  [2]$x := x + 1$

od

while [3](true) do

  if [4]($x \leq 10$)

    [5]$x := x + 1$

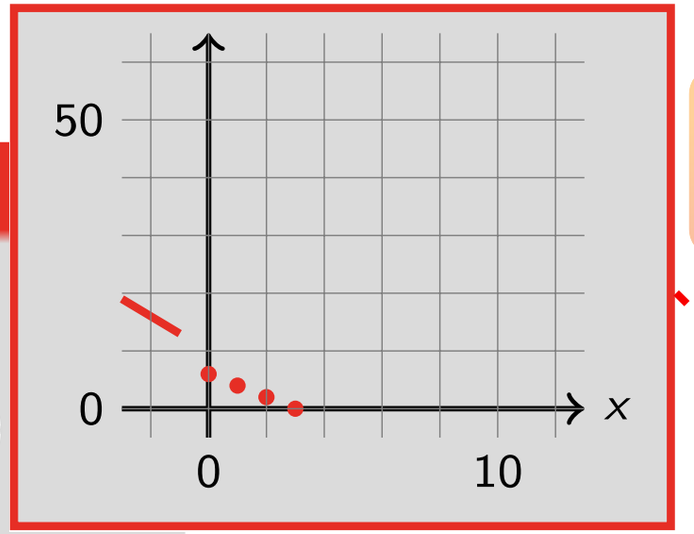
  else

    [6]$x := -x$

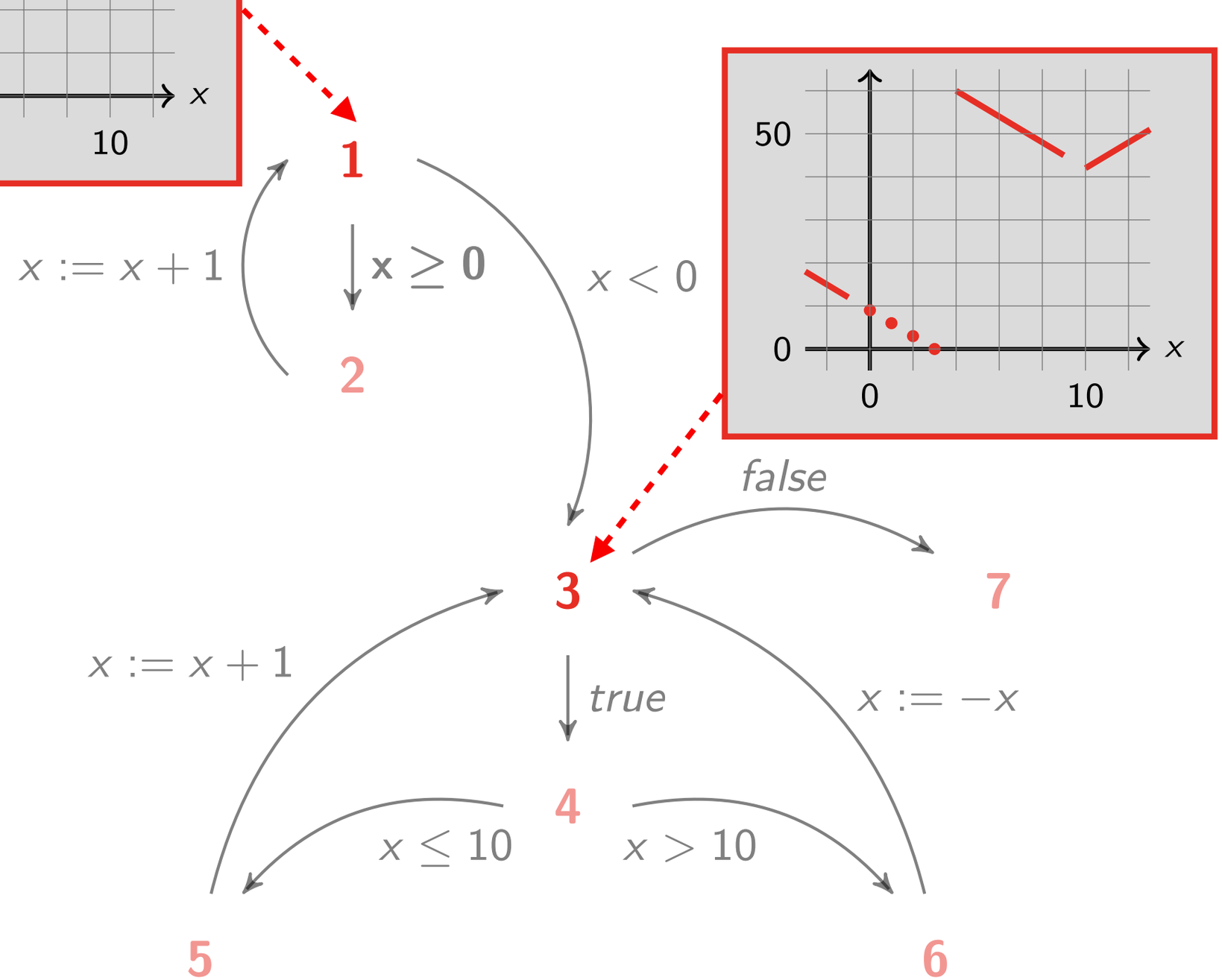
od[7]

Guarantee Property

$\Diamond\ x = 3$



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]$x := x + 1$

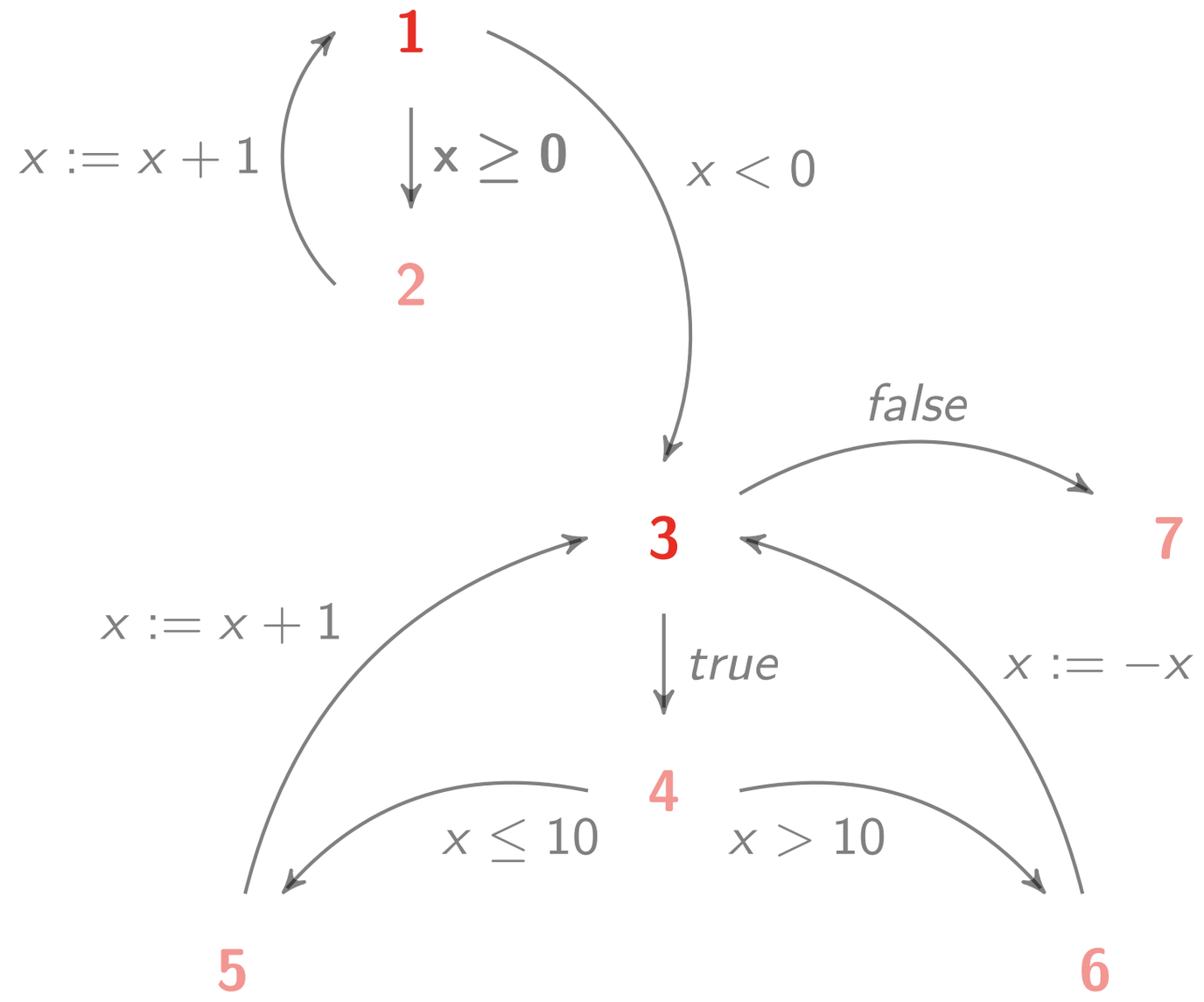
  else

    [6]$x := -x$

od[7]

Recurrence Property

$\Box\Diamond\ x = 3$



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] $x := x + 1$

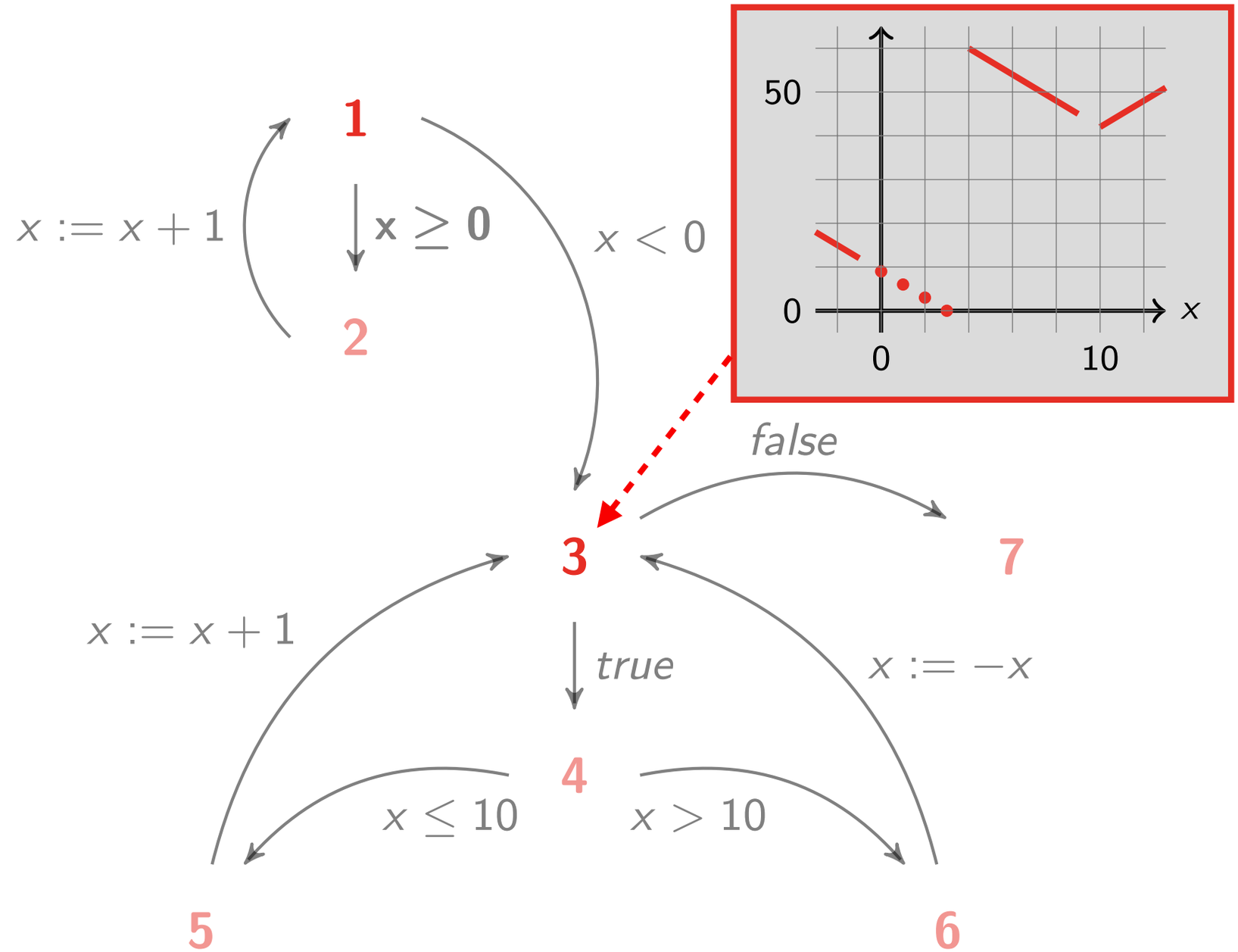
  else

    [6] $x := -x$

od [7]

Recurrence Property

$\square\Diamond\ x = 3$



Example

int : x, y

while [1] $(x \geq 0)$

  [2] $x := x + 1$

od

while [3] (true) do

  if [4] ($x \leq 10$)

    [5] $x := x + 1$

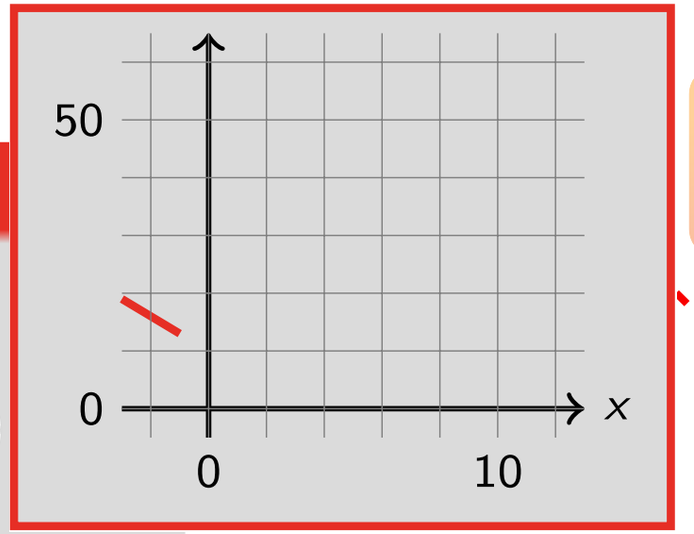
  else

    [6] $x := -x$

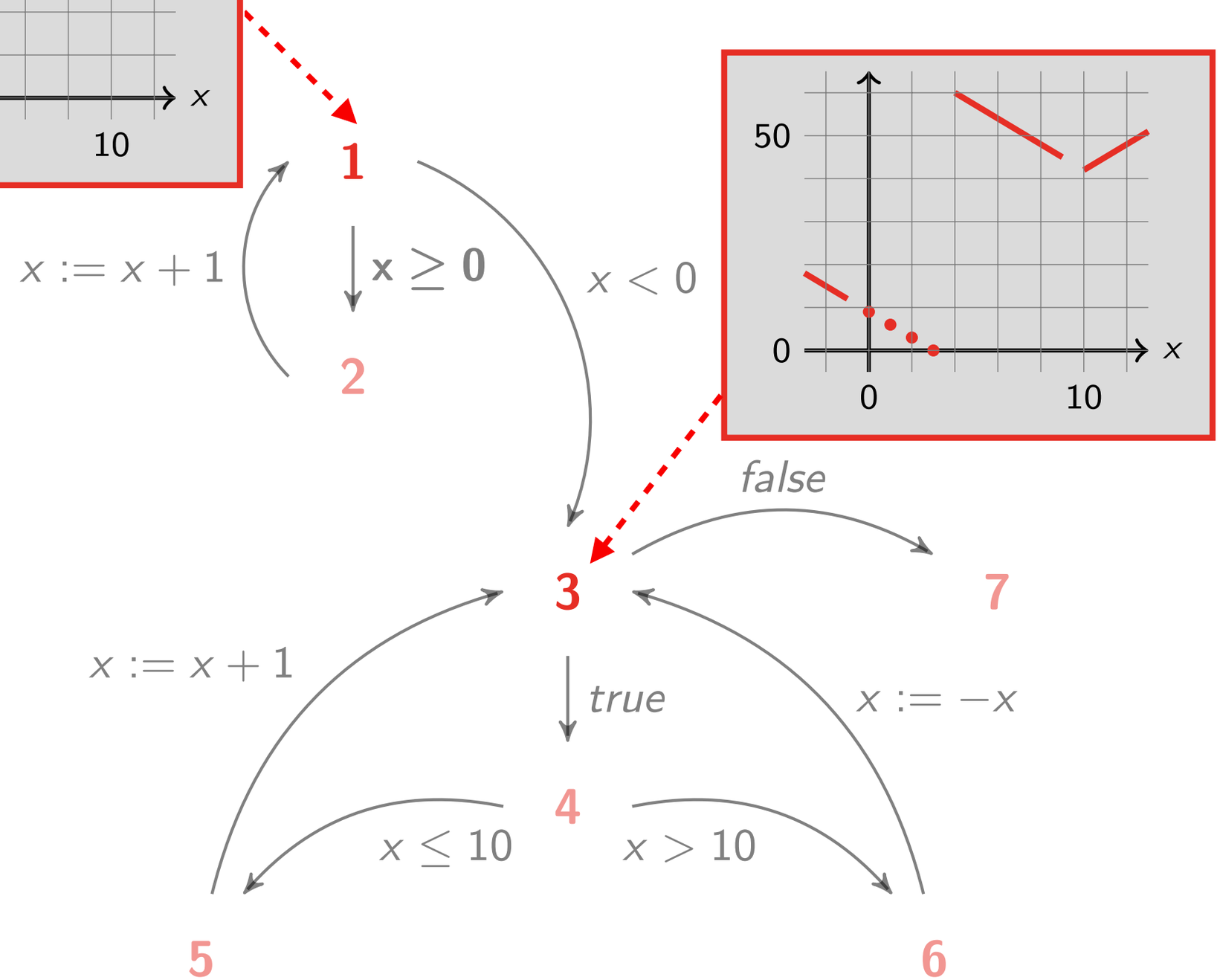
od [7]

Recurrence Property

$\square\Diamond\ x = 3$



the analysis gives $x < 0$ as
sufficient precondition



Guarantee Properties

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\mathcal{T}^{\varphi}_{g} \stackrel{\text{def}}{=} \mathbf{lfp}\ F^{\varphi}_{g}$$

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

$$F^{\varphi}_{g}(v)s \stackrel{\text{def}}{=} \begin{cases} 0 & s \in \varphi \\ \sup\{\ v(s') + 1 \mid \langle s, s' \rangle \in \tau\ \} & s \in \widetilde{\mathbf{pre}}(\mathbf{dom}(v)) \wedge s \notin \varphi \\ \mathbf{undefined} & \text{otherwise} \end{cases}$$

program $\mapsto$ maximal trace semantics $\to$ **φ-guarantee semantics**

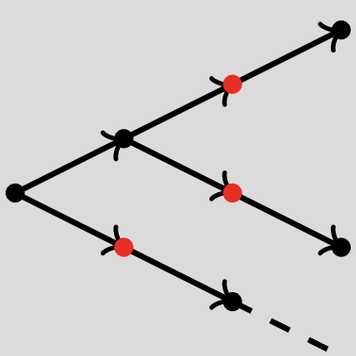
$$\mathcal{T}_{\boldsymbol{g}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$$

$$\mathcal{T}_{\boldsymbol{g}}^{\varphi} \stackrel{\text{def}}{=} \mathbf{lfp}\ \boldsymbol{F}_{\boldsymbol{g}}^{\varphi}$$

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

$$\boldsymbol{F}_{\boldsymbol{g}}^{\varphi}(\boldsymbol{v})\boldsymbol{s} \stackrel{\text{def}}{=} \begin{cases} \mathbf{0} & \boldsymbol{s} \in \varphi \\ \mathbf{sup}\{\ \boldsymbol{v}(\boldsymbol{s}') + \mathbf{1} \mid \langle \boldsymbol{s}, \boldsymbol{s}' \rangle \in \tau\ \} & \boldsymbol{s} \in \widetilde{\mathbf{pre}}(\mathbf{dom}(\boldsymbol{v})) \ \wedge\ \boldsymbol{s} \notin \varphi \\ \mathbf{undefined} & \mathbf{otherwise} \end{cases}$$

Example



program $\mapsto$ maximal trace semantics $\to$ **φ-guarantee semantics**

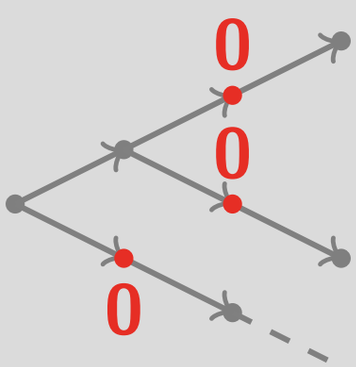
$$\mathcal{T}_g^\varphi \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

