SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Towards the Verification of Top-Down Solvers

Alexandra Graß

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Towards the Verification of Top-Down Solvers

Verifizierung von Top-Down-Lösern

Author:	Alexandra Graß
Supervisor:	Prof. Dr. Helmut Seidl
Advisor:	Sarah Tilscher, Yannick Stade
Submission Date:	June 15, 2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, June 15, 2024 Alexandra Graß

Acknowledgments

Finishing this thesis was a non-trivial task and would not have been possible without the support of numerous people. I would like to thank my advisors for patiently answering every 'Verständnisfrage' there ever was, many interesting discussions and their valuable feedback on every stage of my work. I want to thank Helmut Seidl for creating a chair where people interested in formal methods are welcome and appreciated. My thanks extend to Michael Petter and Tobias Nipkow, who sparked my interest in abstract interpretation through their outstanding teaching.

I want to thank Alexis Engelke, for shaping my understanding of valuable didactics and a vision of academia and his continous trust in me and my competencies. I would like to thank my former tutor colleagous for revitalizing my post-pandemic life at and around university, especially Niklas Binder, Lukas Heine, Tobias Kamm, Benjamin Rickels, Janez Rotman and Christian Zimmerer. I want to thank the fellow members of a group called 'Formaler Haufen', that is, Dominik Okwieka, Alexander Schlenga and Matthias Stübinger, for interesting conversations about formal methods and everything under the sun. I would like to thank those that cheered me on along the way, including Mariem Aboushawareb and numerous other friends. I am grateful for all the members of 'Kunst und Spiel n.e.V.' and the journey that led to its founding.

I want to thank my family for their ongoing support throughout the years, in particular my parents and grandparents, who gave me the means to enjoy rather than endure education; my cousins Julia Bader and Johanna Schneider, who encouraged me to follow my own path; Timon Giese, for approaching computer science from a different angle and regularly questioning my entrenched beliefs, as well as taking on our daily chores in times my thoughts are occupied elsewhere; and my sister Theresa Graß, for firmly believing in me and my abilities in a way that has always been incomprehensible to me. I am incredibly proud of you.

Finally, I would like to express my respect for two outstanding women, namely Radhia Cousot and my grandmother Marieluise Baur. Both of them pursued and succeeded with their interest in science – at a time when it was unfortunately not yet a matter of course. I hope that many more will follow your example.

Abstract

Static analysis is indispensable in modern computer science, as it provides a powerful instrument to optimize code and verify its safety. Generic solvers translate problem statements to systems of equations, which are then solved by means of a fixpoint engine. With regard to termination properties, certain problem domains necessitate an acceleration of the fixpoint iteration through widening and narrowing. Corresponding implementations, however, either guarantee partial correctness of results or termination. Fulfilling both qualities in a single algorithm is non-trivial. The plethora of concepts and ongoing discussion about their advantages and disadvantages reduces confidence in static analysis tools implementing widening and narrowing. In this thesis, we therefore provide a machine-checked verification of the partial correctness of the warrowing TD.

The top-down solver TD is a local generic fixpoint engine. It is characterized by a recursive evaluation strategy in which right-hand sides are considered as black boxes. The state of the solver is administered on the fly. This includes the tracking of dependencies between unknowns, but likewise the identification of suitable iterations for an application of widening and narrowing. Our work extends the verification of the top-down solver by Stade, Tilscher, and Seidl [44]. First, we adapt the fixpoint engine such that the warrowing operator is applied to a dynamically computed subset of fixpoint iterations. Warrowing combines the techniques of both widening and narrowing into a single update operator. We continue by identifying relevant properties of the warrowing operator. Ultimately, we proof that the warrowing TD is sound, i.e., on termination, the solver computes partial post-solutions. Our verification is generic in that it does not impose any constraints on the monotonicity of right-hand sides. The proof is written using Isabelle/HOL and is available online [18].

Contents

1 Introduction

Static analysis is omnipresent in modern software engineering [45]. Linters detect errors in syntax or program logic, enforce coding standards and conventions, and reveal vulnerabilities. Compilers perform extensive optimizations such as constant propagation and dead-code elimination, hence significantly improving the performance of the resulting executables. Due to its key role in software development, it is essential that static analysis provides reliable, i.e., correct results. Simultaneously, the analysis should be performant and compute precise results in a finite, preferably small amount of time.

A formal approach to static analysis is *abstract interpretation*, a mathematical concept coined by P. Cousot and R. Cousot [14]. Instead of addressing problem statements in the *concrete domain* of a programming language specification, they are abstracted, i.e., mapped to an *abstract domain*. If chosen appropriately, abstract interpretation can yield sufficiently accurate results while considerably reducing computational effort. A short motivational example is adapted from Sintzoff [43]:

Example 1: Let $\mathbb{D}_c = \mathbb{Z}$ be the concrete domain of integers and $\mathbb{D}_a = \{(-), (+), (\pm)\}$ an abstract domain of signs. Then $\alpha\colon \mathbb{D}_c \to \mathbb{D}_a$ is an abstraction function identifying the sign of an integer $i \in \mathbb{D}_c$. Arithmetic operations on values $a, b \in \mathbb{D}_a$ are intuitively defined according to the rule of signs:

$$a * b = \begin{cases} (+) & \text{if } a = b \text{ and } a \neq (\pm) \neq b \\ (-) & \text{if } a \neq b \text{ and } a \neq (\pm) \neq b \\ (\pm) & \text{otherwise} \end{cases} \quad \text{and} \quad a + b = \begin{cases} (+) & \text{if } a = b = (+) \\ (-) & \text{if } a = b = (-) \\ (\pm) & \text{otherwise} \end{cases}$$

Consider a function $f\colon \mathbb{D}_c \to \mathbb{D}_c$ where $f(x) = x^2 + 1$ and a conditional code segment `if` $f(c) < 0$ $\{\ldots\}$. Assume $c \in \mathbb{Z}$ is a constant supplied by the user and therefore unknown at compile time. Still, the compiler may identify the code block as *dead code*, as $f(c) \geq 0$ holds: For both $\alpha(c) = (+)$ and $\alpha(c) = (-)$, respectively, above calculation simplifies to $\alpha(c) * \alpha(c) + (+) = (+) + (+) = (+)$. The abstract interpretation of a function $g\colon \mathbb{D}_c \to \mathbb{D}_c$ defined by $g(x) = x^2 - 1$, on the other hand, yields the inconclusive result $\alpha(c) * \alpha(c) + (-) = (+) + (-) = (\pm)$.

In a static program analysis based on abstract interpretation, problem statements can be modeled by means of constraints [15]: Based on the properties of interest, an abstract domain is chosen such that it adequately describes the state of a program execution (*value*) at relevant program points. The state or value of a concrete program point x is generally *unknown*, but can be derived by considering the state at a subset of other program points. See, e.g., Example 1: The sign of an integer expression can be computed with respect to the sign of its subexpressions. When regarding the derivation rules thus obtained as right-hand sides f_x, the static analysis is reduced to *solving a system of equations* $x = f_x$. This approach abstracts the analysis from details such as the concrete program syntax. Corresponding tools are therefore referred to as *generic solvers*.

Solutions of equation systems are commonly obtained by applying a *fixpoint engine* that iteratively approximates a solution. The thesis at hand focuses on a current implementation [46, 44] of the top-down solver TD [10, 32], a fixpoint engine that is at the heart of static program analysis tools such as Goblint [48] and Ciao [21, 23]. Originally targeted at Prolog [33, 31], the solver was later adapted to other programming languages and paradigms [22, 20, 29, 48, 41]. In comparison to fixpoint algorithms like round-robin [19] and worklist iteration [27], the top-down solver TD is a *local* solver [16]. Instead of determining dependencies through a global, preliminary analysis, they are detected on-the-fly [42, 46]. In consequence, the solver only evaluates right-hand sides of unknowns that contribute to the overall query [16].

A fixpoint iteration is guaranteed to terminate iff Kleene's sequence is finite [28, 14]. However, this proposition does not apply to a number of abstract domains used in static analysis, as they span possibly infinite ascending or descending chains. Examples of such domains include intervals [14] and octagons [30]. This deficiency was already identified in the early days of abstract interpretation and addressed by P. Cousot and R. Cousot with the introduction of *widening and narrowing* [14, 13]. Conceptually, the solving stage is now divided into two seperate fixpoint iterations [12, 13]: In a first phase, the solver computes a post-solution[1] by means of an accelerated ascending Kleene fixpoint iteration. Here, precision of results is deliberately traded for an increase of performance and guaranteed termination. In an attempt to recover some of the precision lost, a widening phase is usually complemented with a subsequent narrowing phase. This additional fixpoint iteration covers a descending chain of post-solutions – and is hence optional as it only improves results already valid.