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

Example



program $\mapsto$ maximal trace semantics $\to$ **φ-guarantee semantics**

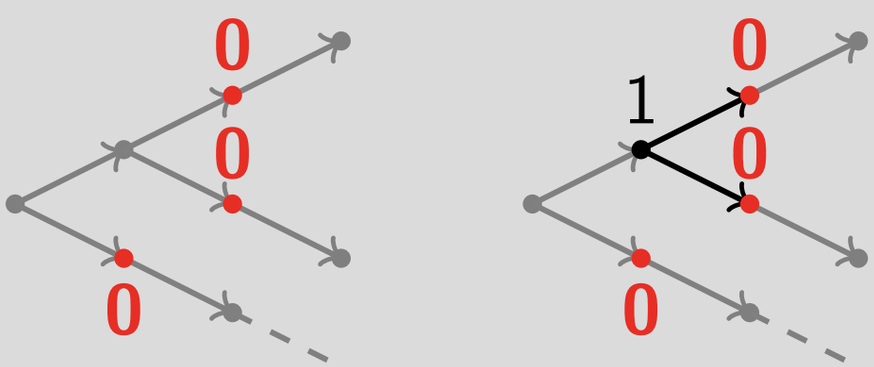
$$\mathcal{T}_g^\varphi \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

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

Example



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

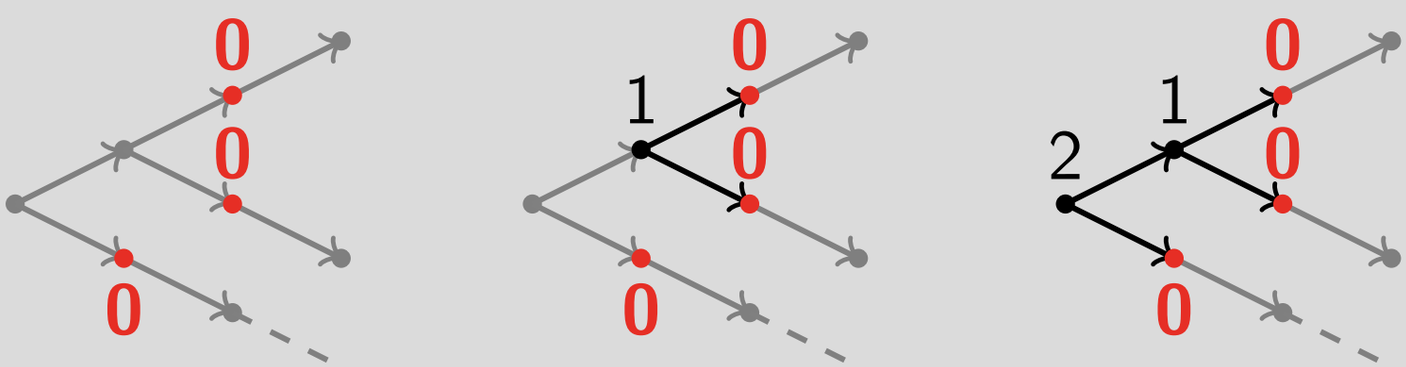
$$\mathcal{T}_g^\varphi \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

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

Example



program $\mapsto$ maximal trace semantics $\to$ **φ-guarantee semantics**

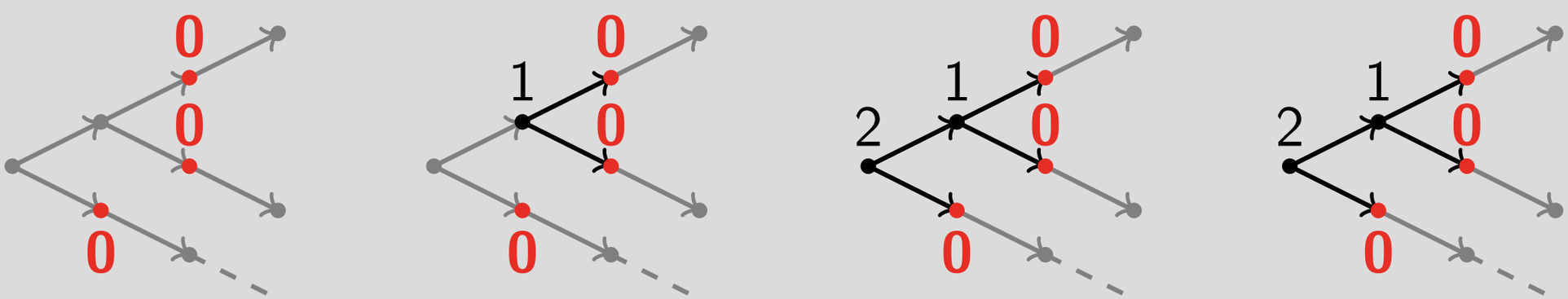
$$\mathcal{T}_g^\varphi \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

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

Example



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

$$\mathcal{T}_{\boldsymbol{g}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$$

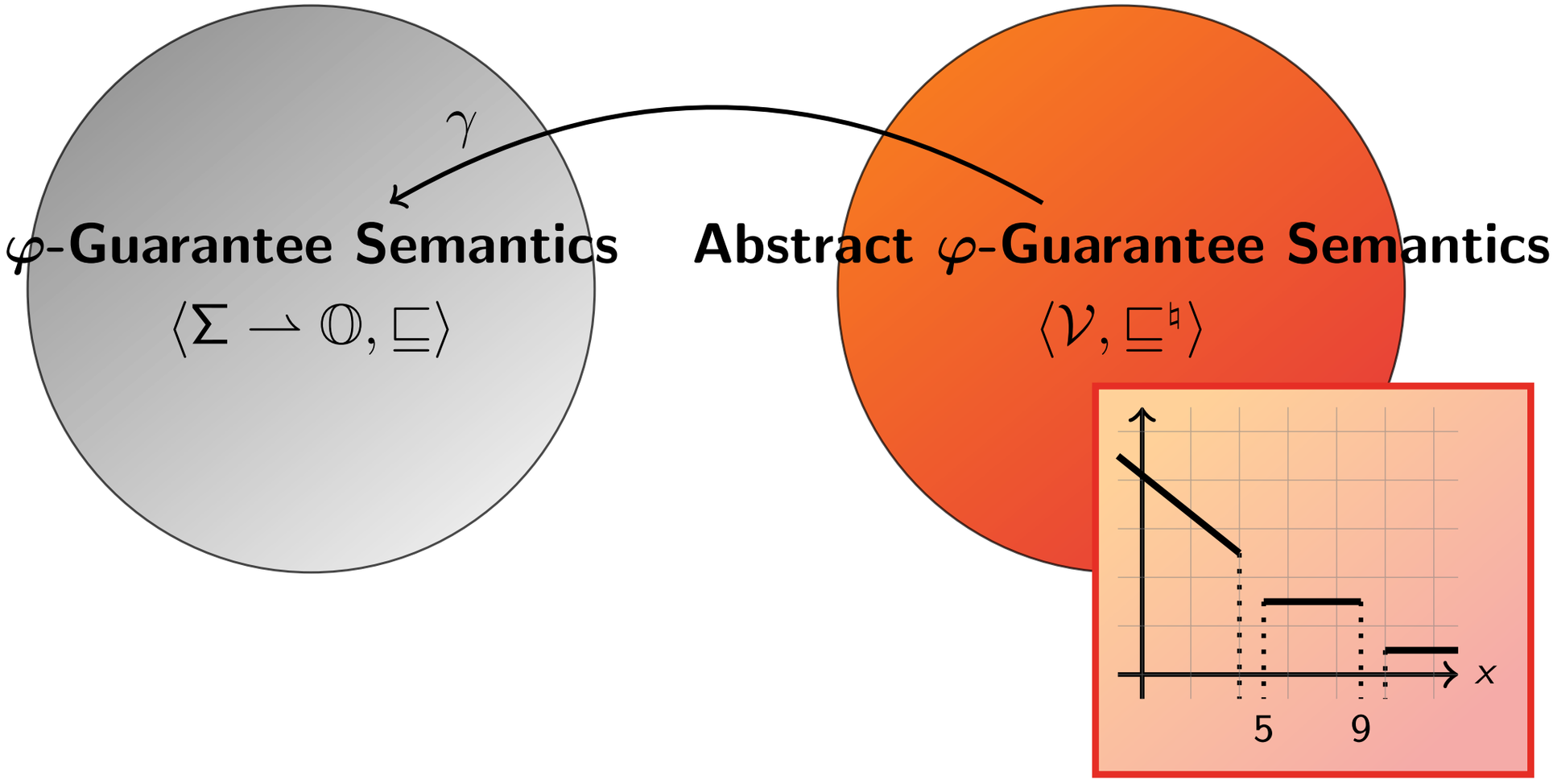
$$\mathcal{T}_{\boldsymbol{g}}^{\varphi} \stackrel{\text{def}}{=} \mathbf{lfp}\ \boldsymbol{F}_{\boldsymbol{g}}^{\varphi}$$

$$\boldsymbol{F}_{\boldsymbol{g}}^{\varphi}(\boldsymbol{v})\boldsymbol{s} \stackrel{\text{def}}{=} \begin{cases} \mathbf{0} & \boldsymbol{s} \in \varphi \\ \mathbf{sup}\{\ \boldsymbol{v}(\boldsymbol{s}') + \mathbf{1} \mid \langle \boldsymbol{s}, \boldsymbol{s}' \rangle \in \tau\ \} & \boldsymbol{s} \in \widetilde{\mathbf{pre}}(\mathbf{dom}(\boldsymbol{v})) \ \wedge\ \boldsymbol{s} \notin \varphi \\ \mathbf{undefined} & \mathbf{otherwise} \end{cases}$$

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

Theorem (Soundness and Completeness)

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


γ
φ-Guarantee Semantics
$\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$
Abstract φ-Guarantee Semantics
$\langle \mathcal{V}, \sqsubseteq^\natural \rangle$
x
5
9

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]$x := x + 1$

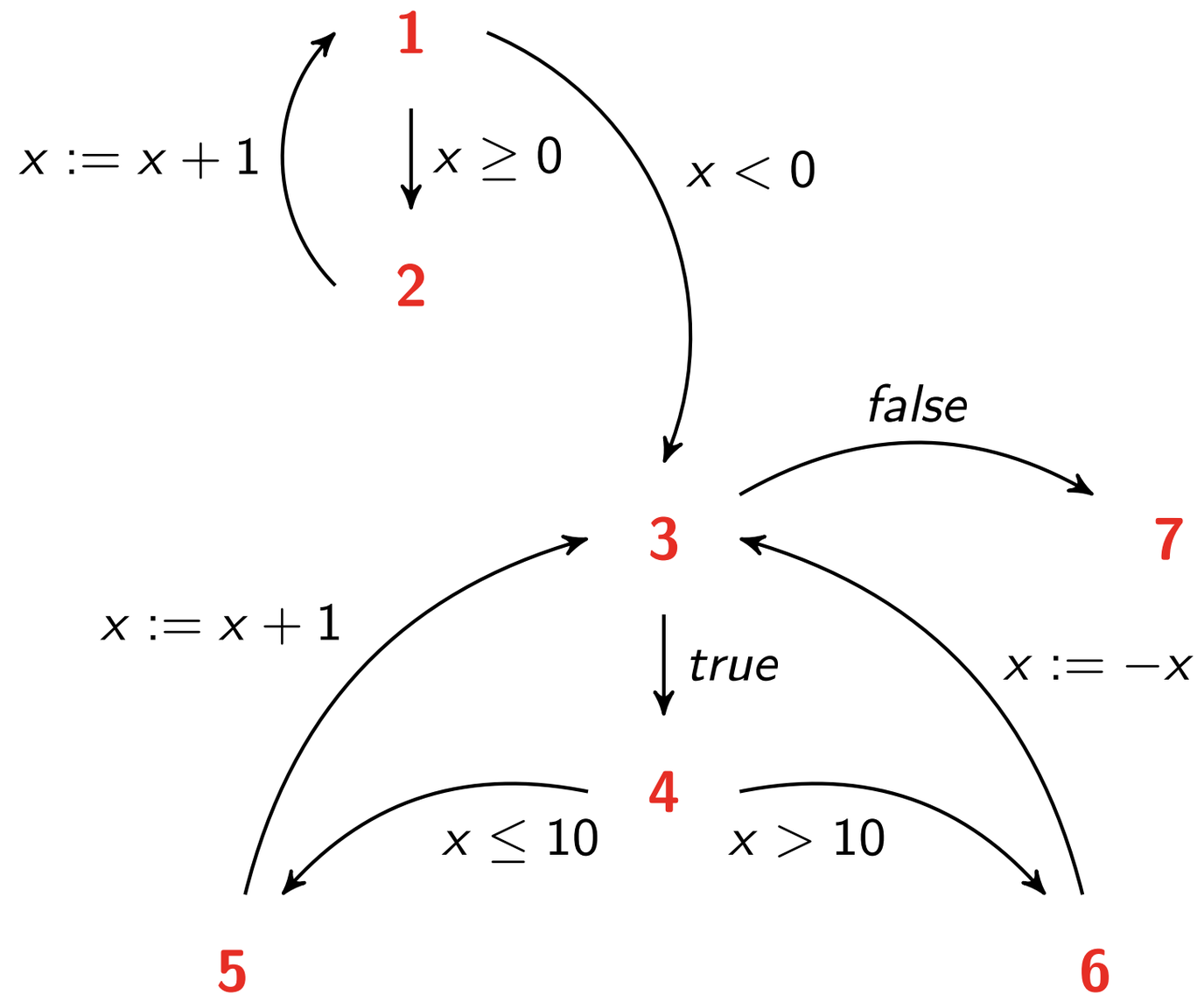
  else

    [6]$x := -x$

od[7]

Guarantee Property

$\Diamond\ x = 3$



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] $x := x + 1$

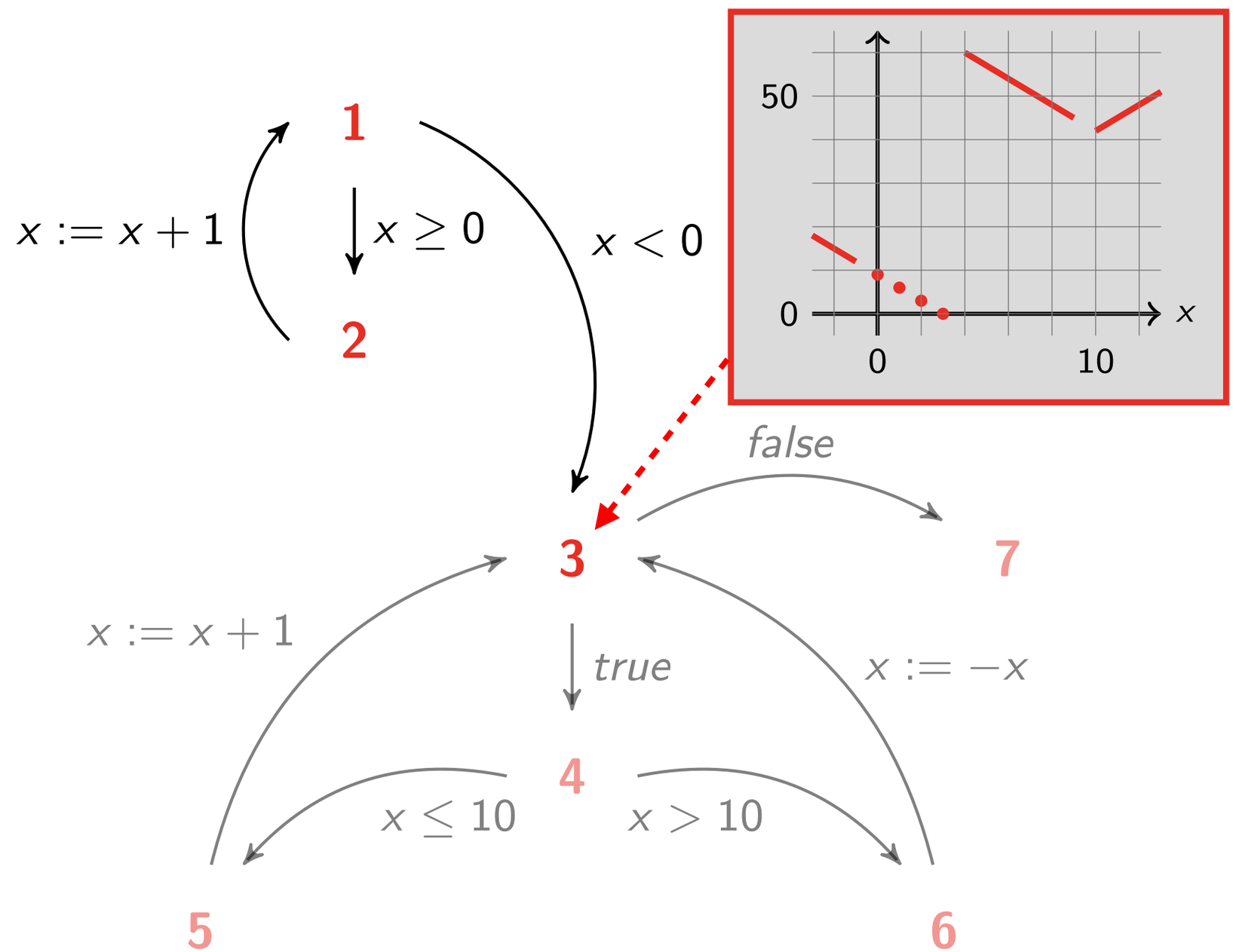
  else

    [6] $x := -x$

od [7]

Guarantee Property

$\Diamond\, x = 3$



Example

int : x, y

[1] while $(x \geq 0)$

  [2] $x := x + 1$

od

[3] while (true) do

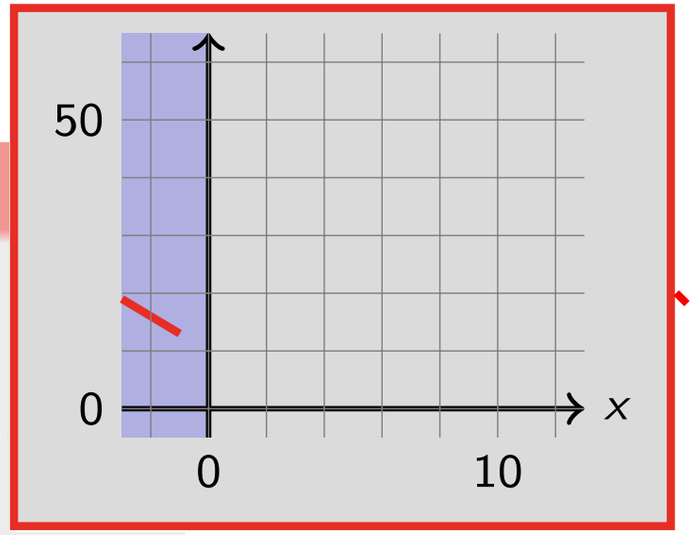
  [4] if $(x \leq 10)$

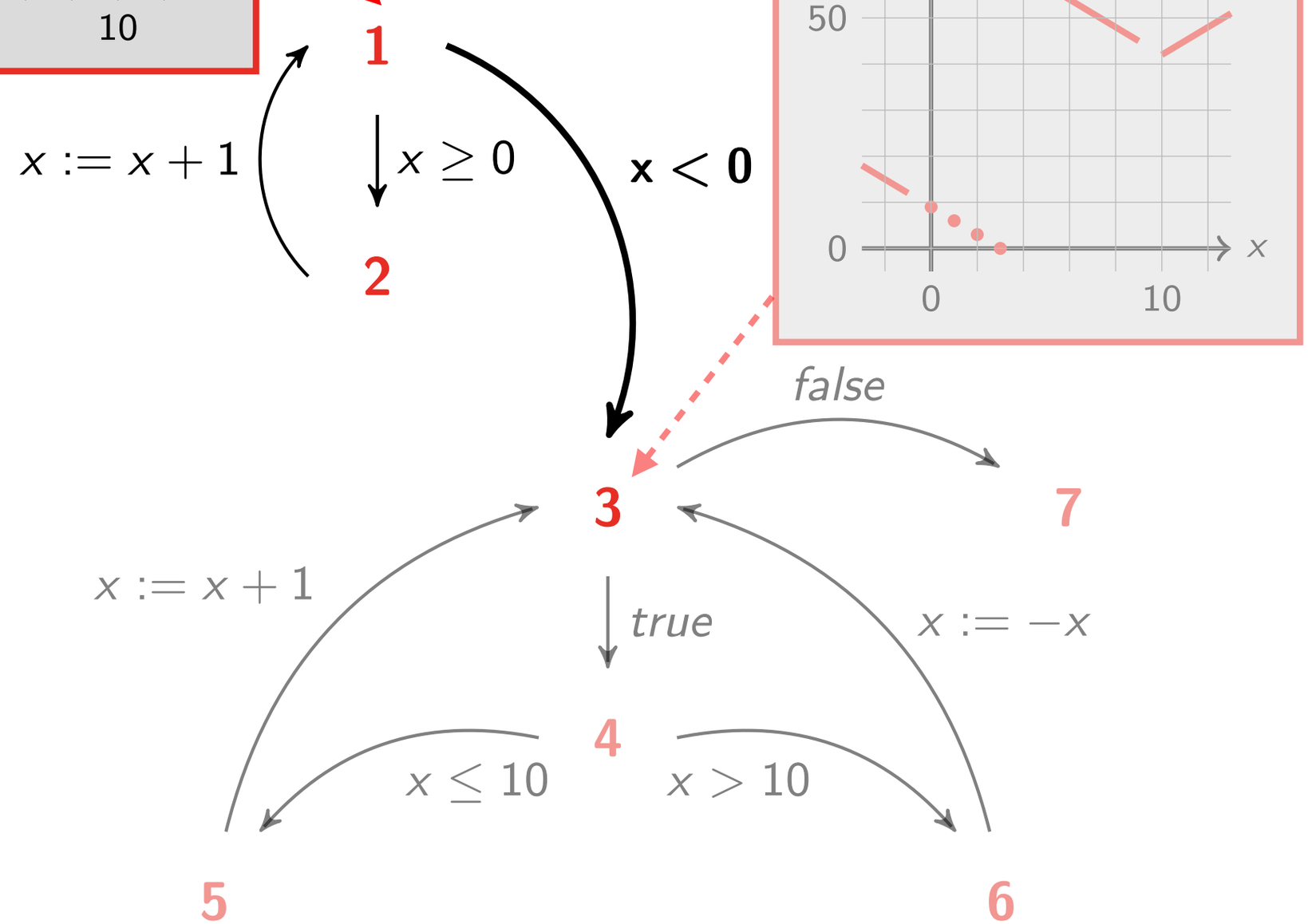
    [5] $x := x + 1$

  else

    [6] $x := -x$

od [7]





Guarantee Property

$\Diamond\ x = 3$

Example

int : x, y

while [1] $(x \geq 0)$

  [2] $x := x + 1$

od

while [3] (true) do

  if [4] ($x \leq 10$)

    [5] $x := x + 1$

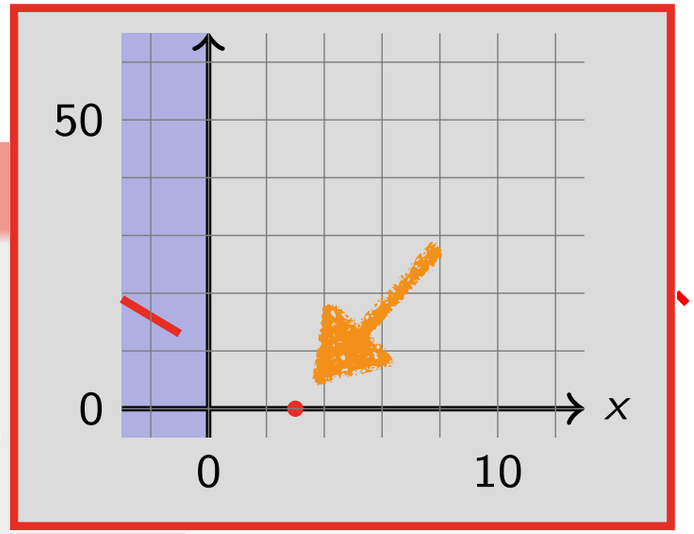
  else

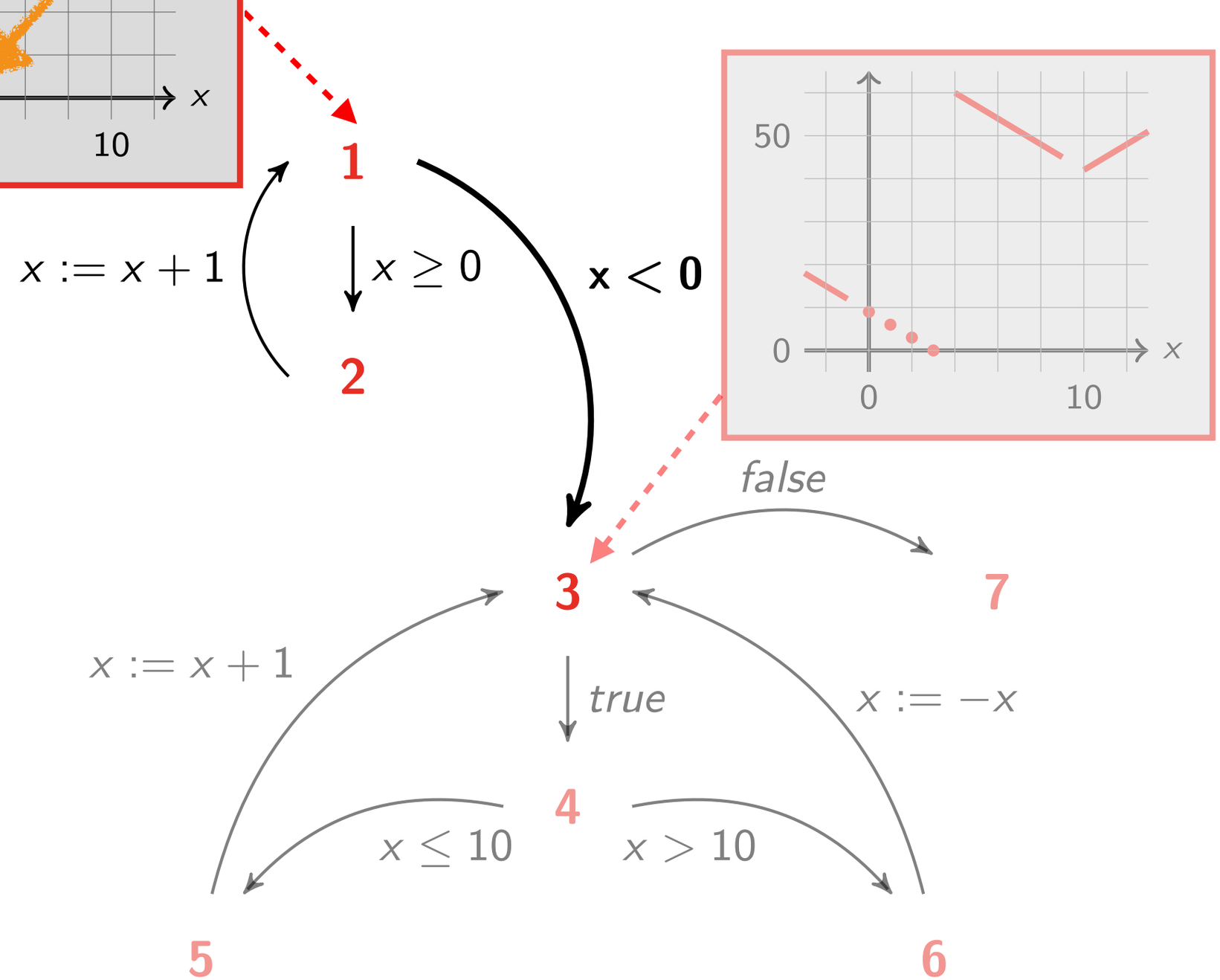
    [6] $x := -x$

od [7]