Targeted at solvers within Goblint, Apinis et al. [3, 2] and Amato et al. [1] later propose a refinement of the previously global switch from widening to narrowing phase. True to the concept of a local solver, their algorithms determine the appropriate phase on a

[1]That is, an overapproximation of the least fixpoint

local, per-variable basis. Additionally, the acceleration mechanism is only applied at a minimal subset of program points, resulting in an improvement of precision [6].

Whereas the above mentioned implementations can guarantee both termination[2] and partial correctness for monotonic systems of equations [36, 1, 2], this is not necessarily the case for non-monotonic right-hand sides. Corresponding systems of equations occur in a variety of analyses, including interprocedural analysis [15]. We observe that appropriate solution strategies can be divided into two general types of approaches, each with a different focus: By combining widening and narrowing in a dynamic *warrowing operator*, fixpoint engines like the top-down solver TD compute sound results [3, 1, 2], but may not terminate even in the presence of seemingly simple systems of equations [42]. In contrast, an algorithm enforcing a strict succession of single widening by single narrowing phases guarantees termination, but may return results that are not a post-solution. Instead, computed assignments provide a post-solution of the lower monotonization of the system [40, 42] – a slightly weaker claim.

The plethora of concepts and ongoing discussion about their advantages and disadvantages reduces confidence in static analysis tools implementing widening and narrowing. To encourage the use of efficient fixpoint engines even in safety-critical applications, we therefore provide a machine-checked verification of the TD extended by widening and narrowing. With a focus on soundness [3, 1, 2], we settle on an implementation based on warrowing and prove its partial correctness. Our proof builds on the verification of the solver TD by Stade et al. [44] and is conducted using Isabelle/HOL [37].

The thesis is structured as follows: In Section 2.1, we start by defining systems of equations and their (post-)solutions, as well as the dependency relation and partial correctness of assignments. We then continue by presenting adequate data structures and the algorithm that constitutes a minimal (plain) version of the TD in Section 2.2 and 2.3. In preparation for the upcoming inductive proof, Section 2.4 establishes the notion of a computation trace. Section 2.5 focuses on an extension of the plain TD with memoization. The background chapter concludes with an extensive introduction to widening and narrowing in Section 2.6. The core of our work is located in Chapter 3: There, we first provide an implementation of the warrowing TD, accompanied by a detailed explanation of the dynamic management of potential warrowing points. Section 3.1 covers the derivation of an invariant and an update relation that both characterize the solver TD. We proceed by analyzing the warrowing operator in Section 3.2 and highlight some of its properties. Ultimately, key aspects of our proof implemented in Isabelle/HOL [18] are outlined in Section 3.3. Our results are then critically discussed in Section 3.4 and compared to related work in Chapter 4.

[2]Assuming ascending and descending chains are finite and only finitely many unknowns are encountered.

2 Background

Let S, T be partially ordered sets. Given arbitrary mappings $f\colon X \to Y$, $g\colon S \to T$ and $h\colon Z \to \mathcal{P}(Z)$ where $\mathcal{P}(M)$ is the powerset of a set M, we introduce the following definitions and notations:

purity	A function f *is pure* if its evaluation is without side effects and the result solely depends on f's input arguments.
monotonicity	g *is monotonic* if for arbitrary $x, y \in S\colon\ x \le y \implies g\,x \le g\,y$
$\top_S$	*Top*, a value $\top_S \in S$ where $x \le \top_S$ for all $x \in S$
$\bot_S$	*Bottom*, a value $\bot_S \in S$ where $\bot_S \le x$ for all $x \in S$
$x \sqcup y$	Pairwise *supremum* or *join* of two values x and y, $x, y, z \in S$, where $x \le x \sqcup y,\ y \le x \sqcup y$ and $\forall x.\, y \le x \implies z \le x \implies y \sqcup z \le x$
$x \sqcap y$	Pairwise *infimum* or *meet* of two values x and y, $x, y, z \in S$, where $x \sqcap y \le x,\ x \sqcap y \le y$ and $\forall x.\, x \le y \implies x \le z \implies x \le y \sqcap z$
$h^+\colon Z \to \mathcal{P}(Z)$	The *transitive closure of* h maps arbitrary $x \in Z$ to the minimal set h^+x, for which both $h\,x \subseteq h^+x$ and $\forall y \in h^+x.\, h\,y \subseteq h^+x$ holds.
$h^*\colon Z \to \mathcal{P}(Z)$	The *transitive and reflexive closure*, defined as $h^*x := \{x\} \cup h^+x$
$f \oplus \{x \mapsto y\}$	An *updated mapping*, where $(f \oplus \{x \mapsto y\})\,x = y$ and $\forall x' \neq x.\ (f \oplus \{x \mapsto y\})\,x' = f\,x'$
g_0	The *empty mapping* $g_0\colon X \to T$, defined by $x \mapsto \bot_T$ for all $x \in X$

When analyzing programs, a certain ambiguity arises regarding the word variable. A variable can be a variable of the program – called *program variable* – or a variable in the context of equations. To avoid confusion, we refer to the latter as *unknowns*.

2.1 Systems of Equations

Following the idea of a generic solver, datatypes are chosen as unrestricted as possible. Assume that $\mathbb{D}$ is a domain of (abstract) values. A system of equations is commonly defined as a collection of equations

$$x = f_x\,(x_1, x_2, \ldots, x_n), \tag{1}$$

where f_x is called the *right-hand side* to an unknown $x \in \mathbb{D}$ and $x_i \in \mathbb{D}$ are parameters the computation of f_x depends on. Our technical approach, however, makes it necessary to distinguish between the concepts of *unknowns* and *their values*. We therefore introduce $\mathcal{U}$ as the set of unknowns and a complementary *assignment* $\sigma\colon \mathcal{U} \rightarrow \mathbb{D}$, which maps unknowns $x \in \mathcal{U}$ to a value $d_x \in \mathbb{D}$. Equations as presented above are now rephrased:

$$x \mapsto f_x \tag{2}$$

In this notation, a system of equations is considered to be a function mapping unknowns $x \in \mathcal{U}$ to their right-hand side $f_x : (\mathcal{U} \rightarrow \mathbb{D}) \rightarrow \mathbb{D}$. The functional f_x expects a single parameter of type $\mathcal{U} \rightarrow \mathbb{D}$. By taking an assignment σ as input, f_x is able to represent complex dependencies between different unknowns and their currently assigned values. From here on, we assume equation systems to comply with the notation in (2). Yet for the sake of readability, examples will be supplied in the more familiar style of (1).

Conceptually, right-hand sides f_x of an equation are considered black boxes. Given an assignment σ, f_x may always be evaluated. Further meta information such as the equations' exact definition, on the other hand, is not assumed to be available to the solver. Motivated by the findings of Hofmann et al. [25], we require functionals f_x to be pure (see also Section 2.2). But whereas this is quite common for other solvers applying widening and narrowing, no restriction is made with respect to monotonicity.

Example 2: Let $\mathbb{D} = \mathbb{N}_0$ and $\mathcal{U} = \{x, y\}$. Then the system of equations defined by

$$\begin{aligned} x &= (\texttt{if}\ x < 100\ \texttt{then}\ y\ \texttt{else}\ 100) \\ y &= x + 1 \end{aligned}$$

is pure and monotonic. Intuitively, both equations are satisfied for $x = 100$ and $y = 101$.

Systems of equations usually encode constraints and dependencies between the various unknowns of the system. In an analysis using equation systems, a solution is therefore of great interest as it fulfills all the requirements specified. Formally, an assignment $\sigma^\star\colon \mathcal{U} \rightarrow \mathbb{D}$ is considered a *total solution*, if for all unknowns $x \in \mathcal{U}$:

$$f_x\,\sigma^\star \;=\; \sigma^\star\,x \tag{3}$$

This renders $\sigma^\star$ a fixpoint of the equation system – remember that equations f_x are defined as functionals depending on $\sigma^\star$. However, such a fixpoint or solution might not always exist. Take, for example, a system that consists of the single equation $x = x + 1$. In this case, there is no value for x in $\mathbb{D} = \mathbb{N}_0$ that ever satisfies the equation's constraints. Given such a system of equations, the solver TD and equivalent solvers do not terminate.