Guarantee Property

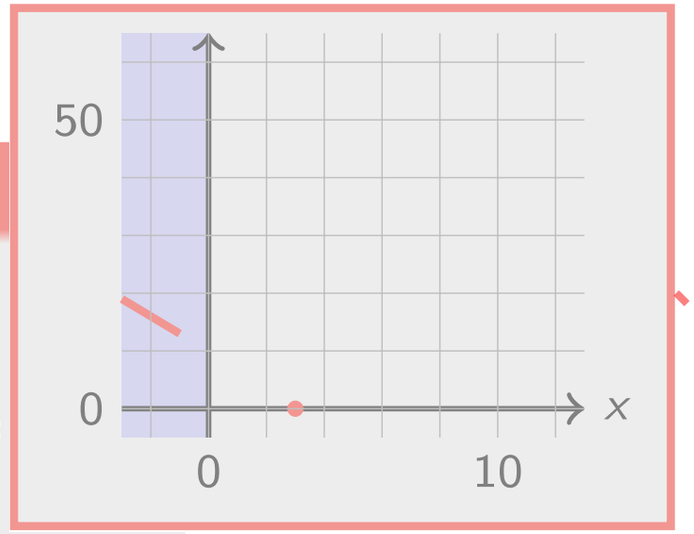
$\Diamond\ x = 3$

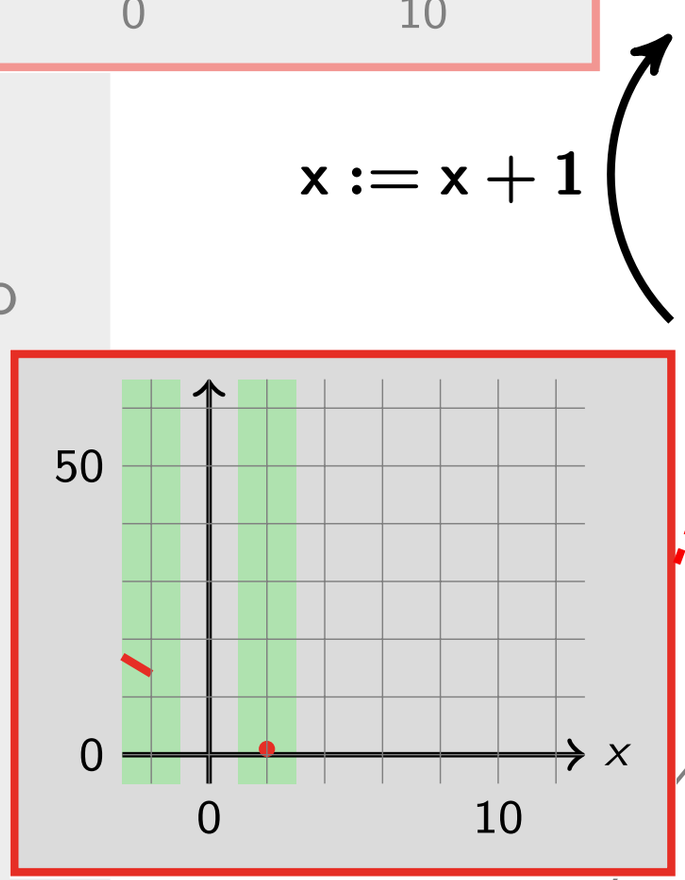


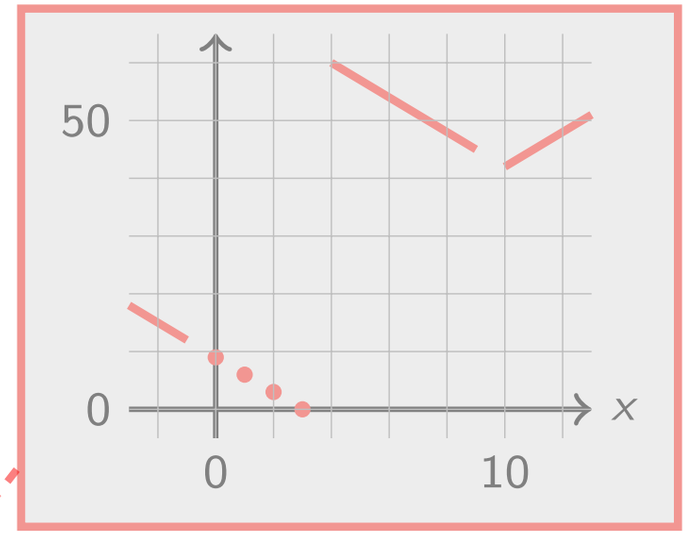


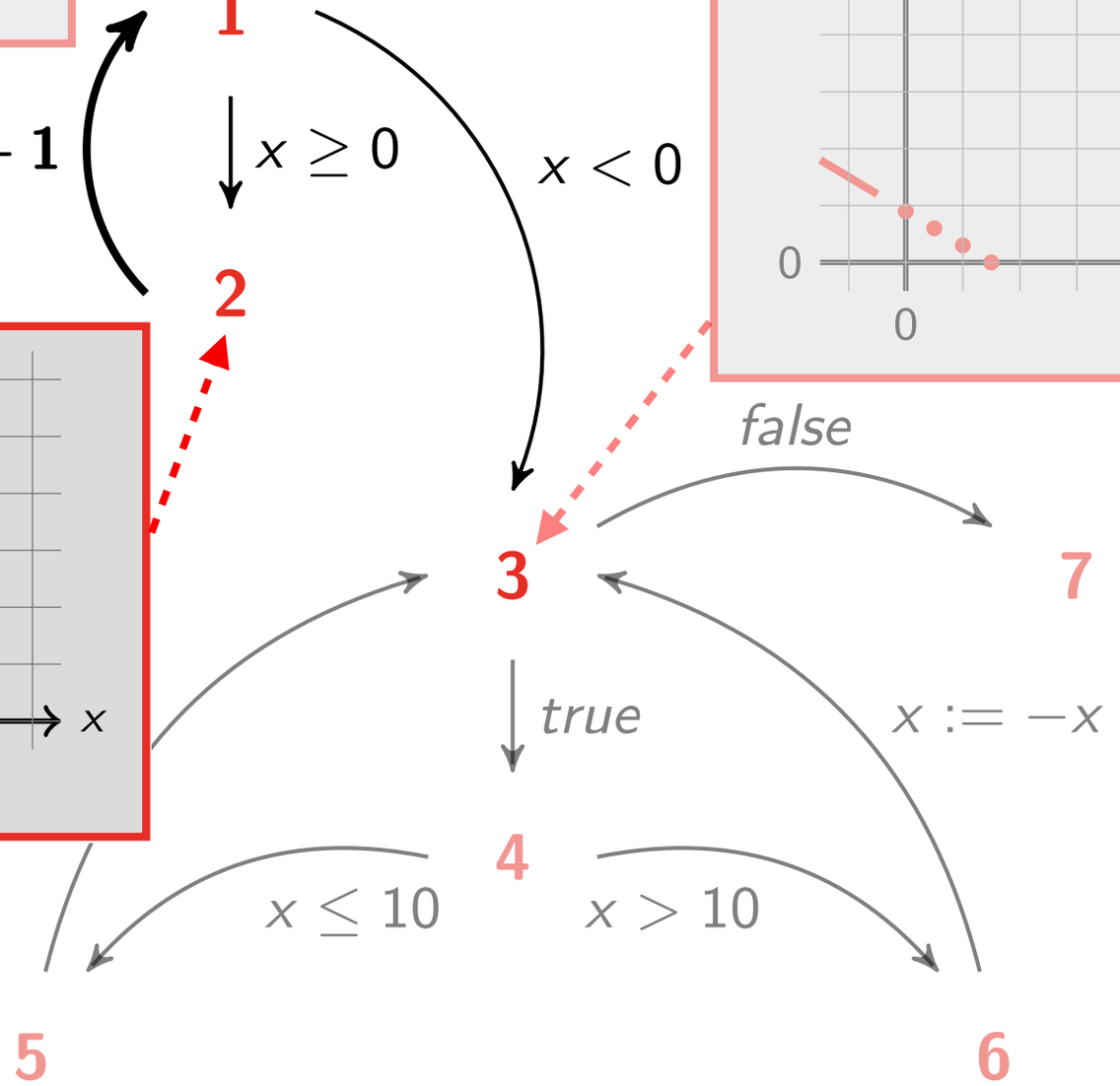
Example

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







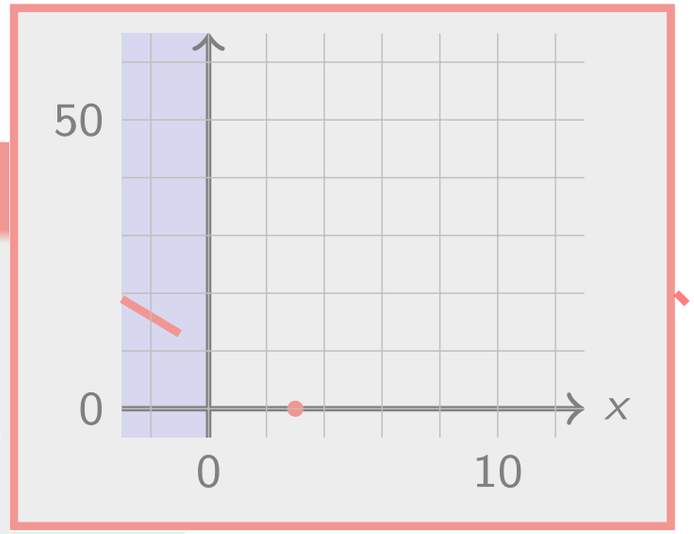


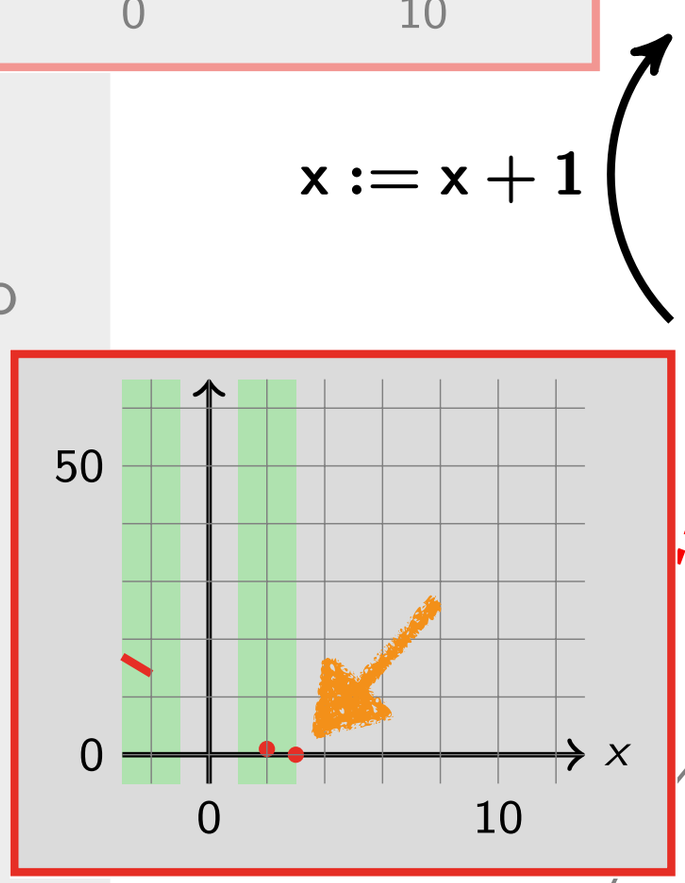
Guarantee Property

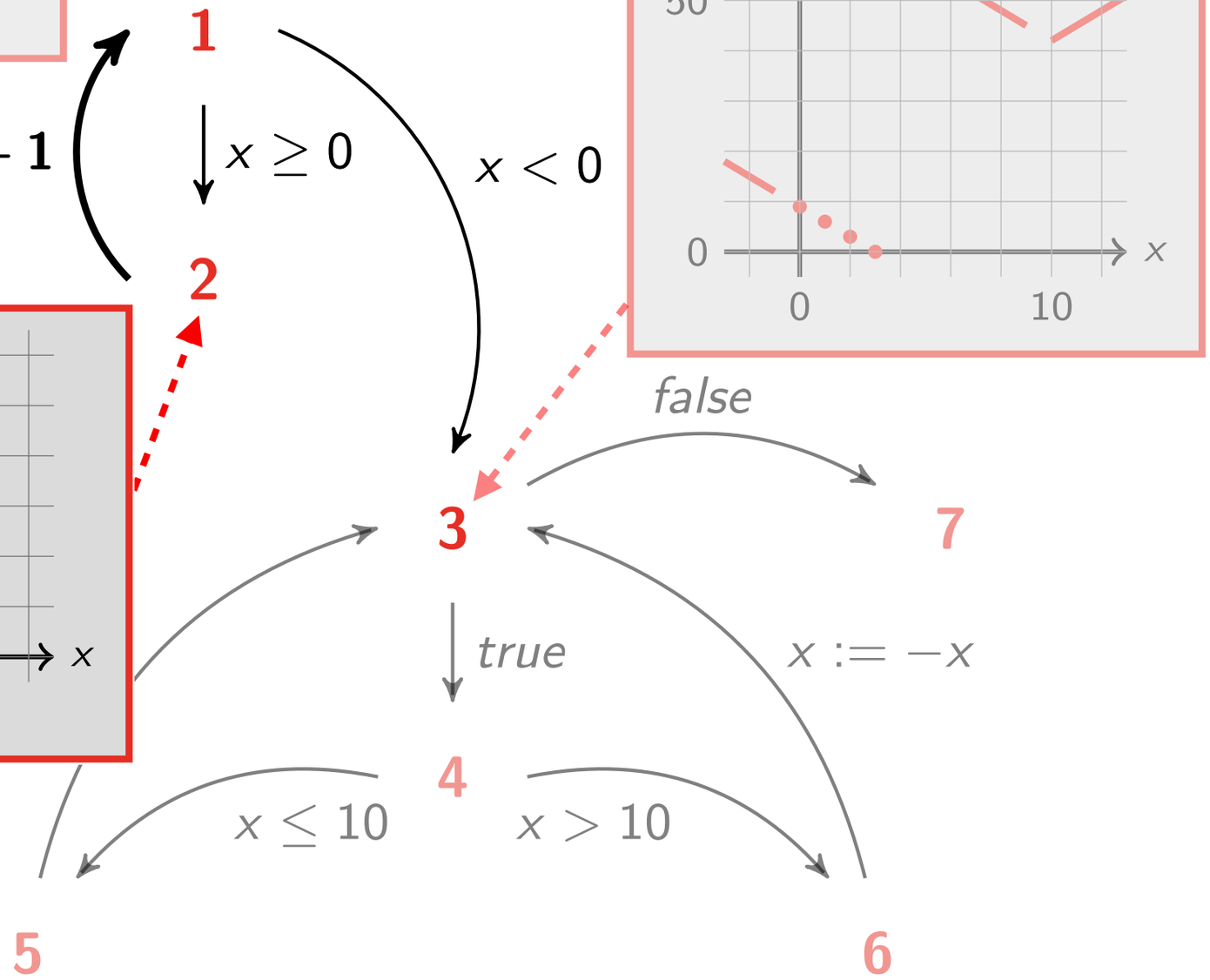
$\Diamond\ x = 3$

Example

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







Guarantee Property

$\Diamond\ x = 3$

Example

int : x, y

[1] while $(x \geq 0)$

  [2] $x := x + 1$

od

[3] while (true) do

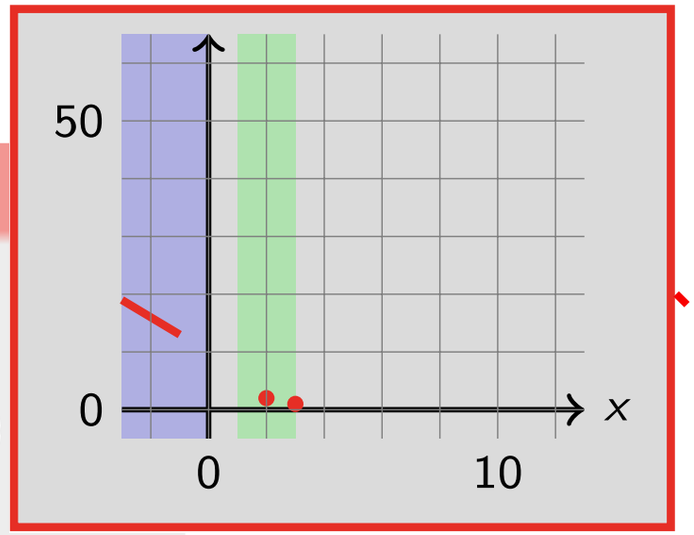
  [4] if $(x \leq 10)$

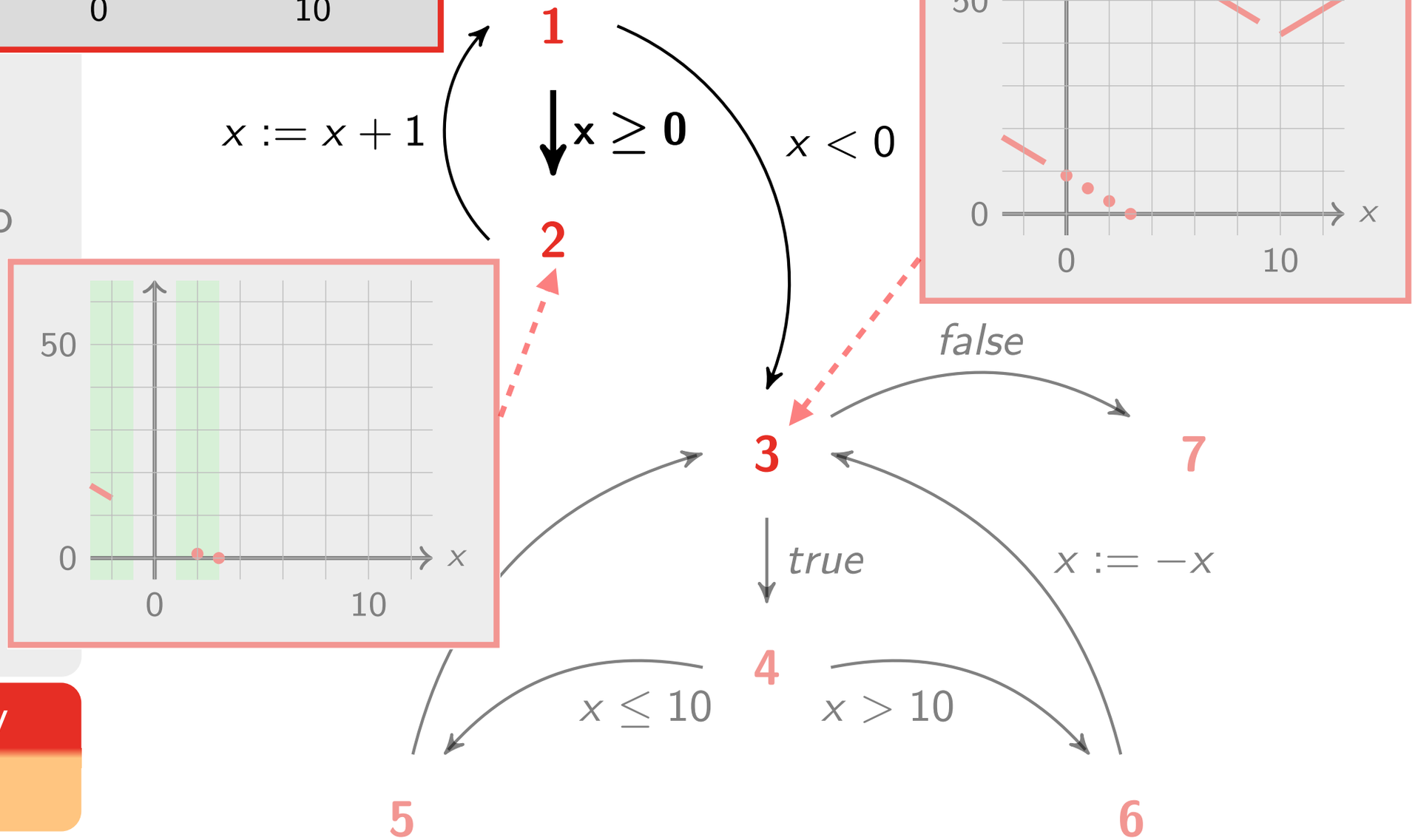
    [5] $x := x + 1$

  else

    [6] $x := -x$

od [7]





Guarantee Property

$\Diamond\, x = 3$

Example

int : x, y

[1] while $(x \geq 0)$

  [2] $x := x + 1$

od

[3] while (true) do

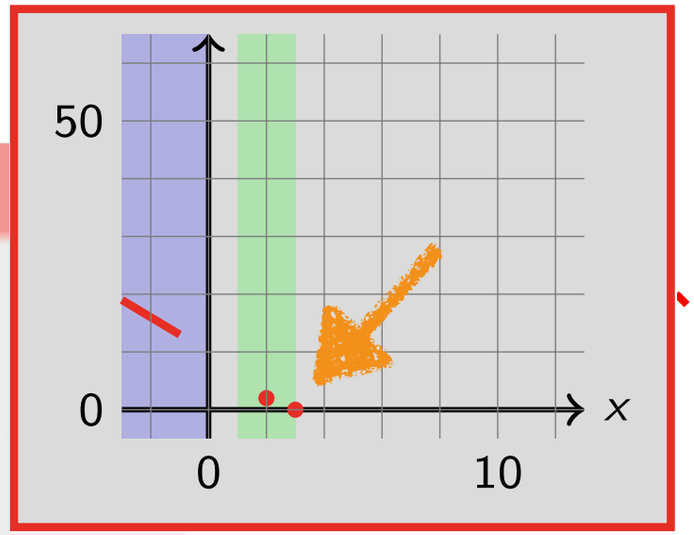
  [4] if ($x \leq 10$)

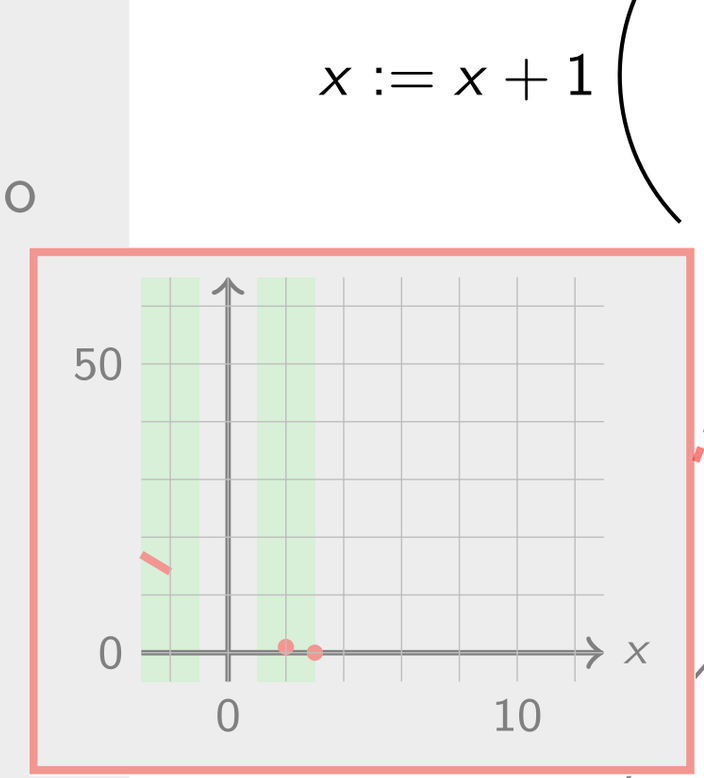
    [5] $x := x + 1$

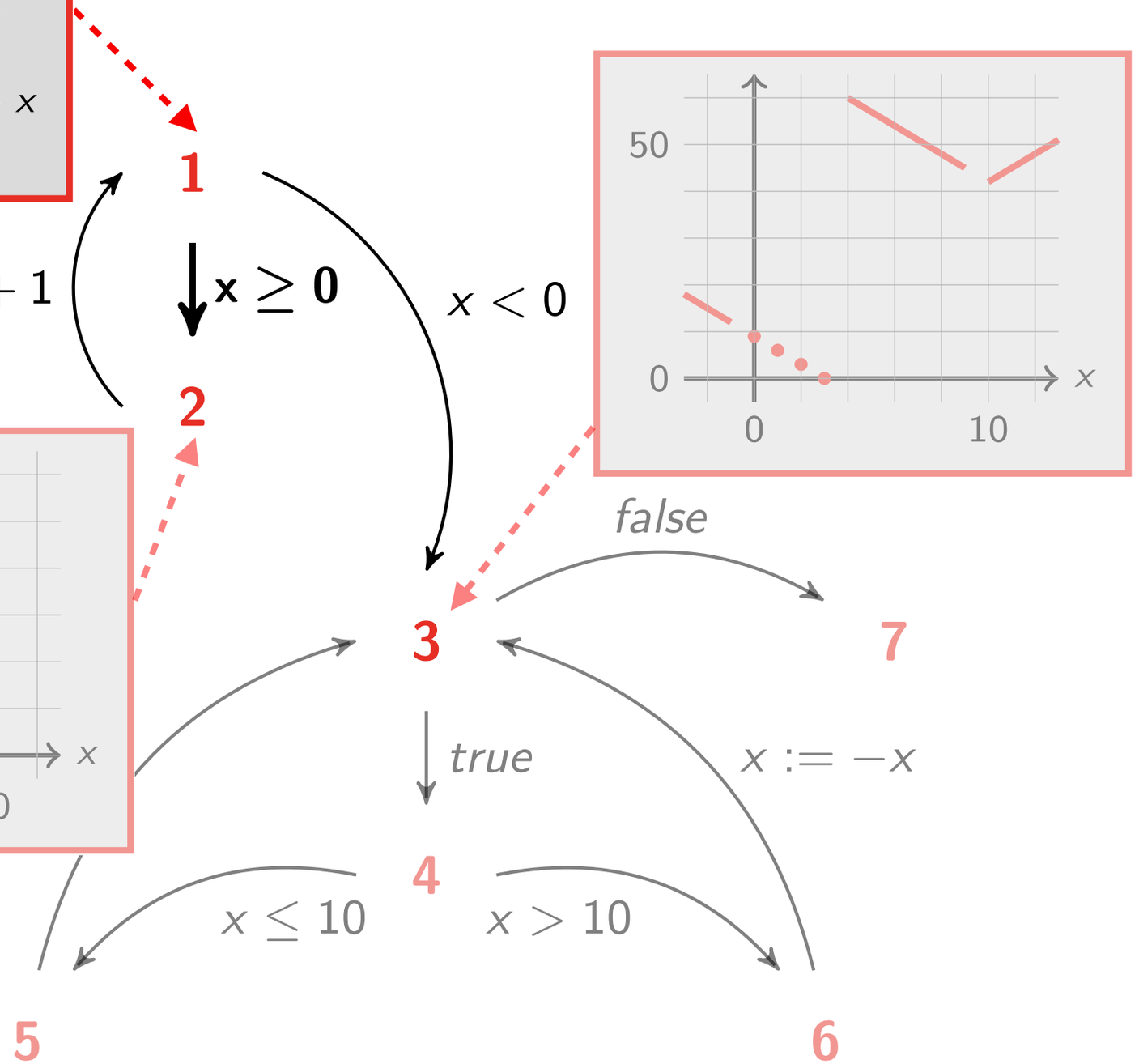
  else

    [6] $x := -x$

od [7]







Guarantee Property

$\Diamond\ x = 3$

Example

int : x, y

[1] while ($x \geq 0$)

  [2] $x := x + 1$

od

[3] while (true) do

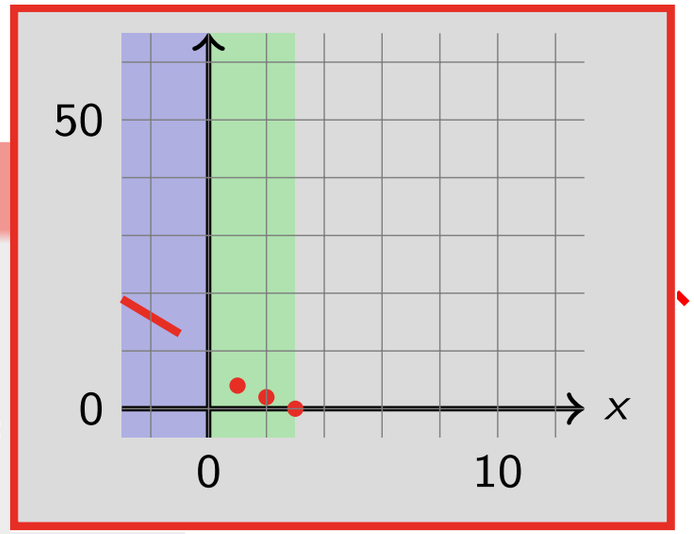
  [4] if ($x \leq 10$)

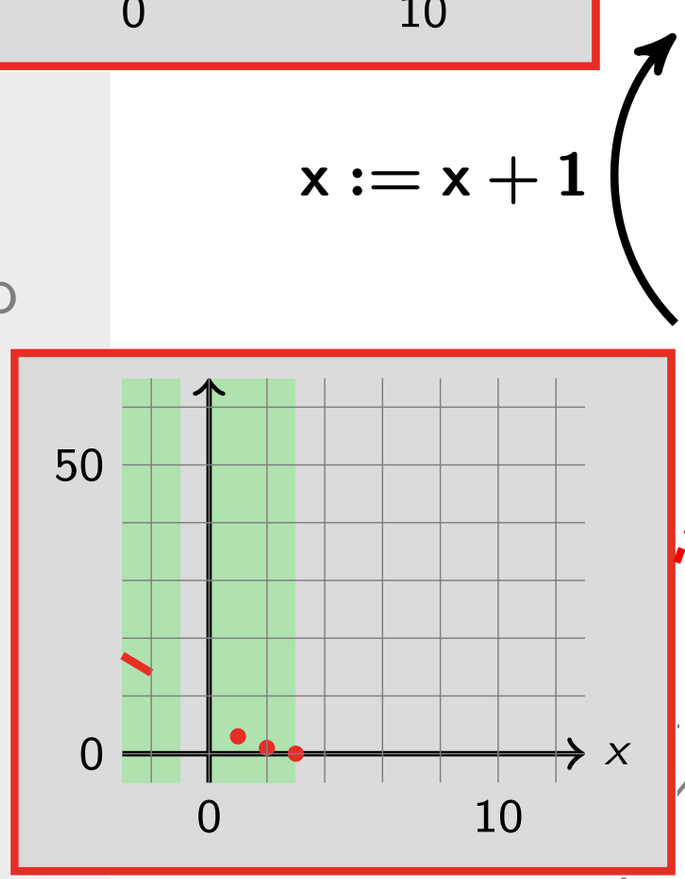
    [5] $x := x + 1$

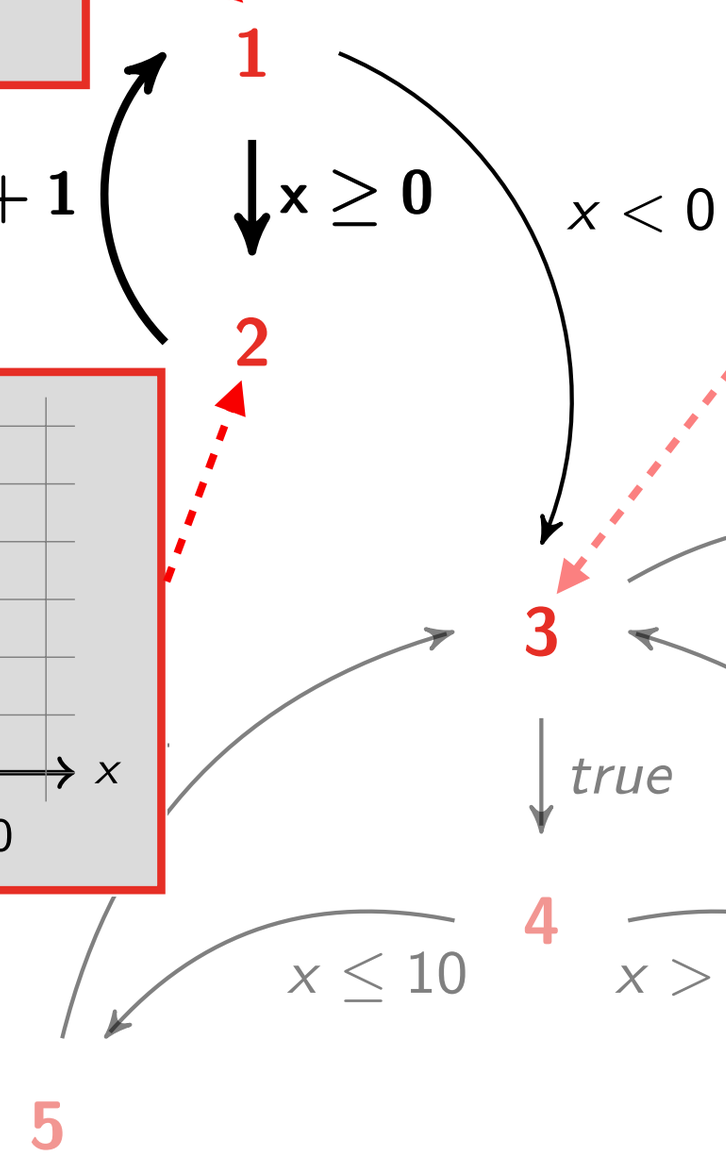
  else

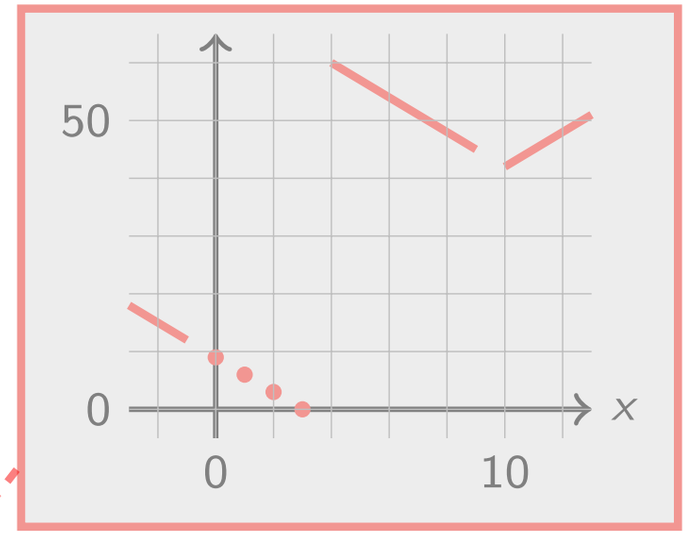
    [6] $x := -x$

od [7]