In essence, though, abstract interpretation performs an overapproximation of a program's state at (possibly abstract) program points. This raises the idea of overapproximating solutions as well. Consequently, we introduce the notion of a *post-solution*: Given a system of equations $x \mapsto f_x$ for unknowns $x \in \mathcal{U}$. Then, if for each $x \in \mathcal{U}$

$$f_x\, \tilde{\sigma}^\star \leq \tilde{\sigma}^\star\, x \tag{4}$$

holds, an assignment $\tilde{\sigma}^\star$ constitutes a post-solution. By choosing $\mathbb{D}$ such that there exists a top element $\top_\mathbb{D} \in \mathbb{D}$, every equation system admits at least one post-solution, namely the trivial post-solution $\forall x.\ x \mapsto \top_\mathbb{D}$.

Let $\mathcal{U}$ be infinite. Naturally, a solver trying to compute a total solution for such a system can never terminate. However, due to locality of reference in typical computer programs, it often suffices to determine a partial solution. Therefore, Stade et al. [44] start by defining the concept of dependencies: For an unknown $x \in \mathcal{U}$ and a mapping $\sigma\colon \mathcal{U} \to \mathbb{D}$, let the function $\mathsf{dep}\ x\, \sigma$ return the set of all unknowns whose value is effectively looked up in σ during the evaluation of $f_x\, \sigma$. All unknowns $y \in \mathsf{dep}\ x\, \sigma \subseteq \mathcal{U}$ thus denote *direct dependencies* of x.

Example 2 (continued): Let $\sigma_1 = \{x \mapsto 0,\, y \mapsto 0\}$ and $\sigma_2 = \{x \mapsto 100,\, y \mapsto 0\}$. Then

$$\begin{aligned}\mathsf{dep}\ x\, \sigma_1 &= \{x, y\}\\ \mathsf{dep}\ x\, \sigma_2 &= \{x\}\end{aligned}$$

are the direct dependencies of unknown $x \in \mathcal{U}$ considering σ_1 and σ_2, respectively.

Based on this definition, an equation system is characterized as *recursive* with respect to σ if $x \in \mathsf{dep}^+\ x\, \sigma$ for any unknown $x \in U$. In other words: There exists a direct or indirect circular dependency within the system.

Ultimately, a mapping $\sigma_p^\star\colon \mathcal{U} \to \mathbb{D}$ is referred to as *partial solution* for a set $s \subseteq \mathcal{U}$, if:

$$\forall x \in s.\ \mathsf{dep}^*\ x\, \sigma_p^\star \subseteq s \wedge f_x\, \sigma_p^\star = \sigma_p^\star\, x \tag{5}$$

A *partial post-solution* is defined analogously. Trivially, a solution $\sigma^\star$ is likewise a partial solution $\sigma_p^\star$ for any chosen set s.

Example 2 (continued): Due to the first equation, the system is always recursive regardless of the chosen assignment. Whereas σ_2 is a partial solution for $s = \{x\}$, this is not the case for any set s containing y. Mapping σ_1 is neither a partial nor total (post-)solution.

By the introduction of a post-solution, the choice of a suitable domain is inevitably restricted, as $\mathbb{D}$ needs to be partially ordered. Additionally, $\mathbb{D}$ is required to provide a special element $\bot$, used as initial value for TD's fixpoint iteration. Even though an arbitrary choice of $\bot$ guarantees valid results [44], the solver will not necessarily compute a least solution [10]. In preparation for future verification, we therefore choose $\bot = \bot_{\mathbb{D}}$, i.e., $\bot$ to be the least element of $\mathbb{D}$. With regard to the definition of widening and narrowing operators in Section 2.6, the final constraint on the domain is the existence of a pairwise supremum operator $\sqcup$ and infimum operator $\sqcap$. Consequently and in accordance with previous work [3, 2, 42], we assume $\mathbb{D}$ to be a complete lattice.

2.2 Representation as Strategy Trees

So far, the right-hand sides of equations where presented as pure black-box functionals $f_x : (\mathcal{U} \to \mathbb{D}) \to \mathbb{D}$. Although gratifyingly general, this approach lacks structure on which to reason upon. We thus aim to specify equations in a fixed, that is, canonical form. In their work, Hofmann et al. [25] introduce the concept of *strategy trees* and prove the existence of an equivalent query-answer strategy for any pure functional. From now on, we assume right-hand sides of equations to be given as just such strategy trees (also called query-answer trees):

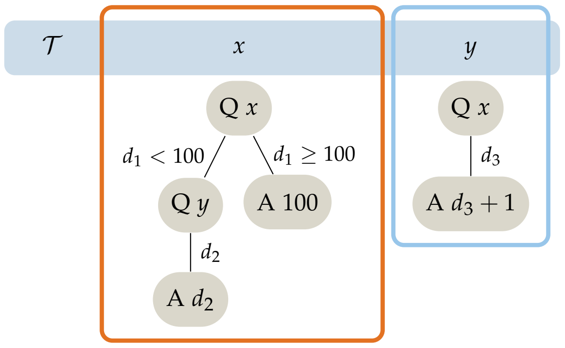
```
datatype ('x, 'd) query_answer_tree =
    Answer 'd
  | Query   'x   'd ⇒ ('x , 'd) query_answer_tree
```

Assume that, given the latest assignment σ', there is some appropriate way to query the current value $\mathtt{d}_\mathrm{y}$ of an unknown y. Then Query nodes `Query` y g implement the strategy of query-answer trees: After computing the value $\mathtt{d}_\mathrm{y}$, it is used to determine the child node that should be evaluated next. This is done by applying the strategy function g, which returns a node $(\mathtt{g}\,\mathtt{d}_\mathrm{y})$. Answer nodes `Answer d`, on the other hand, encode constant values $\mathtt{d} \in \mathbb{D}$. They serve as leaf nodes, i.e., result values of a strategy tree evaluated with the original state σ.

In graphical representations, we abbreviate Query nodes with the letter Q and Answer nodes with A. As will be seen below, repeated queries Q x of an unknown $x \in \mathcal{U}$

are not necessarily bound to return the exact same value. In concrete instances of query-answer trees, values are therefore referred to as d_i, where i is a unique identifier.

Example 2 (continued): The if-clause of right-hand side f_x is realized by two complementary strategies. Technically, there are $|\mathbb{D}|$ mappings or edges leaving a Query node. But generally, they can be parameterized and grouped in equivalence classes – see also d_3 in unknown y's strategy tree.



We continue by concretizing the definition of equation systems (see Section 2.1) with regard to strategy trees and fix a *system of equations* $\mathcal{T}$ as a forest, that is, a function $\mathcal{T}\colon \mathcal{U} \to (\mathcal{U}, \mathbb{D})$ `query-answer tree`. For each unknown $x \in \mathcal{U}$, the function $\mathcal{T}$ maps an unknown x to its right-hand side f_x. As will become apparent in Section 3.3, the recursive nature of query-answer trees allows for a convenient proof by induction.

With equation systems represented as a forest of strategy trees, the computation of a solution to the system can now be automated. The next section provides a first minimal implementation of the top-down solver TD.

2.3 The Top-Down Solver TD

In the following, we present the plain TD as introduced by Tilscher, Stade, et al. [46, 44]. Originating in the universal top-down fixpoint algorithm first described by Charlier and Van Hentenryck in 1992 [10], the solver TD has been frequently adapted and extended over time [16, 2, 42]. The plain TD in [44] is implemented by three mutually recursive functions: `eval`, `query` and `iterate`.

In the listings below, opaque code represents the core functionality of the solver. Efficiency is added by deploying memoization, displayed transparently. The basic idea behind this optimization can be summarized as follows: Already computed values $\mathtt{d_x} \in \mathbb{D}$ do not need to be recalculated iff they are still considered *stable*, i.e., $\mathtt{x} \in \mathtt{stable} \subseteq \mathcal{U}$. An unknown remains stable as long as it only depends on values $\mathtt{d_y}$ which did not change since $\mathtt{d_x}$ was last computed. In other words: It was not destabilized by a call to `destab y`. The dependency of unknowns on each other is dynamically tracked in a mapping `infl`: $\mathcal{U} \to \mathcal{P}(\mathcal{U})$. In Section 2.5, the memoization technique is discussed in more detail. For now, we focus on the core of the algorithm.

To begin with, function `eval` performs the evaluation of query-answer trees as discussed in Section 2.2. Given an unknown $\mathrm{x} \in \mathcal{U}$, (parts of) its right-hand side f_x represented as strategy tree `t` and the present assignment σ of values to unknowns. Additionally, let c be a set of already called values, $c \subseteq \mathcal{U}$. Then, in case tree `t` is of type `Answer d`, function `eval` behaves as expected: The traversal of the query-answer tree terminates and the computed value is returned, as well as the current assignment σ. Otherwise, tree `t` is of type `Query y g`. At first, function `eval` queries a suitable approximation of value $\mathtt{d}_\mathrm{y}$ by providing parameter σ as the mapping of preliminarily computed values for already queried unknowns. This value is then used to determine the subtree $\mathtt{g}\,\mathtt{d}_\mathrm{y}$ (that is, the subexpression of f_x) to be evaluated next. Note that the mutually recursive call to `eval` is performed using an updated version of σ. Potential changes on c occuring in `query` $\mathrm{y}\ c\ \sigma$, on the other hand, are discarded – a behavior that is common to all three of the mutually recursive functions.

```
eval x t c infl stable σ = case t of
    Answer d ⇒ (d, infl, stable, σ)
  | Query y g ⇒
      let (d_y, infl, stable, σ) = query x y c infl stable σ in
      eval x (g d_y) c infl stable σ
```

The implementation of `query` again differentiates two cases. Assume unknown x does not occur in the call stack so far, i.e., $\mathrm{x} \notin c$. To determine an appropriate value $\mathtt{d}_\mathrm{x}$ with respect to assignment σ, the solver TD starts a fixpoint iteration on x. Before descending into the iteration, x is explicitly added to the call stack. Alternatively, if x already is in the set of called unknowns c, `query` detects a circular dependency on x. A further descend into additional recursive calls would thus never terminate. To prevent such a behavior, the execution tree is pruned and the value of x is looked up in σ instead:

```
query y x c infl stable σ =
  let (d_x, infl, stable, σ) =
    if x ∈ c then
      (σ x, infl, stable, σ)
    else
      iterate x (c ∪ {x}) infl stable σ
  in (d_x, infl ⊕ {x ↦ infl x ∪ y}, stable, σ)
```

The actual solving takes place in `iterate`: Starting with a call to `eval`, the solver first determines the latest value $\mathtt{d}_\mathrm{x}$ of the right-hand side $\mathcal{T}\,\mathrm{x}$. Internally, the TD evaluates the equation system using the given assignment σ. Observe that in the basic version of `eval`, there is no need to explicitly pass x as parameter. If no change is found, that is, the evaluation of the unknown returned the same value as already stored in σ, TD reached a fixpoint and the value along with the associated mapping is returned. Else, assignment σ is updated such that it maps x to $\mathtt{d}_\mathtt{x}$ and the iteration is resumed.

```
iterate x c infl stable σ =
  if x ∉ stable then
    let (d_x, infl, stable, σ)
          = eval x (T x) c infl (stable ∪ {x}) σ in
    if σ x = d_x then
      (σ x, infl, stable, σ)
    else
      let (infl, stable) = destab x infl stable in
      iterate x c infl stable (σ ⊕ {x ↦ d_x})
  else
    (σ x, infl, stable, σ)
```

Lastly, function `solve` implements the top-level interface to the top-down solver. A call `solve` x solves the system of equations for an unknown $\mathtt{x} \in \mathcal{U}$, assuming execution terminates. In case $\mathtt{dep}\,\mathtt{x}\,\sigma_r \subset \mathcal{U}$, the returned assignment σ_r is only guaranteed to be a partial solution for unknowns $\mathtt{y} \in s \subset \mathcal{U}$. The set s (see also Equation 5) is implicitly computed by the plain TD on-the-fly. It includes at least all of unknown x's dependencies: $\mathtt{dep}\,\mathtt{x}\,\sigma_r \subseteq s$ [44]. If memoization is enabled, then σ_r is a partial solution for unknowns in `stable`– a stronger proposition, as $s \subseteq$ `stable` holds [44].

Remember that $\sigma_0 : \mathcal{U} \to \mathbb{D}$ is the empty mapping, i.e., the assignment σ_0 that maps every unknown in $\mathcal{U}$ to $\bot_\mathbb{D}$. The set of called unknowns c is initially empty but x. For completeness, we also introduce $\mathtt{infl}_0 : \mathcal{U} \to \mathcal{P}(\mathcal{U})$ as the empty influence mapping where $\bot_{\mathcal{P}(\mathcal{U})} = \{\}$. Ultimately, `solve` initializes a fixpoint iteration on x as follows:

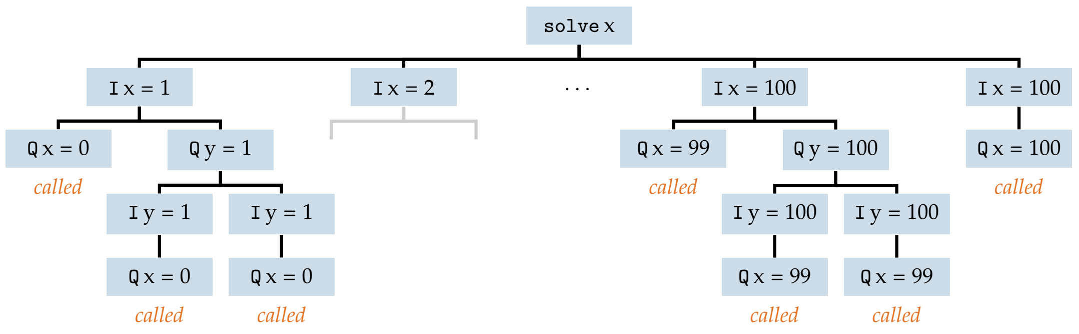
```
solve x = (let (d, _, _, σ) = iterate x {x} infl_0 {} σ_0 in (d, σ))
```

2.4 Computation Trace

To investigate the mechanisms that characterize the top-down solver, we will concentrate on its execution in more detail. A run of the solver TD is always deterministic. Given a system of equations as query-answer trees $\mathcal{T}$ and an unknown for which the equation system is to be (partially) solved. The execution of the solver with this specific input can then be described by a *computation trace*. A computation trace is a tree that models the mutual recursion. Each function call is represented as a node. Direct recursive calls are interpreted as iterations; accordingly, they are positioned in a horizontal row. Mutual calls, on the other hand, are placed on a new layer of the computation trace.

For the sake of visual clarity, some steps are abstracted: The actual strategy of individual equations is only implied by the solver's choice of queries. Function calls `eval (Answer d)` are omitted; instead, evaluation results are attributed to the parent node (i.e., the latest call of) `iterate`. This allows for a complete removal of all layers of `eval` nodes. Instead, `query` nodes[1] are appended directly to the preceding iteration. Additionally, we abbreviate `iterate` nodes with the letter `I` and `query` nodes with `Q`.

Recall Example 2. Calling `solve x` yields the computation trace below:



Beware of a few aspects: Even though the computed solution $\sigma_1 = \{x \mapsto 100, y \mapsto 100\}$ fulfills equation f_x, this is not the case for f_y. That is due to the fact that in the final iteration, x does not depend on y. Or, more precisely: $\texttt{dep}\, x\, \sigma_1 = \{x\}$. Results should thus always be interpreted with care and in compliance with dependencies reflected in the returned assignment. Moreover, the example's trace exposes a serious inefficiency of the plain TD: Even though non of the queried values below unknown y change (that is, the value of x), the solver constantly iterates y twice before recognizing its stabilization. In the next section, we therefore improve the performance of the plain TD by means of memoization.

---

[1]Not to be confused with Query nodes of query-answer trees.