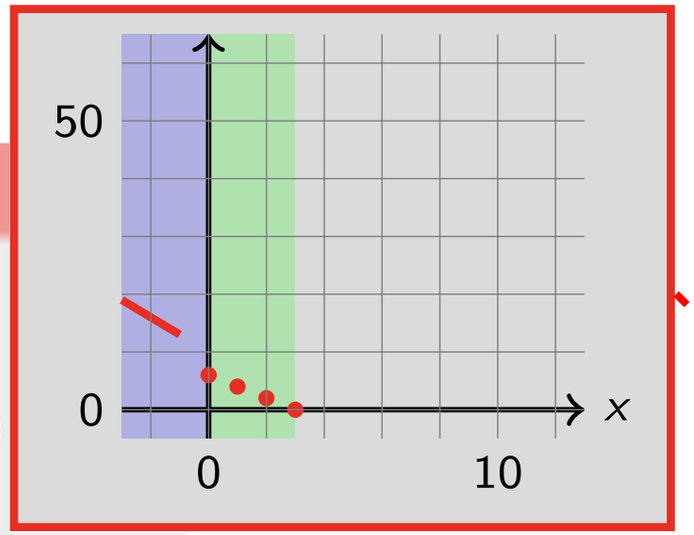


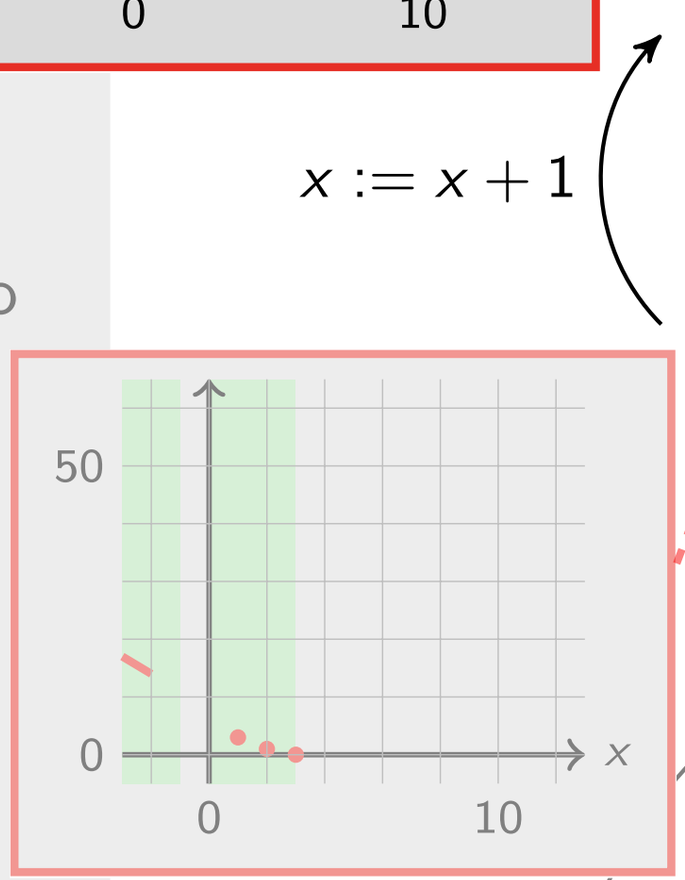
Guarantee Property

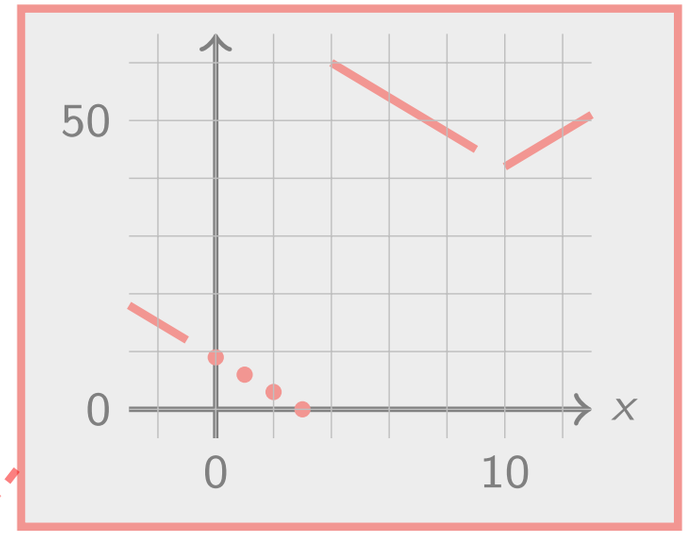
$\Diamond\ x = 3$

Example

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







Guarantee Property

$\Diamond\ x = 3$

Example

int : x, y

[1] while $(x \geq 0)$

  [2] $x := x + 1$

od

[3] while (true) do

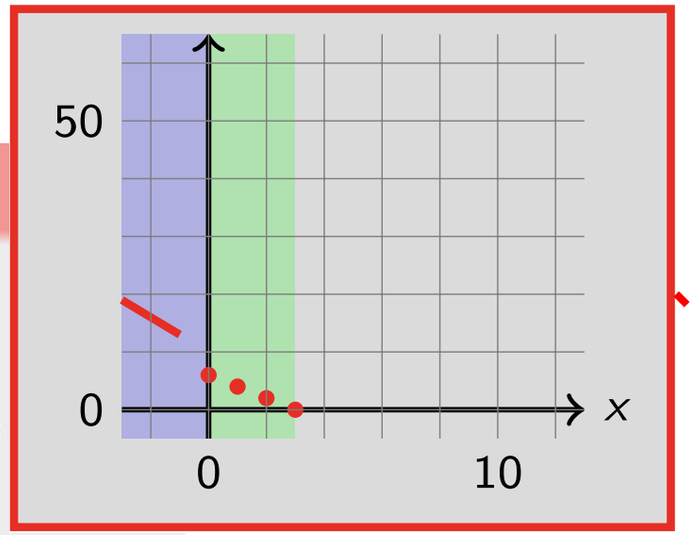
  [4] if $(x \leq 10)$

    [5] $x := x + 1$

  else

    [6] $x := -x$

od [7]



Guarantee Property

$\Diamond\, x = 3$

Example

int : x, y

while [1] $(x \geq 0)$

  [2] $x := x + 1$

od

while [3] (true) do

  if [4] ($x \leq 10$)

    [5] $x := x + 1$

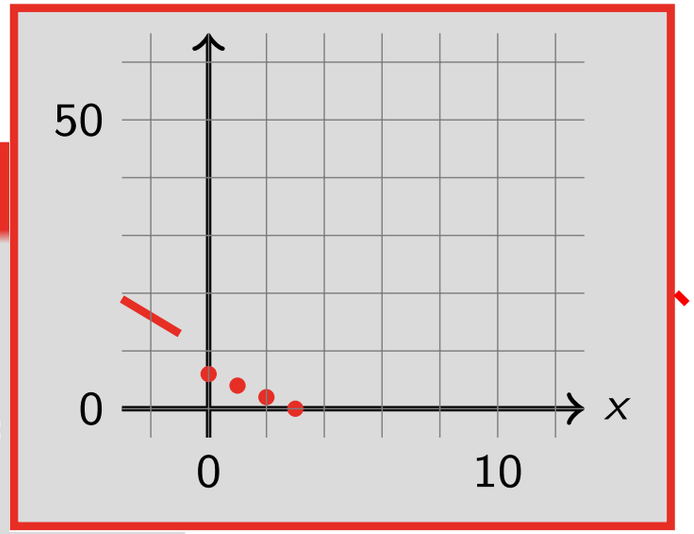
  else

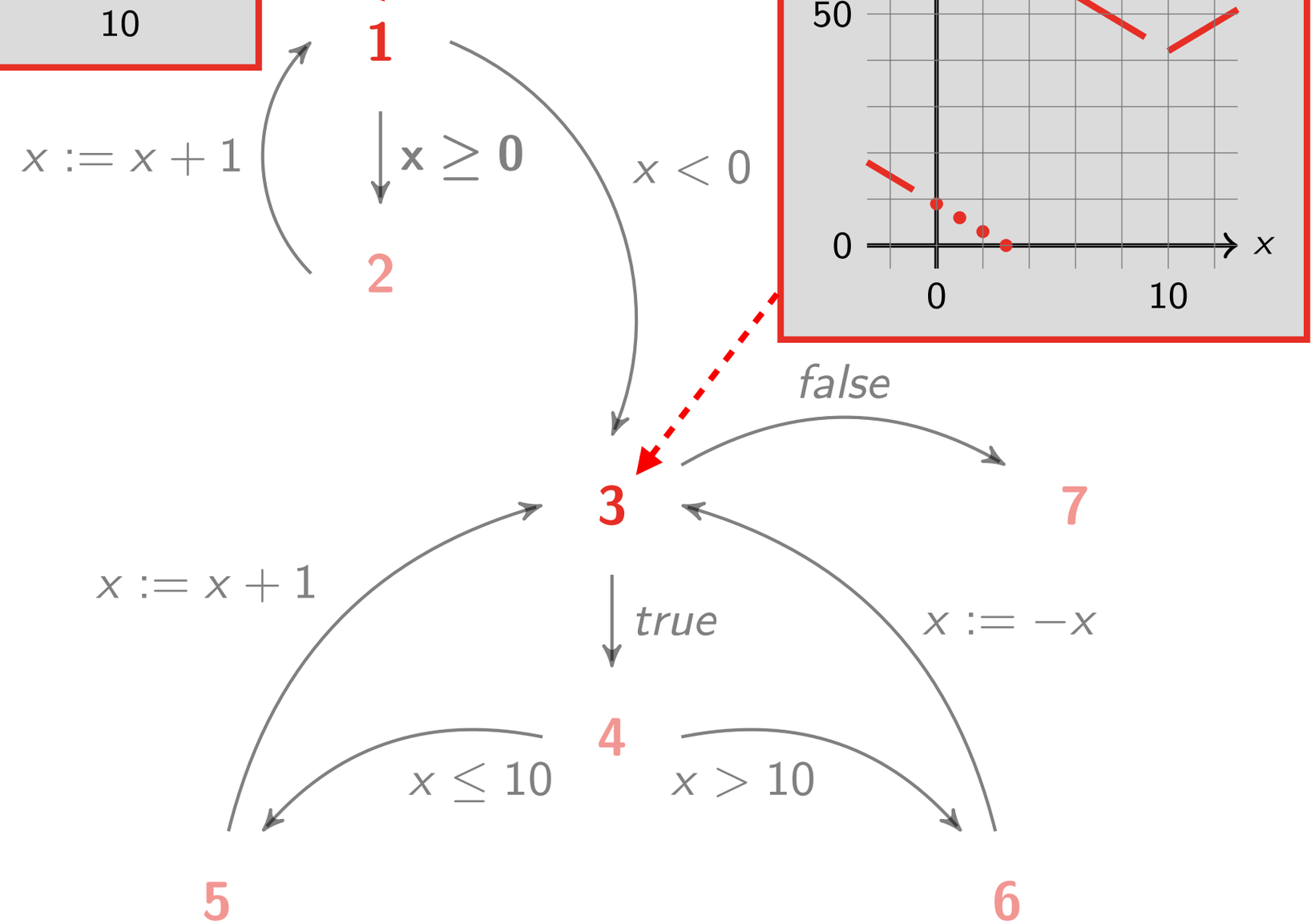
    [6] $x := -x$

od [7]

Guarantee Property

$\Diamond\ x = 3$





Example

int : x, y

while [1]$(x \geq 0)$

  [2]$x := x + 1$

od

while [3](true) do

  if [4]($x \leq 10$)

    [5]$x := x + 1$

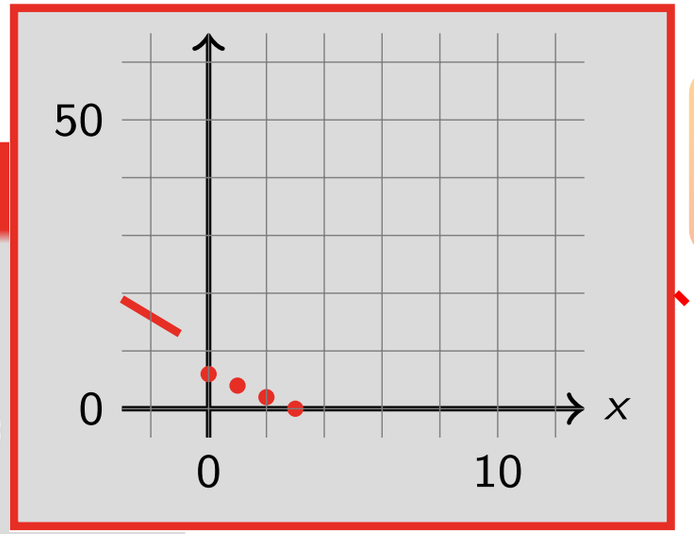
  else

    [6]$x := -x$

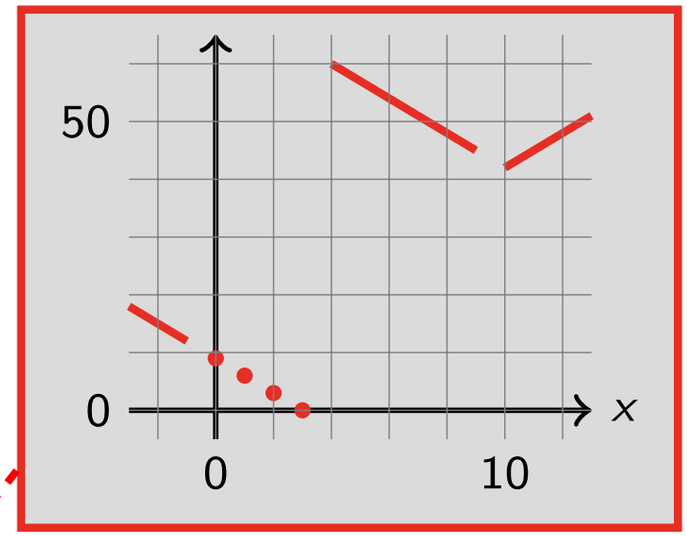
od[7]

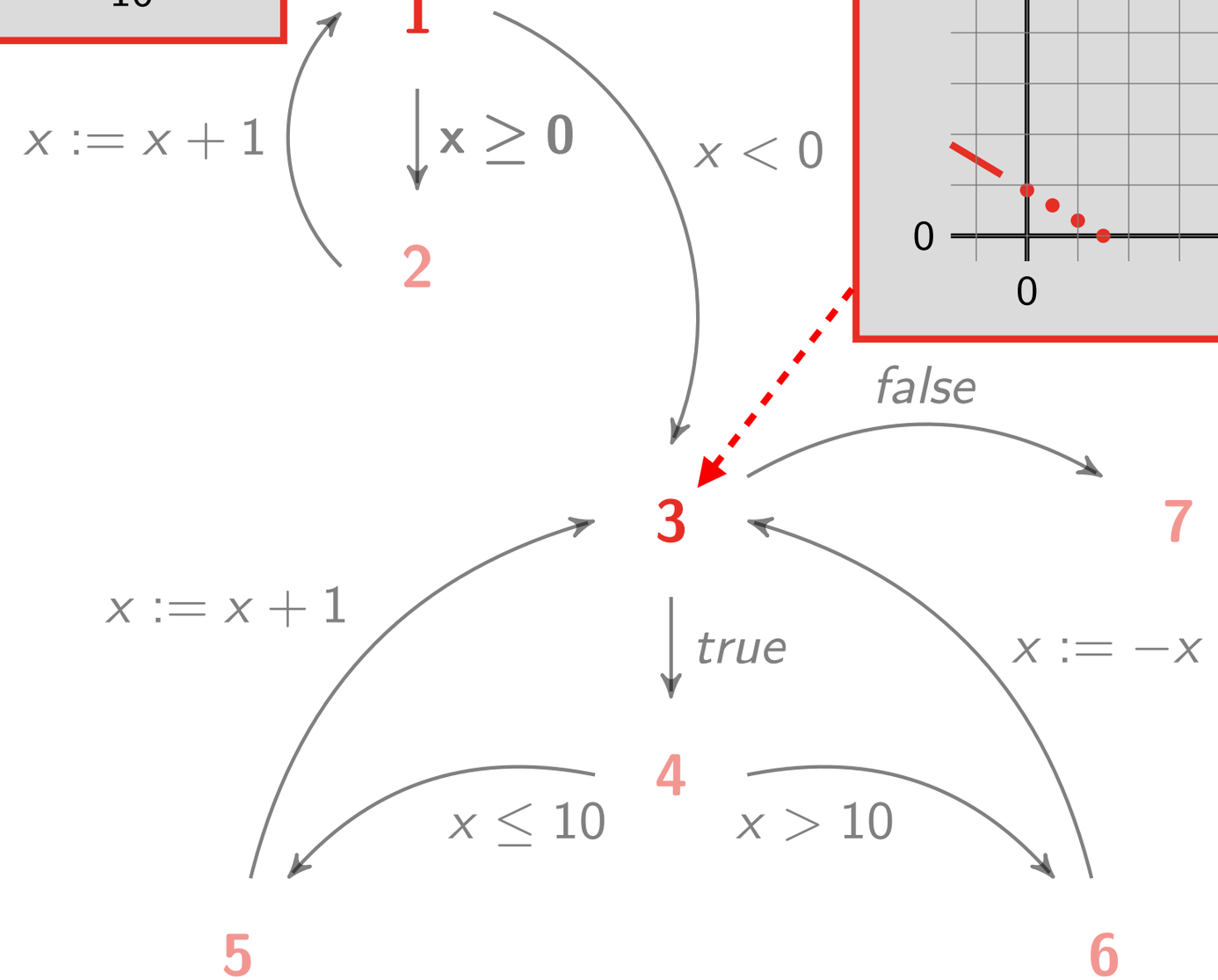
Guarantee Property

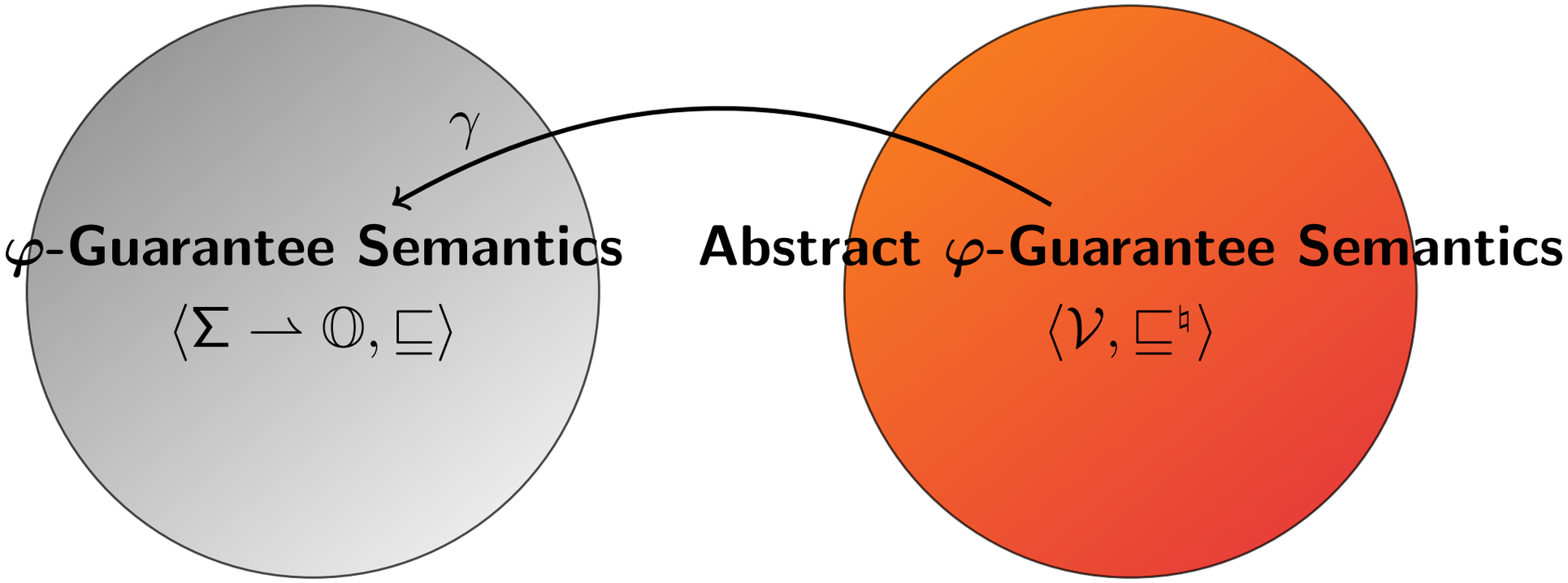
$\Diamond\ x = 3$



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







Theorem (Soundness)