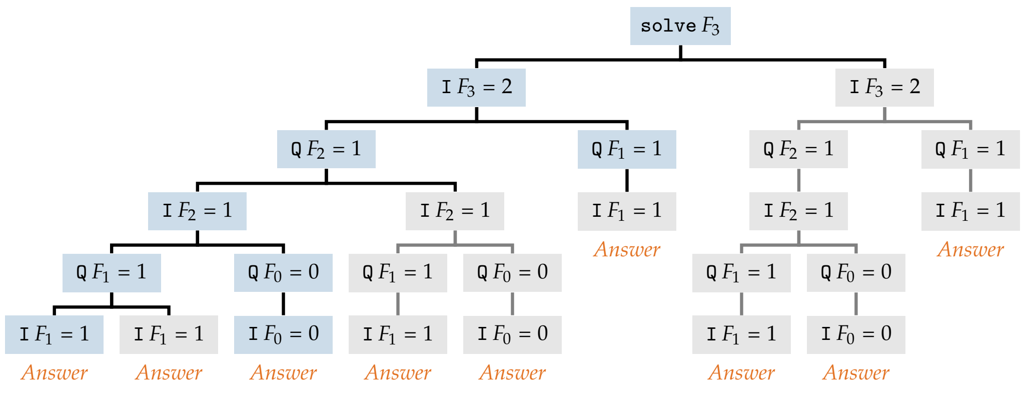
2.5 Memoization

Memoization describes an algorithm's ability to cache already computed values. Provided that none of the input values changed, partial results can be looked up, saving resource-intensive recalculations. A prime example of application is the calculation of recurrence relations like the Fibonacci sequence:

Example 3: Consider a system of equations where $\mathbb{D} = \mathbb{N}_0$ and $\mathcal{U} = \{F_n \mid n \in \mathbb{N}_0\}$. With $i \in \mathbb{N}_0 - \{0, 1\}$, the collection of equations is defined as:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \end{aligned}$$

Note that the number of unknowns as well as their defining right-hand sides is infinite.[2] In contrast to Example 2, the system is not recursive as there are no circular dependencies. Let $s = \{F_0, F_1, F_2, F_3, F_4, F_5\}$. Then $\sigma_p^\star = \{F_0 \mapsto 0,\ F_1, F_2 \mapsto 1,\ F_3 \mapsto 2,\ F_4 \mapsto 3,\ F_5 \mapsto 5\}$ is the system's unique partial solution for s. A call to `solve` F_3, on the other hand, triggers the following computation:



Redundant iterations and their sub-computations are marked in gray. For the problem statement at hand, the runtime of the plain top-down solver TD is exponential. Eliminating superfluous computations reduces the solver's performance to a linear runtime in $\mathcal{O}(n)$.

---

[2]There exists a closed form expression, known as Binet's formula. Nonetheless, the solver TD is unable to compute such an infinite solution: First, there is no initial largest unknown F_∞ to solve for, for which the returned partial solution includes all possible F_n, $n \in \mathbb{N}_0$. And secondly, each unknown in s has to be iterated at least once – an infinite computation that never terminates.

The introduction of an memoization mechanism is conducted with a few adjustments: To administer already calculated values, we establish a set `stable`. Upon initial evaluation of its strategy tree $\mathcal{T}$x, an unknown x is optimistically added to `stable`. If an unknown is then queried again later and found in `stable`, it is still valid – the value can therefore simply be looked up in in the provided assignment σ. The functionality described is implemented in `iterate`, see Listing `iterate` (ll. 2, 4, 10f.) above.

This economical approach is met with an active and comprehensive invalidation strategy. Assume that during iteration of an arbitrary unknown x, the memorized value σ x does not match the computed value $\mathtt{d}_\mathtt{x}$, i.e., the value $\mathtt{d}_\mathtt{x}$ does not constitute a fixpoint yet. We call this case the *Continue Case*. Before resuming iteration on x, the assignment σ is updated to reflect the unknown's recalculated value. However, the change of state must now also be propagated to `stable`, which is done in a helper function `destab` (c.f. Listing `iterate`, l. 8):

```
destab x infl stable =
  let f y (infl, stable) = destab y infl (stable − {y}) in
  fold f (infl ⊕ {x ↦ {}}, stable) (infl x)
```

The algorithm removes every unknown from `stable` whose value was calculated based on the value σ x, whether directly or transitively. To ascertain the exact interdependence, the influence of unknowns on each other is recorded whenever arising, namely in the function `query`: We introduce a mapping `infl`: $\mathcal{U} \to \mathcal{P}(\mathcal{U})$. For each value $\mathtt{d}_\mathtt{z}$ queried during the evaluation of a right-hand side $\mathcal{T}$ y, unknown y is added to the set of unknowns influenced by z (Listing `query`, l. 7). Yet recomputations are expensive; we thus strive to keep the effect of (future) destabilization as limited as possible. For this reason, function `destab` also manages the mapping `infl` and resets all assignments `infl` z to the empty set whenever an unknown z is removed from `stable`.

An elementary characteristic of the destabilization strategy outlined above is its passivity: Unknowns are removed from `stable`, but not reevaluated directly. Unlike solvers like the RLD [24], the TD pursues a demand driven approach where values are only evaluated when queried. In this context, it is especially important to note that x is only removed from `stable` in case of a circular dependency, i.e., x (transitively) influences x. If none of the input values of right-hand side $\mathcal{T}$ x change, reevaluation is not necessary. Subsequently, the iteration terminates with x $\in$ `stable`.

In their work, Stade et al. [44] show the equivalence of the plain TD and its extension with the non-local destabilization mechanism. Partial correctness of both algorithms follows as corollary [44]. From here on, the top-down solver extended by memoization is also referred to as vanilla TD.

2.6 An Introduction to Widening and Narrowing

Recall Example 2 once again. Memoization is able to prune redundant subtrees below nodes `iterate` x. However, the actual performance problem persists: The sequence of values $\sigma\,\mathrm{x}$ is only slowly increasing, so it takes many iterations for the algorithm to converge. Although memoization can skip superfluous calculations, it cannot accelerate essential iterations that inevitably arise with a recursive system of equations. At worst, the solver will not terminate: Once more, have a look at the minimal example $x = x + 1$. The sequence $(x_i)_{i \in \mathbb{N}}$, where x_i corresponds to the unknowns's value $\sigma\, x$ after the i-th iteration, is infinite. The resulting loss of potential was already recognized in the early days of abstract interpretation. P. Cousot and R. Cousot [14, 13] first proposed the widening technique as viable remedy.

In this work, we aim to verify the vanilla TD extended by widening and narrowing. The remainder of this chapter is dedicated to an introduction to the concept and a formal definition of both operators. Detailed examples are provided to demonstrate the principle of function and motivate the techniques used. For an exact implementation of the solver with widening and narrowing, please consult Chapter 3.

We start by introducing a widening operator $\nabla\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ that takes two values a and b in $\mathbb{D}$. This binary operator is largely free to choose, but must comply with the following lower bounds [14, 12, 42]:

$$a, b \leq a \nabla b \tag{6}$$

Above definition is sound in case the pairwise supremum is defined. The constraint is thus synonymous with inequality $a \sqcup b \leq a \nabla b$. Given an old value a (in our implementation $\sigma\,\mathrm{x}$) and a new value b ($\mathtt{d}_\mathrm{x}$). Then the widening operator should (over-)approximate a combination of the two values a and b. Instead of proceeding with value $\mathtt{d}_\mathrm{x}$ in the *Continue Case* (function `iterate`), the solver now uses the widened value. Observing both values at the same time allows a prediction of the evolution of the overall value sequence. A well-chosen widening operator considerably speeds up calculations without sacrificing more accuracy than necessary.

Example 4: In the context of an interval analysis, let $\mathbb{D}' = \mathbb{I} \cup \bot$. We fix $\bot = [\,]$, where $[\,]$ denotes the empty interval. Non-empty intervals of integers are specified by:

$$\mathbb{I} = \Big\{ [l, u] \mid l \in \{-\infty\} \cup \mathbb{Z},\ u \in \mathbb{Z} \cup \{+\infty\},\ l \leq u \Big\}$$

A partial ordering of intervals $\leq$ is intuitively defined as inclusion, more specifically

$$\forall d \in \mathbb{I}.\ [\,] \leq d$$
$$[l_1, u_1] \leq [l_2, u_2] \iff l_2 \leq l_1 \wedge u_1 \leq u_2.$$

Assume $a, b \in \mathbb{D}'$. The union of two intervals $a \sqcup b$ is then given as $[\,] \sqcup d = d \sqcup [\,] = d$ for arbitrary $d \in \mathbb{D}'$. Else, both operands are non-empty intervals and

$$[l_1, u_1] \sqcup [l_2, u_2] = [l_1 \sqcap l_2, u_1 \sqcup u_2],$$

where $x \sqcup y := \max(x, y)$ and $x \sqcap y := \min(x, y)$ for $x, y \in \mathbb{Z} \cup \{-\infty, \infty\}$. Ultimately, widening is defined with respect to the intervals' endpoints. In case of empty interval operands, the widening operator behaves exactly as the union operator above. Otherwise $[l_1, u_1] \nabla [l_2, u_2] = [l, u]$ with

$$l = \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases} \qquad \text{and} \qquad u = \begin{cases} u_1 & \text{if } u_2 \leq u_1 \\ \infty & \text{otherwise} \end{cases}$$

Note that a widening operator is not necessarily commutative. Due to the underlying intention to accelerate ascending chains, a special focus is often placed on situations where the new value b is greater than the old value a.

A solver that applies widening to its value iteration can be shown to return partially correct post-solutions in case of termination[14, 3, 12]. However, these results are potentially very imprecise. Therefore, widening is often followed by or, more generally, complemented with a narrowing phase that recovers some precision. This in turn narrows the result decrementally. A narrowing operator $\Delta\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ should always be selected in such a way that it continues to guarantee (partial) correctness after each application. As narrowing only ever improves an existing result, termination can be enforced reliably by limiting the amount of descending iterations. The aforementioned properties are commonly ensured by [14, 12, 42]:

$$a \sqcap b \leq a \,\Delta\, b \leq a \tag{7}$$

Initially, widening and narrowing were applied in two separate fixpoint iterations, each returning their own post-solution [14, 13]. Yet this approach has two limitations. For one, information and hence precision may be lost due to the strict independence of both phases. But even worse, pure narrowing can only be used at the expense of a severe restriction: Right-hand sides of the equation system must be monotonic [2, 42]. Apinis et al. [3] address these problems by establishing the warrowing operator $\boxslash$: $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$:

$$a \boxslash b := \begin{cases} a \,\Delta\, b & \text{if } b \leq a \\ a \,\nabla\, b & \text{otherwise} \end{cases} \tag{8}$$

As with widening and narrowing, the warrowing operator can overapproximate the new value calculated in `iterate` and thus speed up computations. However, warrowing no longer strictly distinguishes between (global) widening and narrowing phases. Instead, the appropriate operator is selected depending on the current development of the unknown's value. As a result, a single pass suffices to compute a post-solution – provided the solver terminates.

Example 4 (continued): In accordance with the definitions above, we supply an intersection operator on intervals. It is defined by $[\,] \sqcap d = d \sqcap [\,] = [\,],\ d \in \mathbb{D}'$ and

$$[l_1, u_1] \sqcap [l_2, u_2] = \begin{cases} [l_1 \sqcup l_2, u_1 \sqcap u_2] & \text{if } l_1 \sqcup l_2 \leq u_1 \sqcap u_2 \\ [\,] & \text{otherwise} \end{cases}$$

in any other case. We also describe a compatible narrowing function that fulfills aforementioned constraints. Again, $[\,] \,\Delta\, d = d \,\Delta [\,] = [\,]$ for all d in $\mathbb{D}'$ and $[l_1, u_1] \,\Delta\, [l_2, u_2] = [l, u]$ where

$$l = \begin{cases} l_2 & \text{if } l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases} \qquad \text{and} \qquad u = \begin{cases} u_2 & \text{if } u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}.$$

Careful readers may already notice that an iteration trivially terminates if $\mathtt{d}_x \leq \sigma\, x = [l_1, u_1]$, but neither $l_1 = -\infty$ nor $u_1 = \infty$. Such a constellation may occur if the preceding iteration did not apply widening. Analogous to the literature [14, 42, 46], we now use the operators previously defined to perform an interval analysis on code variables:

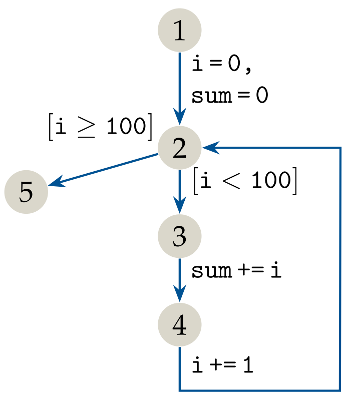
```
i = 0; sum = 0;
while (i < 100) {
  sum += i;
  i++;
}
```

Given the code snippet to the left, we start by establishing a system of constraints. This first approximation is then used to set up an equation system, which in turn describes the solution of our analysis. In this exercise, we are interested in abstract variable assignments $\mathbb{D}$.[3] For each program point, they map variables to a valid interval of possible values.

Consider the program's control flow graph below. Reasonable constraints arise from the semantics of individual statements. For example, the initialization of variables with

[3]Due to the simplicity of the chosen example, it would even be possible to calculate exact results using constant propagation.

constant c is implemented by remapping affected variables to a single value interval $[c, c]$. Particularly noteworthy is the effect of the loop head and its condition: Depending on the chosen branch, affected values are filtered according to the condition's effect. All other changes are expressed by means of interval arithmetic.[4]



$$
\begin{aligned}
p_2 &\geq p_1 \oplus \{\texttt{i} \mapsto [0,0]\} \\
&\qquad \oplus \{\texttt{sum} \mapsto [0,0]\} \\
p_2 &\geq p_4 \oplus \{\texttt{i} \mapsto p_4\,\texttt{i} + [1,1]\} \\
p_3 &\geq p_2 \oplus \{\texttt{i} \mapsto p_2\,\texttt{i} \sqcap [-\infty, 99]\} \\
p_4 &\geq p_3 \oplus \{\texttt{sum} \mapsto p_3\,\texttt{sum} + p_3\,\texttt{i}\} \\
p_5 &\geq p_2 \oplus \{\texttt{i} \mapsto p_2\,\texttt{i} \sqcap [100, \infty]\}
\end{aligned}
$$

Where there are several constraints for a program point, these can be combined into one right-hand side by using unification. In the ensuing analysis, particular attention is paid to the loop head. This is due to the fact that if we are interested in the abstract values of the variables i and sum at the end of the program, recursive queries occur at program point p_2. Without going into the reasons, we additionally apply warrowing to only a few selected iteration steps.[5] In Figure 1, they are marked with the operator effectively used (∇ for widening vs. Δ for narrowing).

The resulting computation trace shows considerable speed up. As intended, the solver terminates after just a few iteration steps and can therefore significantly improve its runtime. However, it is important to emphasize that by applying the widening operator, some information may be permanently lost. That is the case for the variable sum: Whereas i's value is accurately determined as 100, all that is known about the sum is that it is non-negative. Depending on the exact question, this result may suffice – or necessitate a refinement of operators or the equation system.

[4]For completeness: Interval addition is defined by $[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$ for non-empty intervals and $[\,]$ if there is at least one empty interval operand.
[5]For the underlying logic, refer to the implementation in Chapter 3.