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

Recurrence Properties

program $\mapsto$ maximal trace semantics $\to$ **φ-recurrence semantics**

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

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

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

program $\mapsto$ maximal trace semantics $\to$ **φ-recurrence semantics**

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

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

program $\mapsto$ maximal trace semantics $\to$ **φ-recurrence semantics**

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

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

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

program $\mapsto$ maximal trace semantics $\to$ **φ-recurrence semantics**

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

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

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

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

program $\mapsto$ maximal trace semantics $\rightarrow$ **φ-recurrence semantics**

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

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

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

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

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

program $\mapsto$ maximal trace semantics $\to$ **φ-recurrence semantics**

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

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

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

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

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

$$\vdots$$

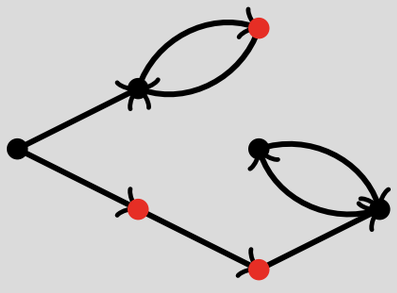
program $\mapsto$ maximal trace semantics $\to$ **φ-recurrence semantics**

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

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

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

Example



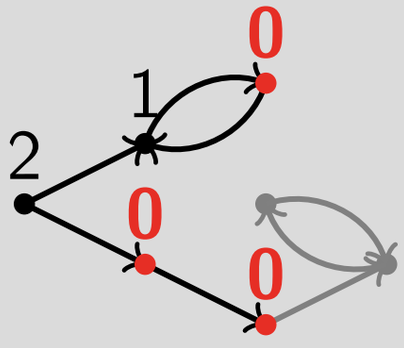
program $\mapsto$ maximal trace semantics $\rightarrow$ **φ-recurrence semantics**

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

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

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

Example



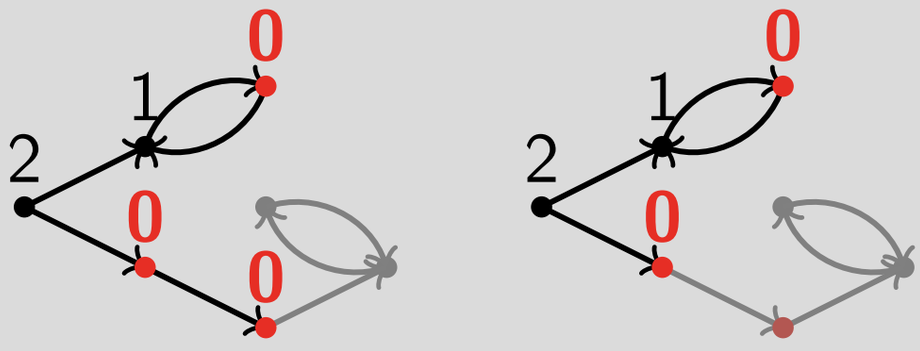
program $\mapsto$ maximal trace semantics $\rightarrow$ **φ-recurrence semantics**

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

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

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

Example



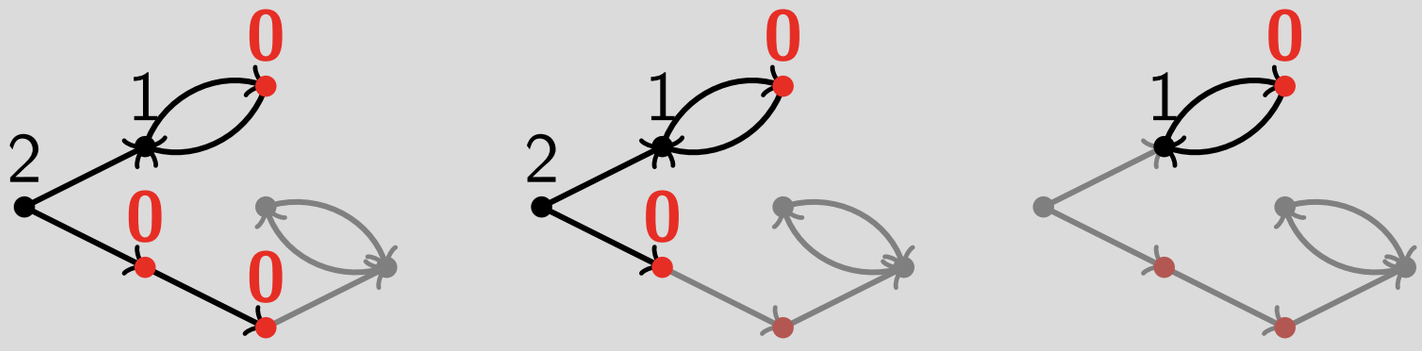
program $\mapsto$ maximal trace semantics $\rightarrow$ **φ-recurrence semantics**

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

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

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

Example



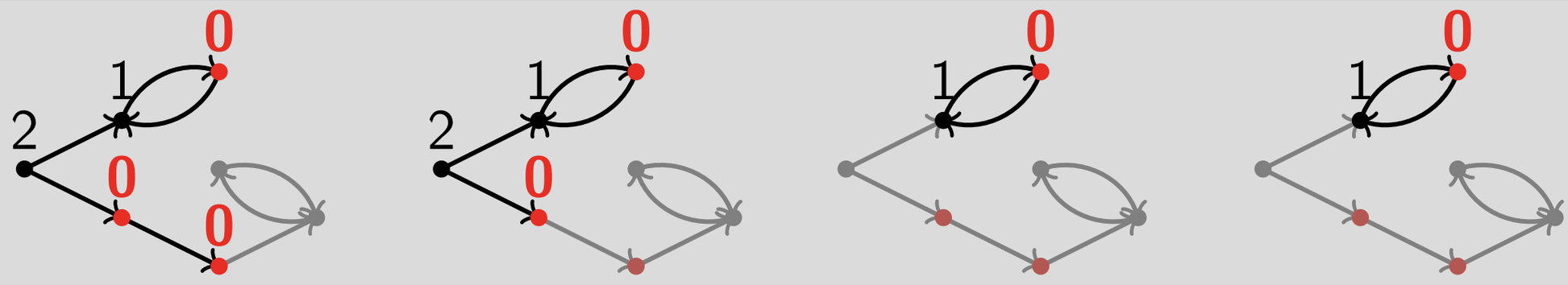
program $\mapsto$ maximal trace semantics $\to$ **φ-recurrence semantics**

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

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

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

Example



program $\mapsto$ maximal trace semantics $\to$ **φ-recurrence semantics**

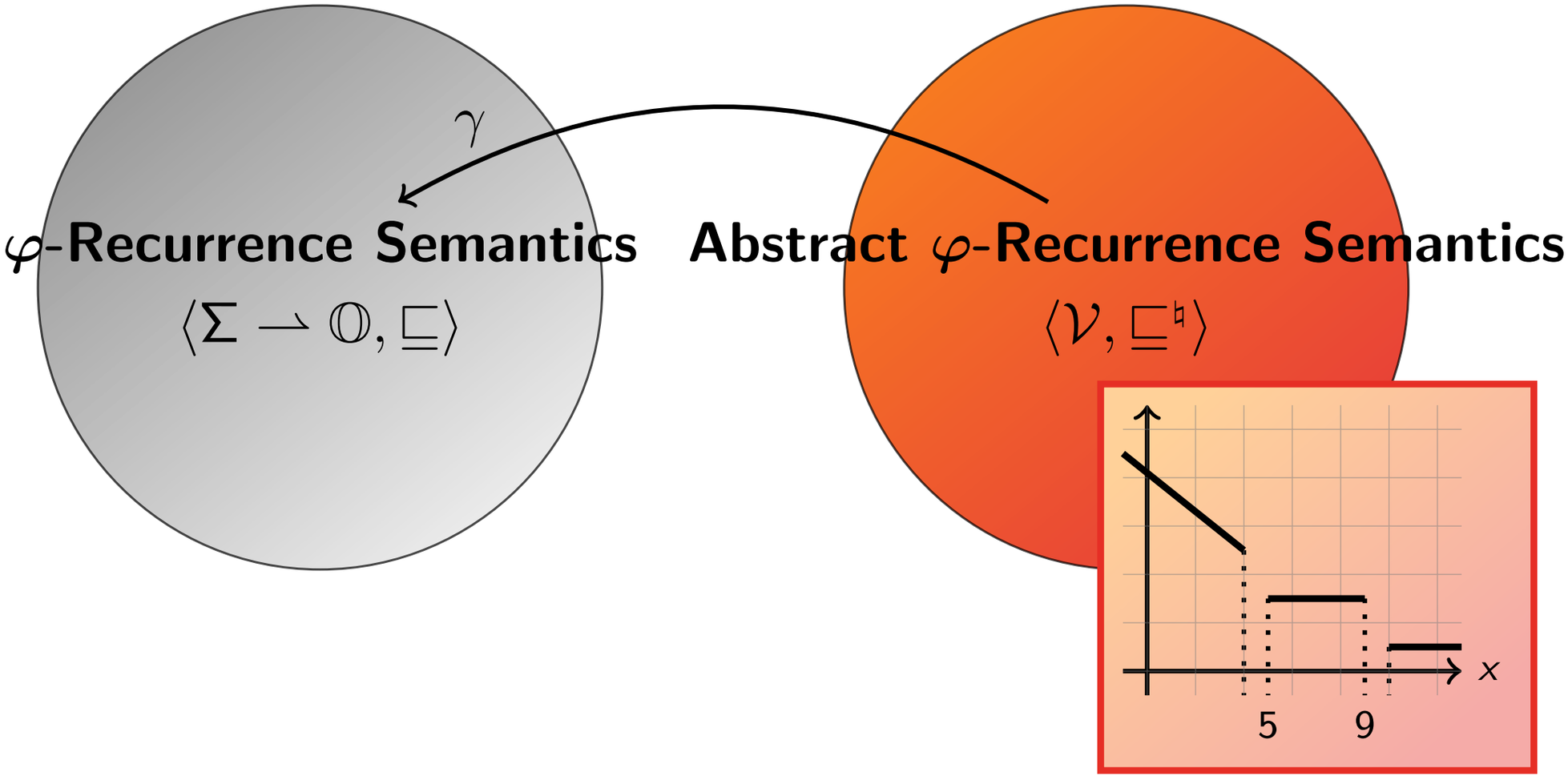
$$\mathcal{T}_r^\varphi \in \Sigma \rightharpoonup \mathbb{O}$$

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

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

Theorem (Soundness and Completeness)

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


γ
φ-Recurrence Semantics
$\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$
Abstract φ-Recurrence Semantics
$\langle \mathcal{V}, \sqsubseteq^\natural \rangle$
x
5
9

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]$x := x + 1$

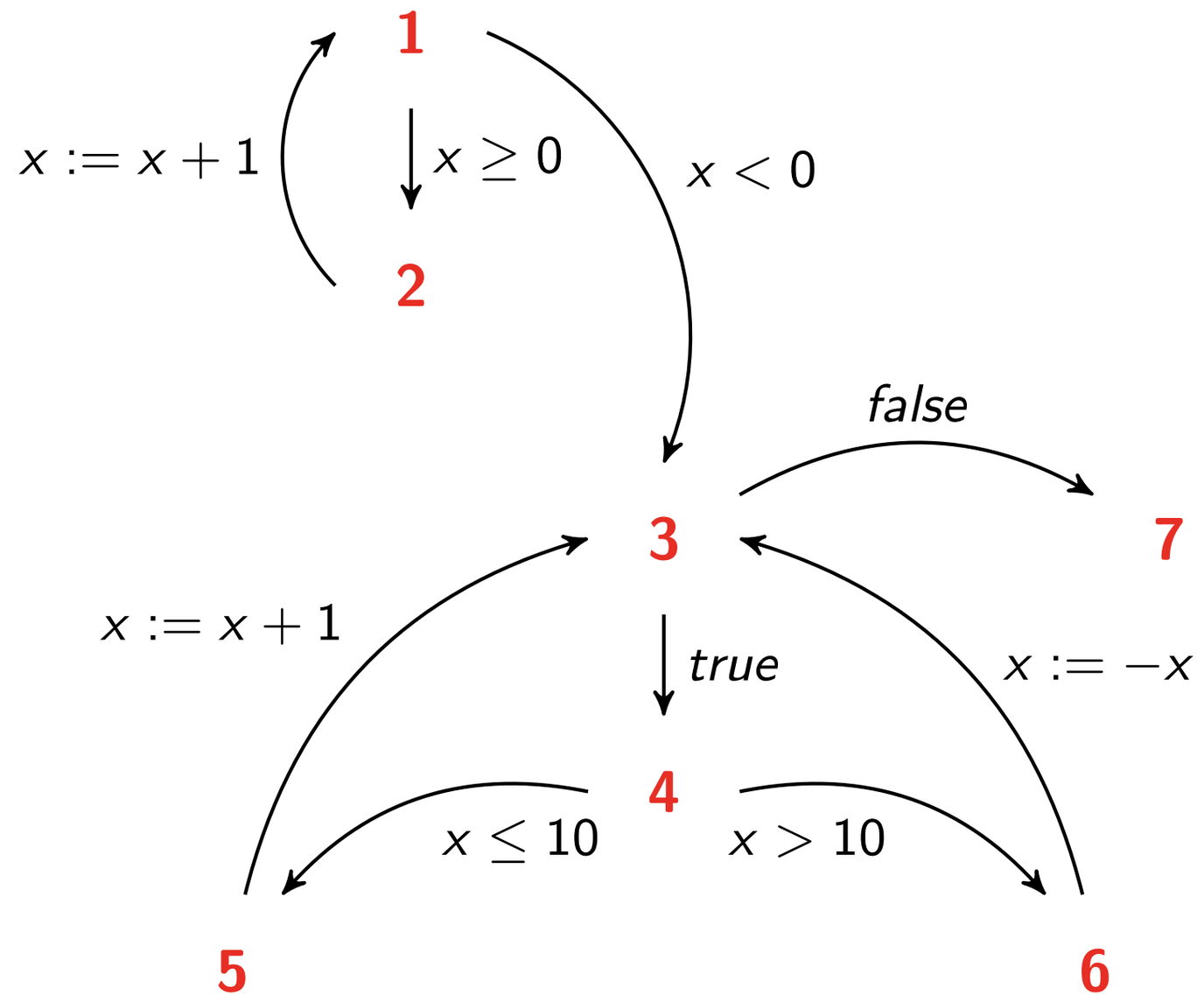
  else

    [6]$x := -x$

od[7]

Recurrence Property

$\Box\Diamond\ x = 3$



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] $x := x + 1$

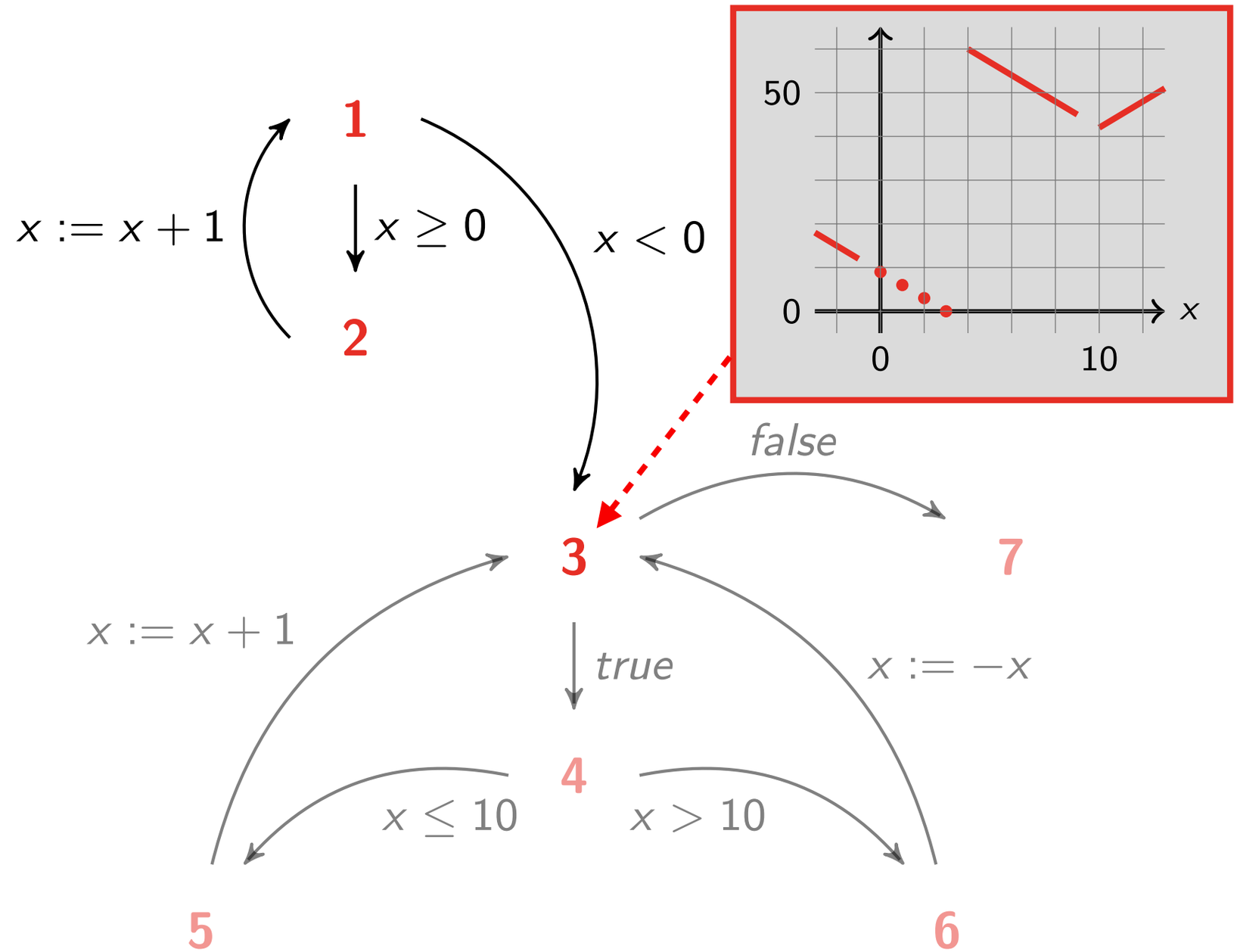
  else

    [6] $x := -x$

od [7]

Recurrence Property

$\square\Diamond\ x = 3$



Example

int : x, y

[1] while $(x \geq 0)$

  [2] $x := x + 1$

od

[3] while (true) do

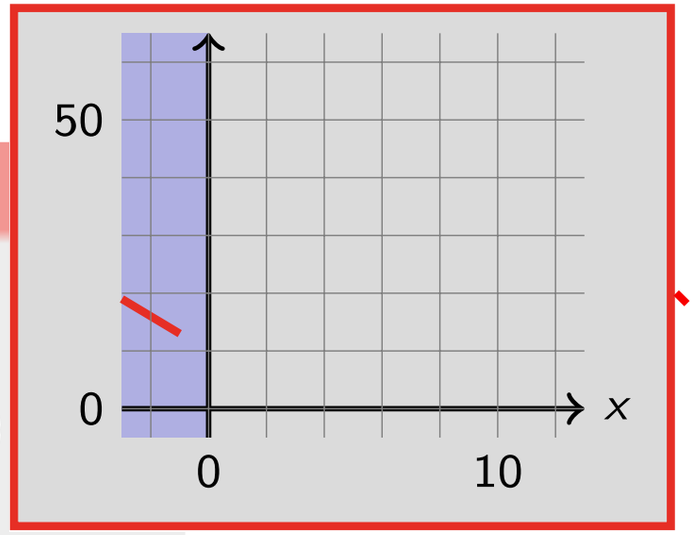
  [4] if $(x \leq 10)$

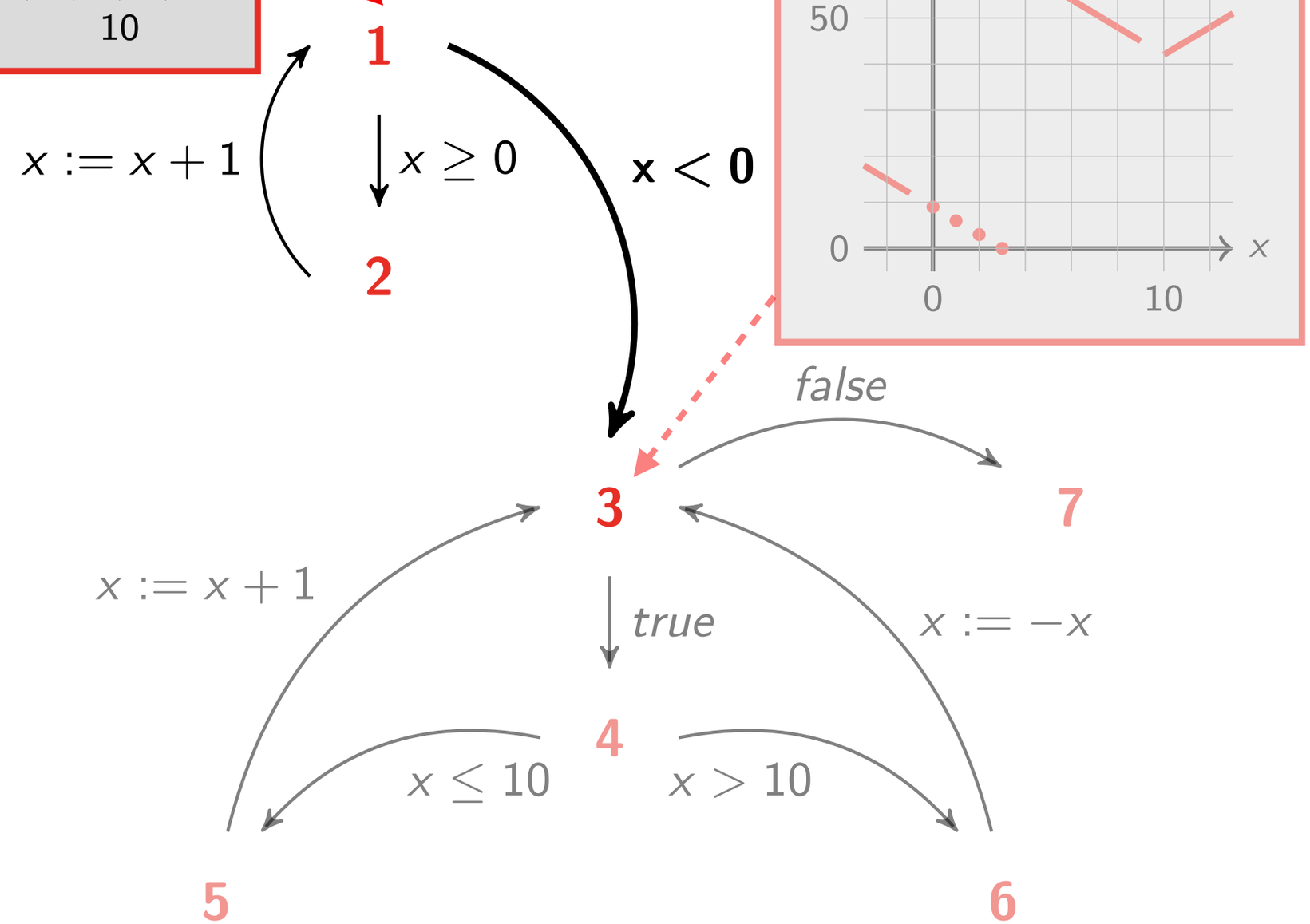
    [5] $x := x + 1$

  else

    [6] $x := -x$

od [7]



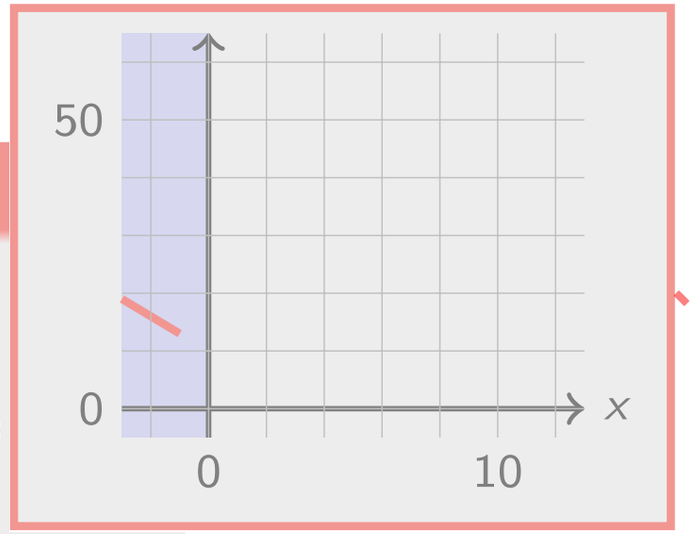


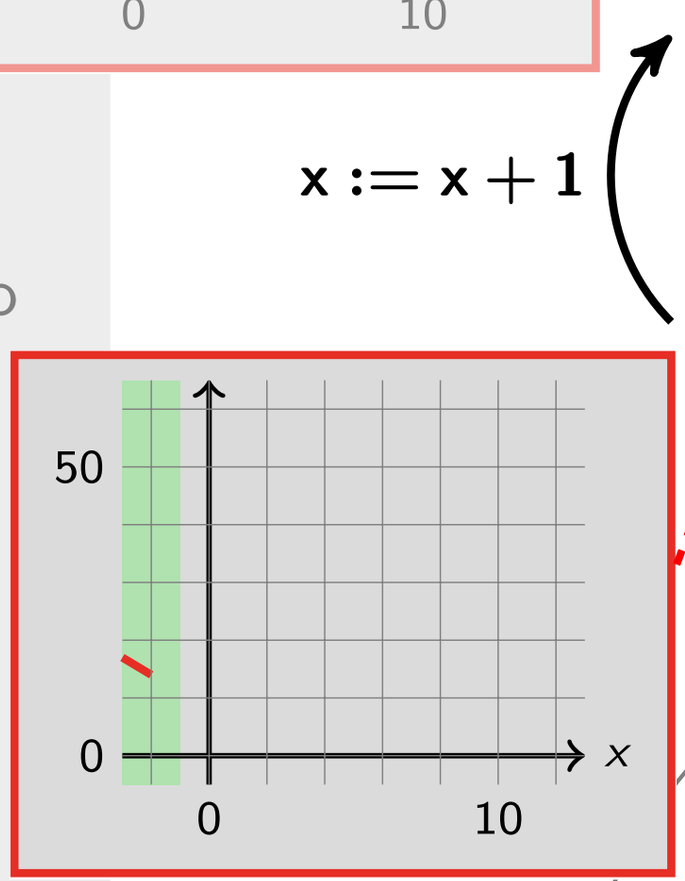
Recurrence Property

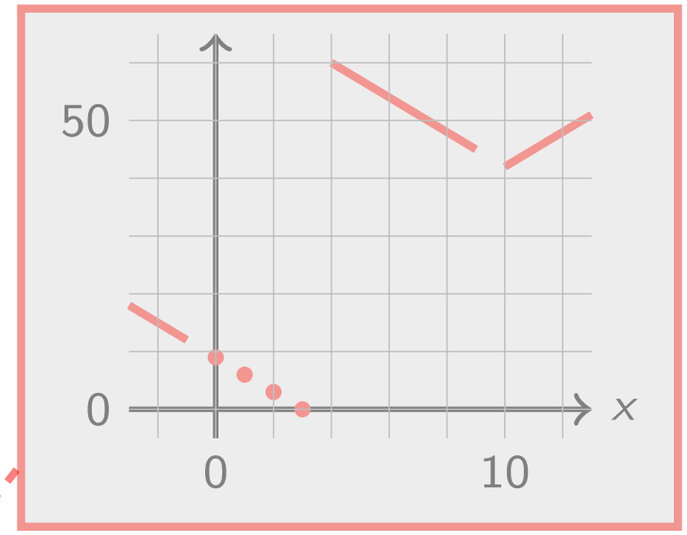
$\square\diamond\ x = 3$

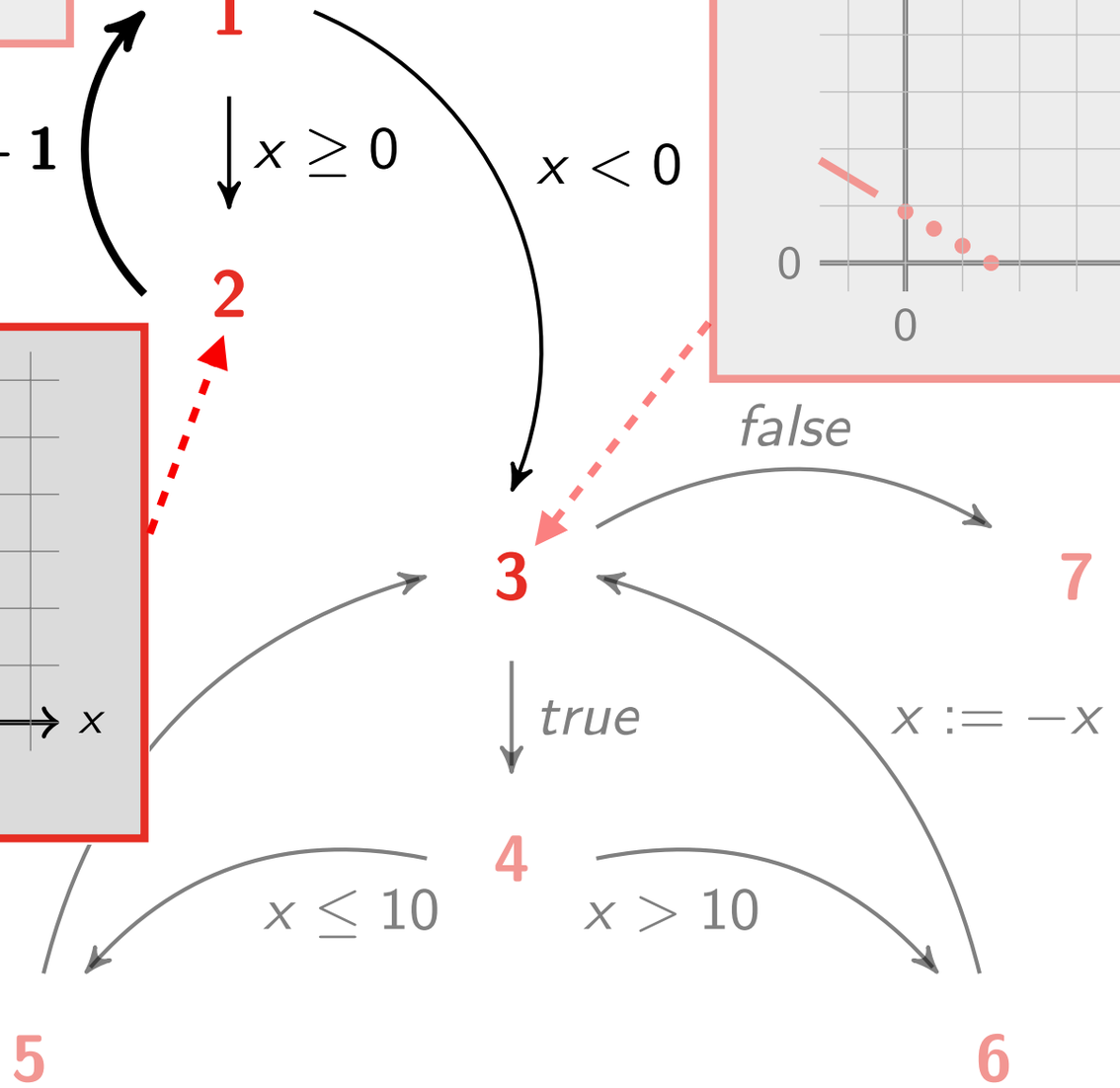
Example

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









Recurrence Property

$\square\diamond\ x = 3$

Example

int : x, y

[1] while $(x \geq 0)$

  [2] $x := x + 1$

od

[3] while (true) do

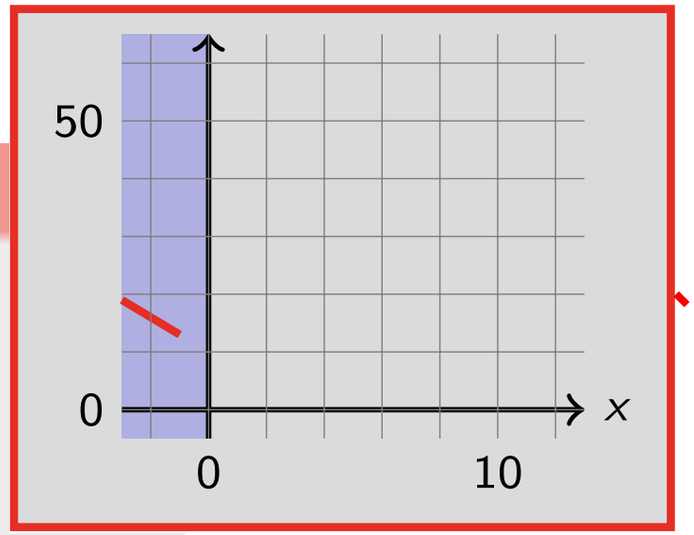
  [4] if ($x \leq 10$)

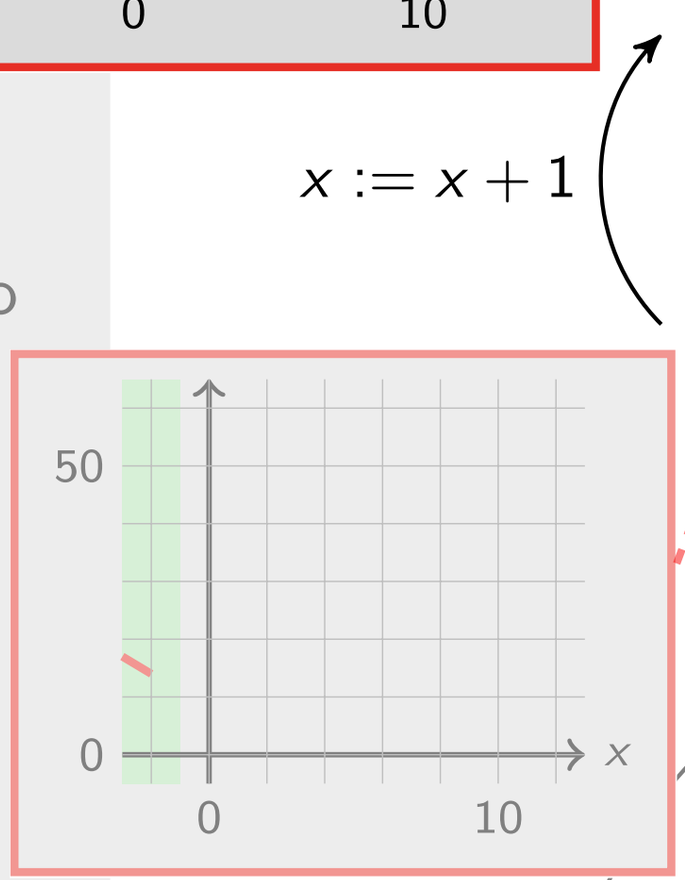
    [5] $x := x + 1$

  else

    [6] $x := -x$

od [7]





Recurrence Property

$\square\diamond\ x = 3$

Example

int : x, y

while [1] $(x \geq 0)$

  [2] $x := x + 1$

od

while [3] (true) do

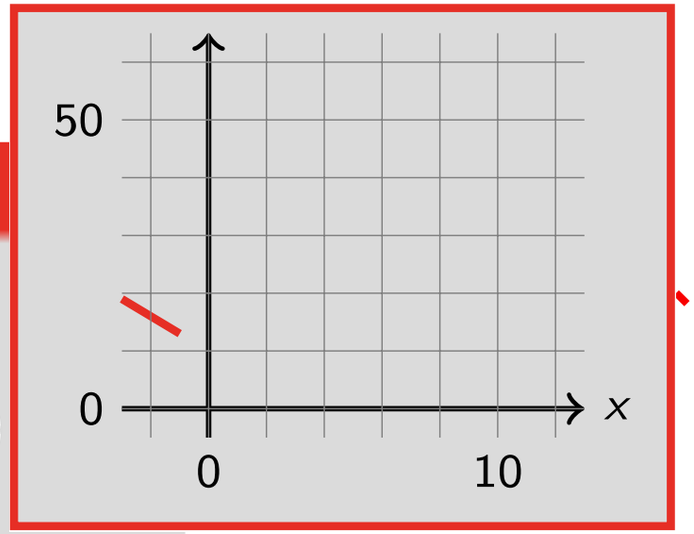
  if [4] ($x \leq 10$)

    [5] $x := x + 1$

  else

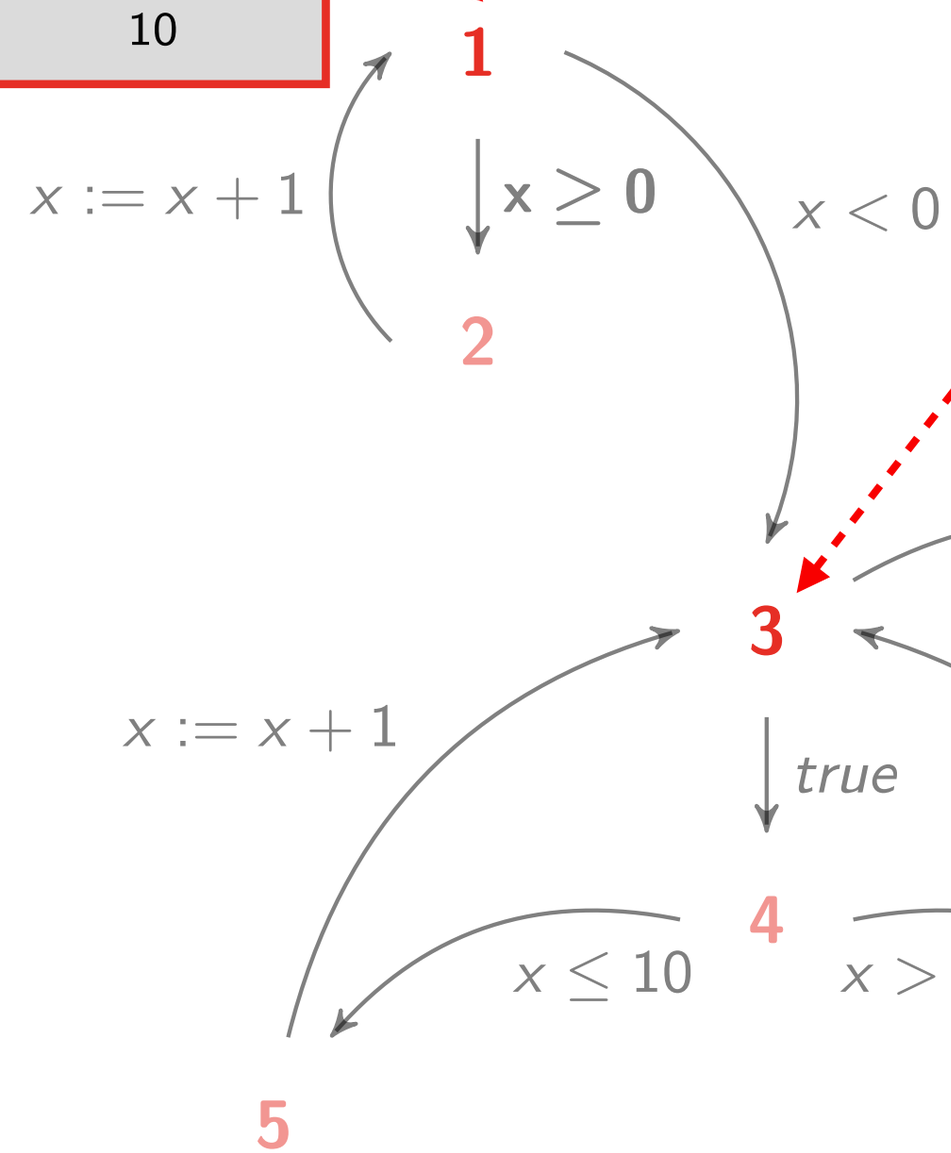
    [6] $x := -x$

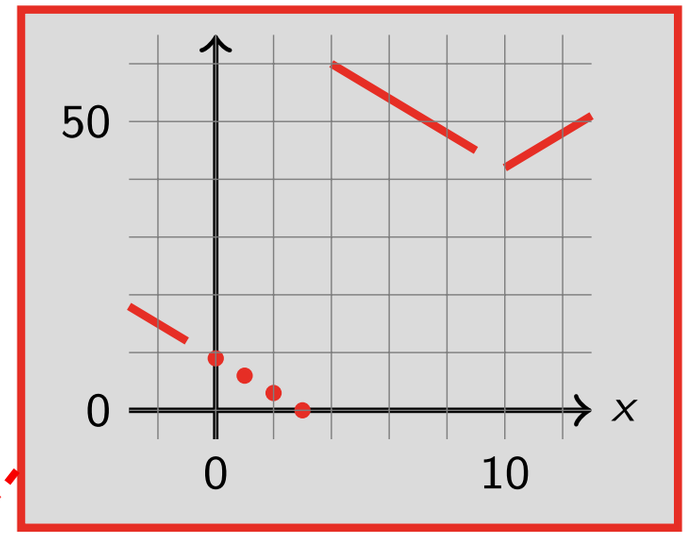
od[7]