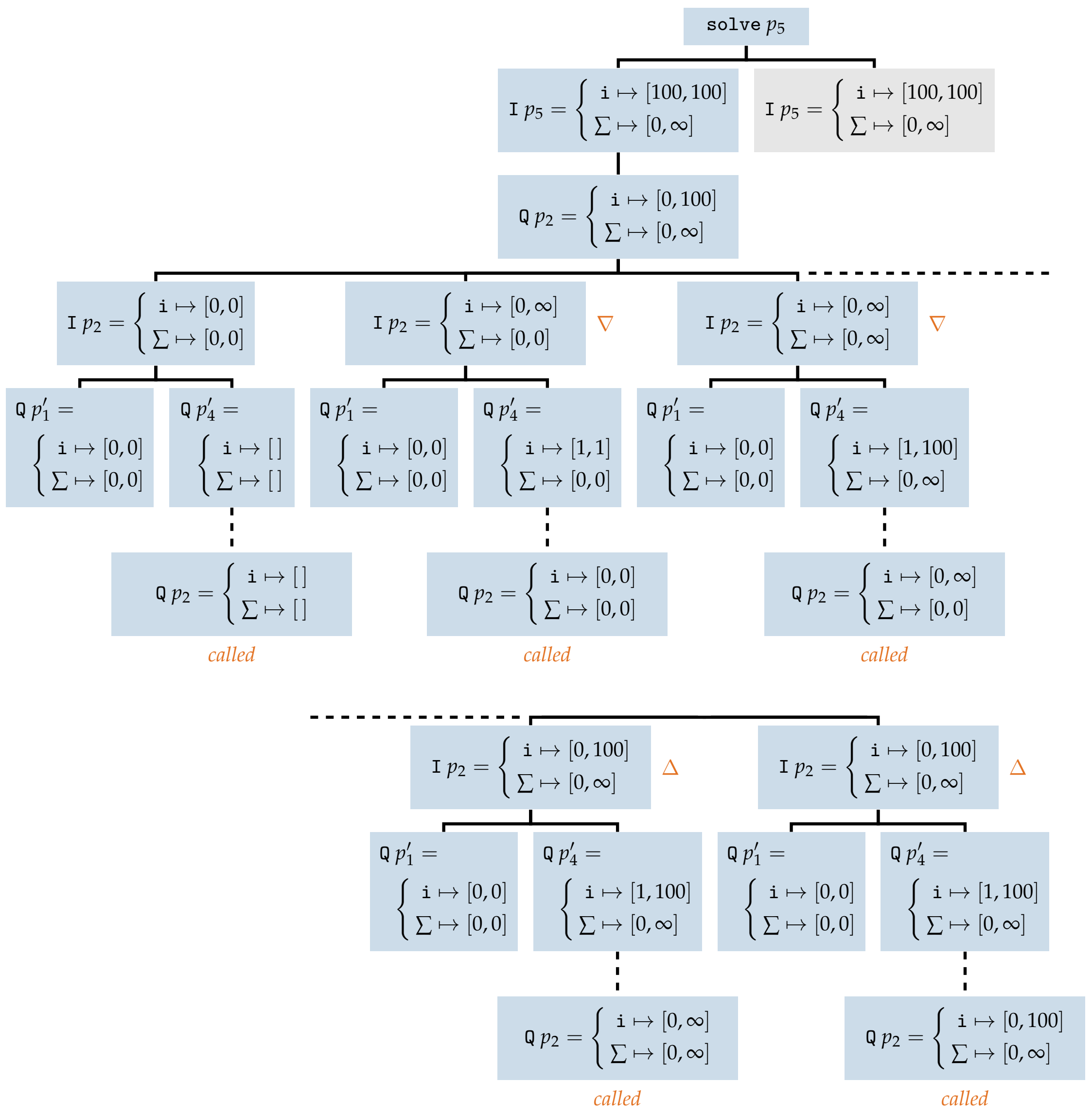

solve p_5
I p_5 = { i ↦ [100, 100], Σ ↦ [0, ∞] }
I p_5 = { i ↦ [100, 100], Σ ↦ [0, ∞] }
Q p_2 = { i ↦ [0, 100], Σ ↦ [0, ∞] }
I p_2 = { i ↦ [0, 0], Σ ↦ [0, 0] }
I p_2 = { i ↦ [0, ∞], Σ ↦ [0, 0] } ▽
I p_2 = { i ↦ [0, ∞], Σ ↦ [0, ∞] } ▽
Q p_1' = { i ↦ [0, 0], Σ ↦ [0, 0] }
Q p_4' = { i ↦ [], Σ ↦ [] }
Q p_1' = { i ↦ [0, 0], Σ ↦ [0, 0] }
Q p_4' = { i ↦ [1, 1], Σ ↦ [0, 0] }
Q p_1' = { i ↦ [0, 0], Σ ↦ [0, 0] }
Q p_4' = { i ↦ [1, 100], Σ ↦ [0, ∞] }
Q p_2 = { i ↦ [], Σ ↦ [] }
called
Q p_2 = { i ↦ [0, 0], Σ ↦ [0, 0] }
called
Q p_2 = { i ↦ [0, ∞], Σ ↦ [0, 0] }
called
I p_2 = { i ↦ [0, 100], Σ ↦ [0, ∞] } △
I p_2 = { i ↦ [0, 100], Σ ↦ [0, ∞] } △
Q p_1' = { i ↦ [0, 0], Σ ↦ [0, 0] }
Q p_4' = { i ↦ [1, 100], Σ ↦ [0, ∞] }
Q p_1' = { i ↦ [0, 0], Σ ↦ [0, 0] }
Q p_4' = { i ↦ [1, 100], Σ ↦ [0, ∞] }
Q p_2 = { i ↦ [0, ∞], Σ ↦ [0, ∞] }
called
Q p_2 = { i ↦ [0, 100], Σ ↦ [0, ∞] }
called

Figure 1: The computation trace of a call to `solve` p_5 in Example 4. In the first iteration, the solver detects a circular dependency on the loop head p_2. Consequently, warrowing is applied from the second iteration onwards. To clearly separate the effect of the warrowing operator applications and the assignments along the edges of the control flow graph, we introduce additional points p_1' and p_4'. They mirror the value of program point p_1 where the effect of `i = 0`, `sum = 0` has already been applied or p_4 combined with the increment `i += 1`, respectively.

3 Verification of the Warrowing TD

The warrowing TD improves the vanilla TD [44] by including widening and narrowing in the iteration phase. Following Tilscher et al. [46], we implement the solver such that warrowing is applied as rarely as possible. This ensures that results remain precise, while the performance benefits from the efficiency supplied by widening. The additional mechanism is realized as follows:

- We introduce a new set `point`. It is used to dynamically track suitable warrowing points, thereby achieving that warrowing is only applied to one unknown per loop.
- Once the iteration on an unknown x is completed, x is removed from `point`. As shown by Tilscher et al. [46], this additional step can prevent loss of information when analyzing a system of equations with nested loops.

Starting from the implementation of the vanilla TD (see the Listings in Section 2.3), warrowing is included by two minor but effective changes. The function `eval` remains unchanged, except for the additional passing of parameter `point`, abbreviated with `p`:

```
eval x t c infl stable p σ = case t of
    Answer d ⇒ (d, infl, stable, p, σ)
  | Query y g ⇒
      let (d_y, infl, stable, p, σ) = query x y c infl stable p σ in
      eval x (g d_y) c infl stable p σ
```

Remember that a cyclic dependency on an unknown x is detected in function `query`, where the condition $x \in c$ causes an end of vertical recursion. Accordingly, the warrowing TD classifies x as a suitable candidate for warrowing. In consequence, the unknown x is added to the set of potential warrowing points `point`:

```
query y x c infl stable p σ =
  let (d_x, infl, stable, p, σ) =
    if x ∈ c then
      (σ x, infl, stable, p ∪ {x}, σ)
    else
      iterate x (c ∪ {x}) infl stable p σ
  in (d_x, infl ⊕ {x ↦ infl x ∪ y}, stable, p, σ)
```

The information gained through the refined self-observation in `query` is then used in function `iterate`. Relevant changes occur at three lines of code:

```
1  iterate x $c$ infl stable p $\sigma$ =
2    if x $\notin$ stable then
3      let ($'d_x$, infl, stable, p', $\sigma$)
4            = eval x (T x) $c$ infl (stable $\cup$ {x}) p $\sigma$ in
5      let $d_x$ = if x $\notin$ p then $'d_x$ else $\sigma$ x $\boxslash$ $'d_x$ in
6      if $\sigma$ x = $d_x$ then
7        ($\sigma$ x, infl, stable, p' − {x}, $\sigma$)
8      else
9        let (infl, stable) = destab x infl stable in
10       iterate x $c$ infl stable p' ($\sigma \oplus$ {x $\mapsto$ $d_x$})
11   else
12     ($\sigma$ x, infl, stable, p − {x}, $\sigma$)
```

For an unknown x that is not in `point` at the beginning of a function call `iterate`, the execution does not differ from that of the vanilla TD. In case x $\notin$ `stable` and x $\in$ `point`, on the other hand, the warrowing TD applies the warrowing operator. At first, right hand side $\mathcal{T}$ x is evaluated. The result value $'d_x$ is then combined with the outdated value stored in the returned assignment σ as follows: `let` $d_x = \sigma\, x \boxslash {}'d_x$. Computations are subsequently resumed using the widened or narrowed value d_x instead of the previous evaluation's exact result $'d_x$.

An unknown x once added to `point` will only remain in the set of warrowing points until iteration on x terminates. Whereas this step could be omitted, it would possibly cause the solver to calculate a less precise post-solution by over-eagerly applying widening. To prevent such cases, we remove x from the set of warrowing points as soon as a fixpoint of the iteration is reached. Find that these fixpoints can also be reached implicitly (c.f. line 12): Assume that x $\in$ `point` but the self-reference is outdated, i.e., $x \notin \text{dep}\, x\, \sigma$, where σ is the assignment returned by the call to `eval`. Then possibly $\sigma\, x \neq d_x$, but the call to `destab` will not remove x from the set of stable unknowns. Iteration will hence terminate in the next iteration step due as unknown x is in `stable`.

Finally, the function `solve` is adapted such that the set `point` is initialized with the empty set. The return values remain the same, as `point` is only used to pass on information internally. Due to the removal of unknowns from `point` after a completed iteration, it will be empty again once the first call to `iterate` terminates:

```
solve x = let (_, _, stable$'$, _, $\sigma$) = iterate x {x} infl$_0$ {} {} $\sigma_0$ in
          (stable$'$, $\sigma$)
```

3.1 Preliminaries

Our main objective in this thesis is to verify soundness of the solver TD extended by warrowing. A solver is sound if it returns correct results in case a call `solve` x terminates. In turn, a result of the warrowing TD is correct if it returns a partial post-solution σ' for all unknowns in the computed set $\texttt{stable}'$ and $\mathrm{x} \in \texttt{stable}'$ holds. Along the lines of Stade, Tilscher, and Seidl [44], we conduct the proof by a mutual induction on the mutually recursive definition of functions `eval`, `iterate` and `query`. The induction scheme is supported by an invariant; it needs to apply to all steps along a computation trace of terminating executions. For an appropriate choice of invariant, soundness follows as corollary.