Recurrence Property

$\square\Diamond\ x = 3$





Example

int : x, y

while [1] $(x \geq 0)$

  [2] $x := x + 1$

od

while [3] (true) do

  if [4] ($x \leq 10$)

    [5] $x := x + 1$

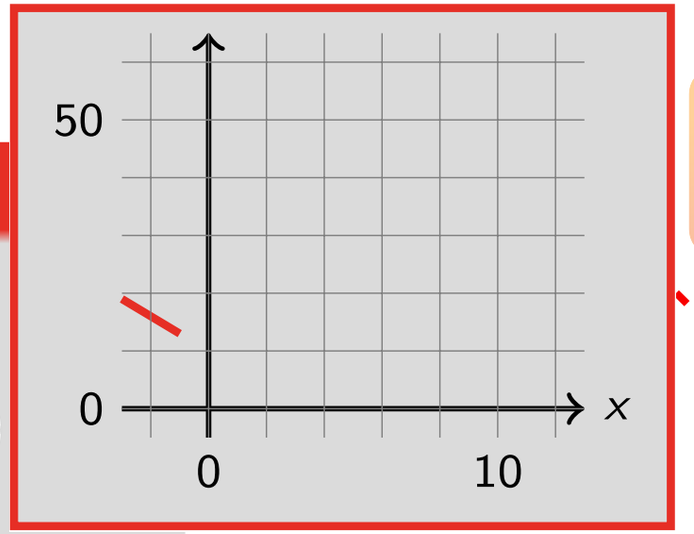
  else

    [6] $x := -x$

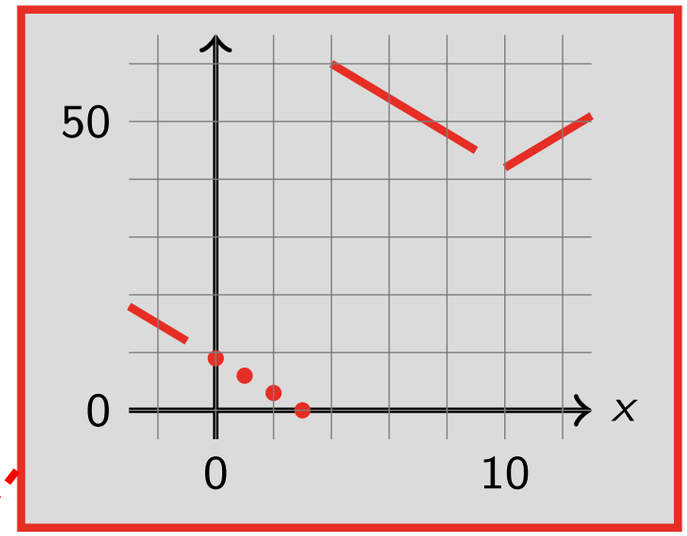
od [7]

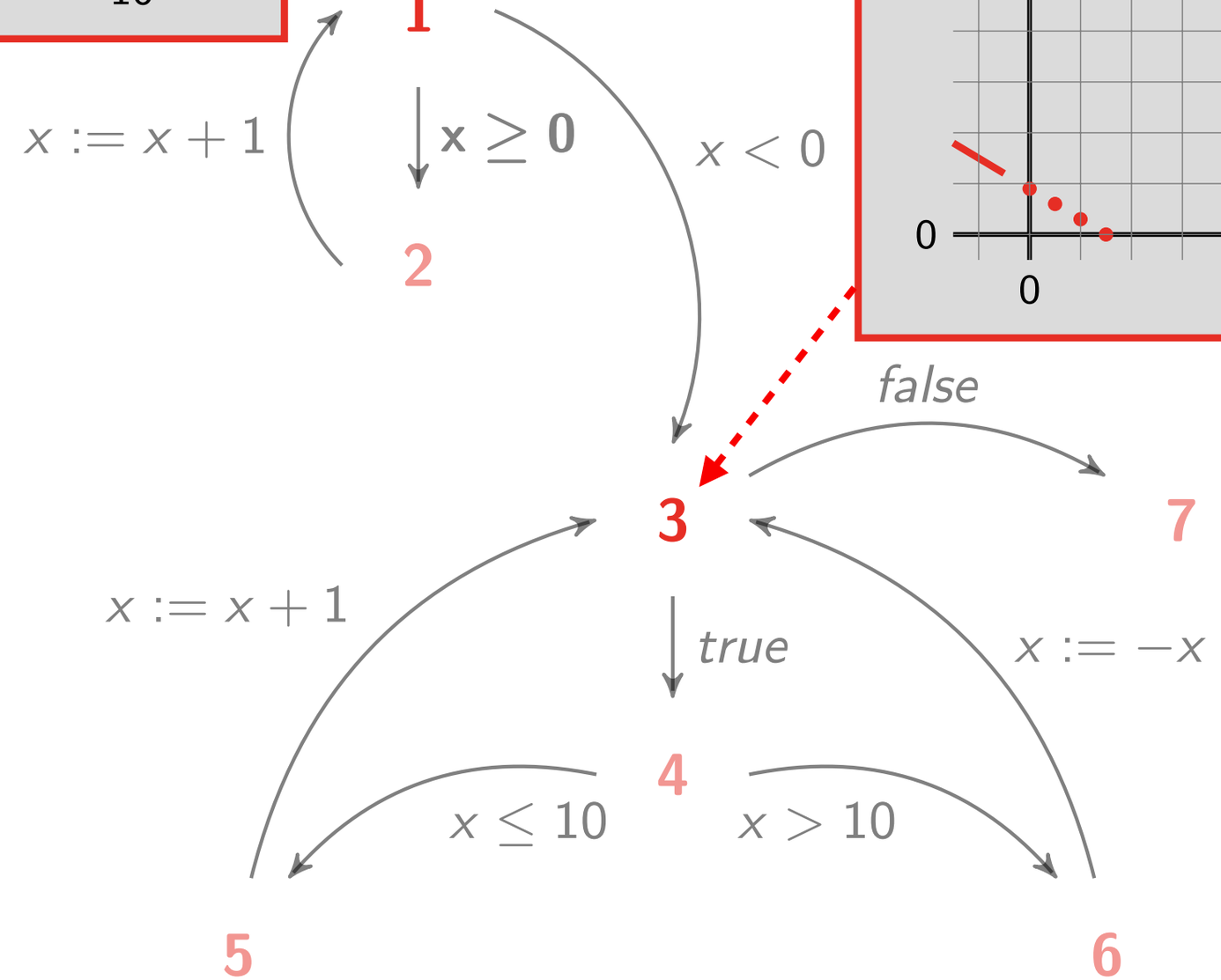
Recurrence Property

$\square\Diamond\ x = 3$



the analysis gives $x < 0$ as
sufficient precondition





Dual Widening

Definition

Let $\langle \mathcal{D}, \sqsubseteq \rangle$ be a poset. A *dual widening* $\bar{\nabla} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ obeys:

(1) for all element $x, y \in \mathcal{D}$, we have $x \sqsupseteq x \mathbin{\bar{\nabla}} y$ and $y \sqsupseteq x \mathbin{\bar{\nabla}} y$

(2) for all decreasing chains $x_0 \sqsupseteq x_1 \sqsupseteq \cdots \sqsupseteq x_n \sqsupseteq \cdots$, the chain

$$y_0 \stackrel{\text{def}}{=} x_0 \qquad y_{n+1} \stackrel{\text{def}}{=} y_n \mathbin{\bar{\nabla}} x_{n+1}$$

is ultimately stationary

Dual Widening

Definition

Let $\langle \mathcal{D}, \sqsubseteq \rangle$ be a poset. A *dual widening* $\bar{\nabla} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ obeys:

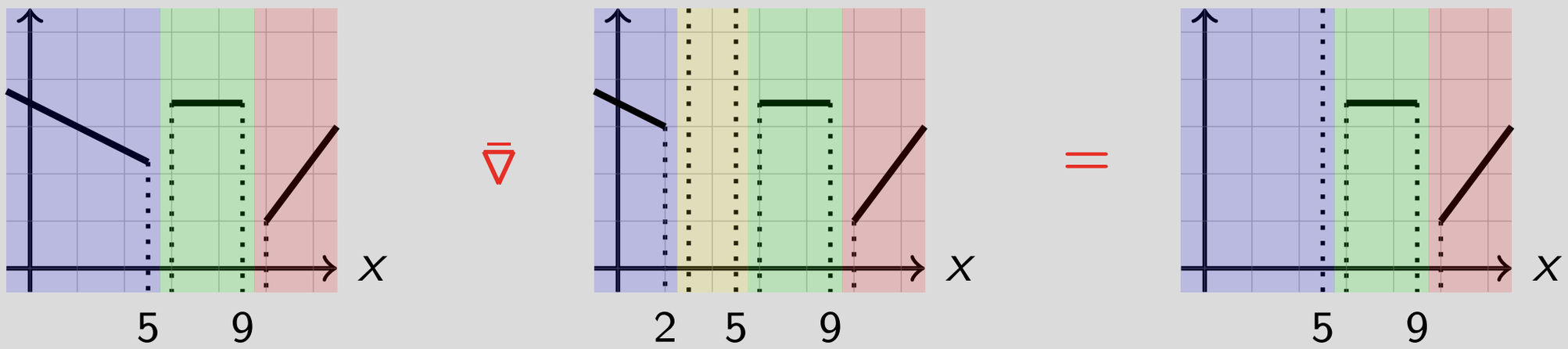
(1) for all element $x, y \in \mathcal{D}$, we have $x \sqsupseteq x \mathbin{\bar{\nabla}} y$ and $y \sqsupseteq x \mathbin{\bar{\nabla}} y$

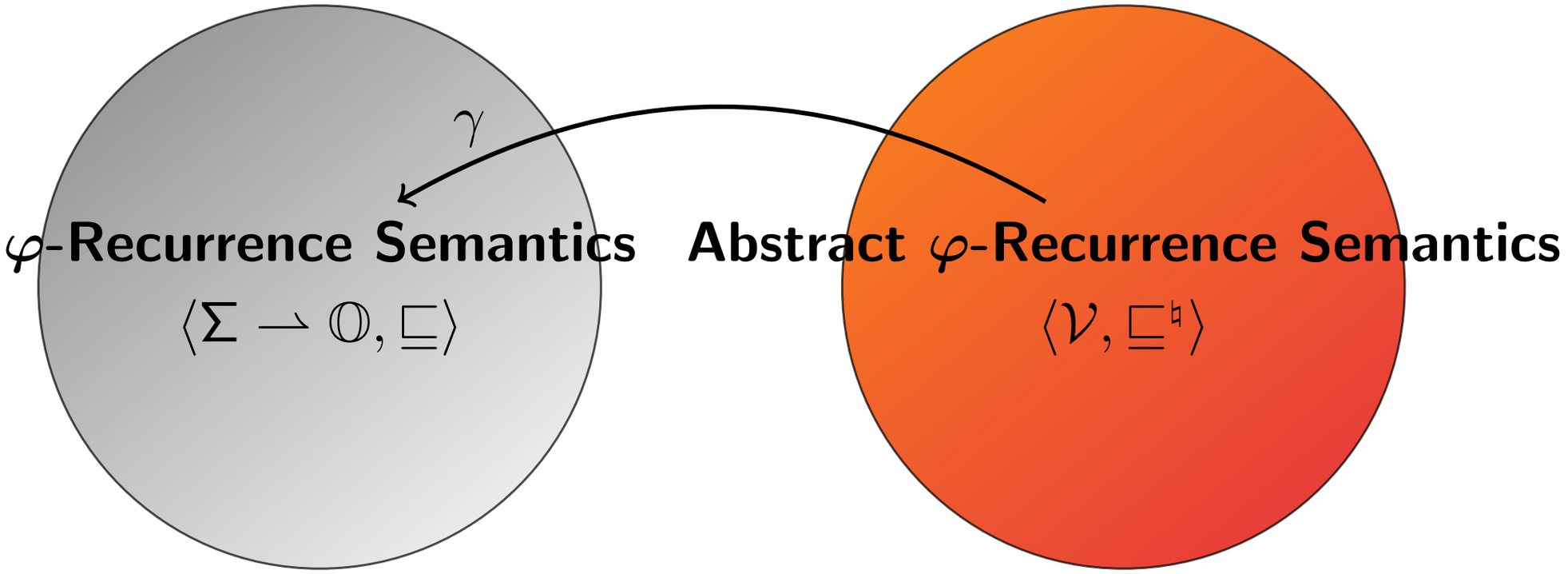
(2) for all decreasing chains $x_0 \sqsupseteq x_1 \sqsupseteq \cdots \sqsupseteq x_n \sqsupseteq \cdots$, the chain

$$y_0 \stackrel{\text{def}}{=} x_0 \qquad y_{n+1} \stackrel{\text{def}}{=} y_n \mathbin{\bar{\nabla}} x_{n+1}$$

is ultimately stationary

Example





Theorem (Soundness)

the abstract φ-recurrence semantics is **sound**
to prove the recurrence property $\Box\Diamond\varphi$

Peterson's Algorithm

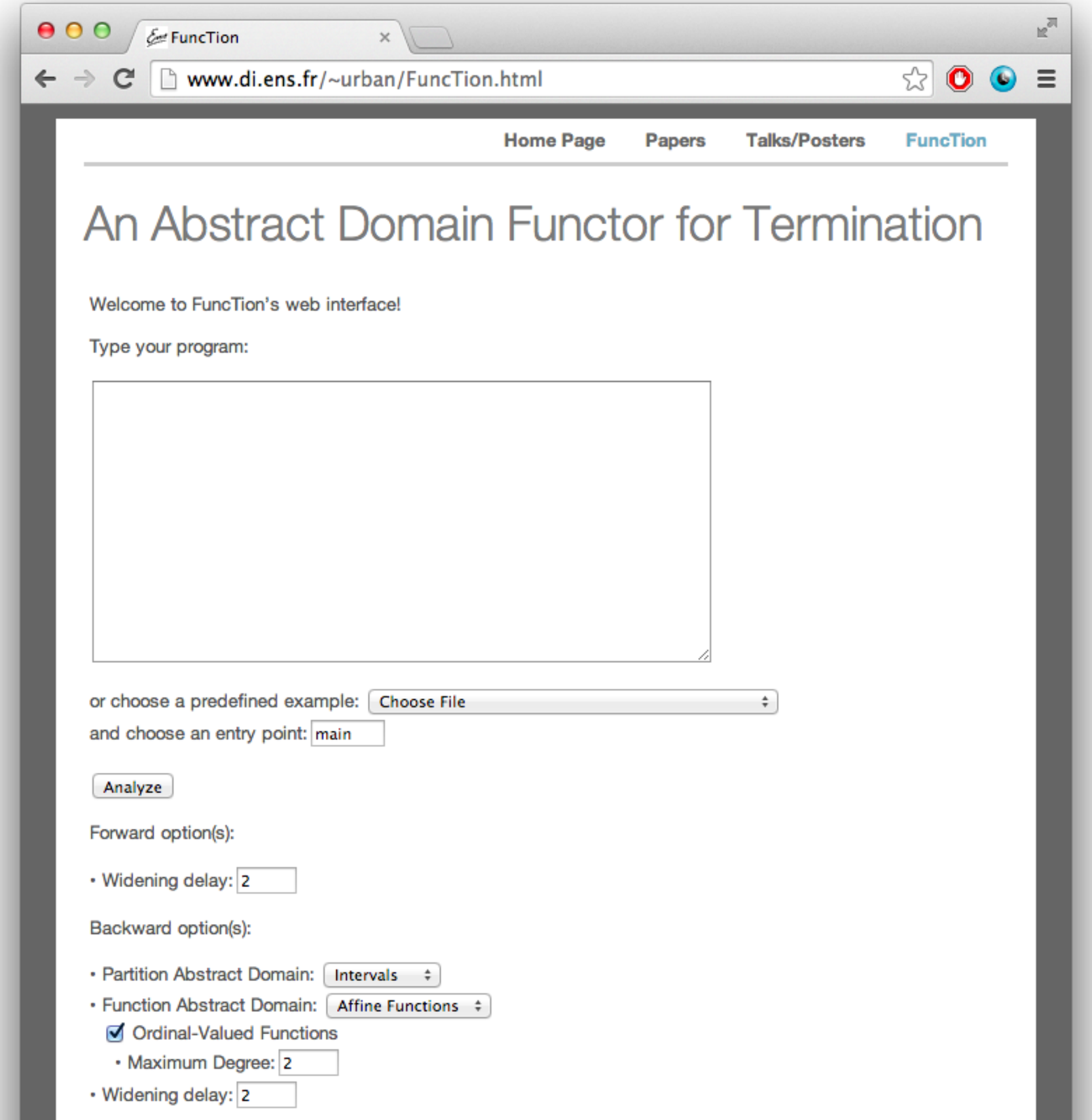
Example

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

$$\left[\begin{array}{l}\text{while } {}^{1}(\ true\ ) \text{ do}\\ \quad {}^{2}flag_1 := 1\\ \quad {}^{3}turn := 2\\ \quad \text{await } {}^{4}(\ flag_2 = 0 \vee turn = 1\ )\\ \quad {}^{5}\textsc{critical\_section}\\ \quad {}^{6}flag_1 := 0\end{array}\right] \parallel \left[\begin{array}{l}\text{while } {}^{1}(\ true\ ) \text{ do}\\ \quad {}^{2}flag_2 := 1\\ \quad {}^{3}turn := 1\\ \quad \text{await } {}^{4}(\ flag_1 = 0 \vee turn = 2\ )\\ \quad {}^{5}\textsc{critical\_section}\\ \quad {}^{6}flag_2 := 0\end{array}\right]$$

Recurrence Property

$\square\diamond$ 5 : true


FuncTion
www.di.ens.fr/~urban/FuncTion.html
Home Page
Papers
Talks/Posters
FuncTion
An Abstract Domain Functor for Termination
Welcome to FuncTion's web interface!
Type your program:
or choose a predefined example:
Choose File
and choose an entry point:
main
Analyze
Forward option(s):
• Widening delay:
2
Backward option(s):
• Partition Abstract Domain:
Intervals
• Function Abstract Domain:
Affine Functions
Ordinal-Valued Functions
• Maximum Degree:
2
• Widening delay:
2

- implemented in **OCaml**
- ad-hoc parser generated using **Menhir**

  `http://cristal.inria.fr/~fpottier/menhir/`
- abstract domains implemented on top of the **APRON library**

  `http://apron.cri.ensmp.fr/library/`
- participating to **SV-COMP 2014** (demo) and **SV-COMP 2015**

Reduction to Safety

- Biere & Artho & Schuppan - *Liveness Checking as Safety Checking* (ENTCS 2002)
- Bradley & Somenzi & Hassan & Zhang - *An Incremental Approach to Model Checking Progress Properties.* (FMCAD 2011)

Reduction to Fair Termination

- Podelski & Rybalchenko - *Transition Predicate Abstraction and Fair Termination* (POPL 2005)
- Cook & Gotsman & Podelski & Rybalchenko & Vardi - *Proving that Programs Eventually do Something Good* (POPL 2007)

Abstract Interpretation

- Massé - *Property Checking Driven Abstract Interpretation-Based Static Analysis* (VMCAI 2003)

Conclusions

- family of **abstract domains** for proving **guarantee** and **recurrence** program properties
  - piecewise-defined ranking functions
  - backward analysis
  - sufficient preconditions for the program properties

Future Work

- express and handle **fairness** properties
- other **liveness** properties
  - **persistence properties** $\Diamond\Box\varphi$
  - **existential** liveness properties

Thank You!