The invariant we propose for the warrowing TD is almost equivalent to that of Stade et al. [44], except for one significant difference: It is no longer possible to show that the calculated assignment σ' is a partial solution of the equation system. Due to the overapproximating nature of widening, precision may be lost permanently. However, the constraints on the choice of a widening and narrowing operator allow for a slightly weaker conclusion: Fixpoints computed by the warrowing TD still provide a valid (partial) post-solution. The core statement of our invariant thus is:

$$\forall x \in \texttt{stable} - c \,.\, \mathcal{T}\,\sigma\,x \leq \sigma\,x$$

Remember that the called set c collects all unknowns currently under evaluation. By design of the solver, their values are enforced to remain unchanged even during re-evaluation. Set $\texttt{stable} - c$ therefore represents the subset of *truly stable* unknowns, i.e., those that truly reached a fixpoint.

Additional statements support the core invariant in the ensuing inductive proof. For this, we collect observations on the interaction of parameters at various points in a computation trace. The current assignment of parameters is also referred to as *state* of the solver; it is considered *valid* if such a state can result from an initial call to `solve`. Sufficient conditions for a valid state are:

- For all truly stable unknowns $\mathrm{y} \in \texttt{stable} - c$, the influence relation `infl` is complete. Let $\mathcal{D} := \texttt{dep}\,\sigma\,\mathrm{y}$ be the set of unknowns a truly stable unknown y depends on. Then y is influenced by every x in $\mathcal{D}$. This correlation must be represented in `infl`: $\forall \mathrm{y} \in \texttt{stable} - c.\ \forall \mathrm{x} \in \texttt{dep}\,\sigma\,\mathrm{y}.\ \mathrm{y} \in \texttt{infl}\ \mathrm{x}$.
- $\{\mathrm{x} \in \mathcal{U} \mid \texttt{infl}\,\mathrm{x} \neq \{\}\} \subseteq \texttt{stable}$, i.e., the `infl` mapping is minimal in that it only contains influences of `stable` unknowns on other $\mathrm{u} \in \mathcal{U}$. For any $\mathrm{x} \notin \texttt{stable}$, either x was not queried before and thus no influences were collected yet. Or, it was destabilized and hence removed from `stable`. In that case, we expect `destab` to exhaustively remove all influenced unknowns from `infl x`.

- An unknown $\mathrm{x} \in \mathcal{U}$ on the call stack is optimistically added to the `stable` set before evaluation of x begins. Accordingly, $\mathrm{x} \in c$ implies $\mathrm{x} \in$ `stable` which corresponds to $c \subseteq$ `stable`.[1]

All of the aforementioned constraints are aggregated in a predicate valid. We adopt the formulation of Stade et al. [44], but adjust condition *(ii)* by relaxing it to a *post-solution criterion* for all truly stable unknowns $\mathrm{u} \in \mathtt{stable} - c$:

Definition 1 (Warrowing TD Invariant [44])**.** *With a solver state described by* $c, \mathtt{stable} \subseteq \mathcal{U}, \sigma \colon \mathcal{U} \to \mathbb{D}$ *and* $\mathtt{infl} \colon \mathcal{U} \to \mathcal{P}(\mathcal{U})$ *the predicate* valid $c\ \sigma$ `infl stable` *is satisfied if:*

(i) $c \subseteq \mathtt{stable}$
(ii) $\forall \mathrm{u} \in \mathtt{stable} - c \,.\, \mathcal{T}\,\mathrm{u}\,\sigma \leq \sigma\,\mathrm{u}$
(iii) $\{\mathrm{x} \in \mathcal{U} \mid \mathtt{infl}\,\mathrm{x} \neq \{\}\} \subseteq \mathtt{stable}$
(iv) $\forall \mathrm{y} \in \mathtt{stable} - c \,.\, \forall \mathrm{x} \in \mathsf{dep}\,\sigma\,\mathrm{y}.\ \mathrm{y} \in \mathtt{infl}\,\mathrm{x}$

Interestingly enough, the set of potentially warrowing points `point` is never mentioned in this definition. We conclude that the solver TD with memoization shows correct behavior regardless of how set `point` is handled. Conceptually, two extremes can be differentiated: One is that the solver never adds any unknowns to set `point` – in that case, the warrowing TD is equivalent to the vanilla TD. As shown by Stade et al. [44], such a solver is partially correct. Apart from this scenario, we also consider the case that every unknown encountered is added to `point`. In consequence, warrowing is applied to all unknowns iterated upon. Again, the warrowing TD produces sound results [3] – but trades precision for improved runtime characteristics. All in all, if we demand post-solutions instead of solutions, the solver's handling of the set `point` only ever influences its termination characteristics, but not its correctness.

Further propositions regarding the memoization mechanism are specified in a predicate update. Let $\mathtt{fun} \in \{\mathtt{query}\,\mathrm{x}\,\mathrm{y}\,,\ \mathtt{iterate}\,\mathrm{x}\,,\ \mathtt{eval}\,\mathrm{x}\}$. For a function call `fun stable infl` $\mathtt{point}\,\sigma = (d, \mathtt{infl}', \mathtt{stable}', \mathtt{point}', \sigma')$, the predicate update relates the input solver state in `stable`, `infl` to the output solver state in $\mathtt{stable}'$, $\mathtt{infl}'$. We adopt the definition by Stade et al. without modifications:

Definition 2 (Update Relation [44])**.** *For* $\mathrm{x} \in \mathcal{U}, \mathtt{stable}, \mathtt{stable}' \subseteq \mathcal{U}$ *and* $\mathtt{infl}, \mathtt{infl}' \colon \mathcal{U} \to \mathcal{P}(\mathcal{U})$ *the predicate* update x infl stable $\mathtt{infl}'$ $\mathtt{stable}'$ *is satisfied if*

(i) $\mathtt{stable} \subseteq \mathtt{stable}'$*, i.e., the solver only increases the set of (pseudo-)stable unknowns,*

[1]Inserting an unknown into the `stable` set too early would completely inhibit its fixpoint iteration. Consequently, there is a transitional phase: During iteration on some unknown $\mathrm{x} \in \mathcal{U}$, indeed $\mathrm{x} \in c$ but $\mathrm{x} \notin$ `stable`. This circumstance is incorporated in Theorem 1 by introducing a set $c' := c - \{\mathrm{x}\}$.

(ii) $\forall \mathtt{u}.\ (\mathtt{infl}'\ \mathtt{u} - \mathtt{infl}\ \mathtt{u}) \cap (\mathtt{stable} - \{\mathtt{x}\}) = \{\}$, *i.e., the solver never detects influences on stable unknowns* $\mathtt{v} \in \mathtt{stable}$ *that were previously unidentified. This rule need not apply to unknown* $\mathtt{x}$*, for which additional influencing unknowns may be discovered.*

(iii) $\forall \mathtt{u} \in \mathtt{stable}.\ \mathtt{infl}\,\mathtt{u} \subseteq \mathtt{infl}'\,\mathtt{u}$, *i.e., the solver only ever gains knowledge about the influence of stable unknowns but never loses such information.*

Ultimately, we will proof that the invariant valid is preserved by all three of the mutually recursive functions defining the warrowing TD. To this end, we show that the invariant remains intact throughout all intermediate computations. Such a statement is not trivially given for the destab mechanism; we therefore refer to the proposition of an auxiliary lemma by Stade et al., which is adapted to reflect the post-solution criterion:

Lemma 1 (Destabilization and Update of σ Preserve Partial Post-Solution [44])**.** *For* $\mathtt{x} \in \mathcal{U}$, $\mathtt{infl}, \mathtt{infl}' \colon \mathcal{U} \to \mathcal{P}(\mathcal{U})$ *and* $\mathtt{stable}, \mathtt{stable}' \subseteq \mathcal{U}$*, let* $\mathtt{infl}'$ *and* $\mathtt{stable}'$ *such that* $\mathtt{destab\ x\ infl\ stable} = (\mathtt{infl}', \mathtt{stable}')$ *holds. Let* $c' := c - \{x\}$ *and* $\sigma' := \sigma \oplus \{\mathtt{x} \mapsto d_{\mathtt{x}}\}$*. Assume*

(i) $\forall \mathtt{x} \in \mathtt{stable} - c\,.\ \mathcal{T}\,\mathtt{x}\,\sigma \leq \sigma\,\mathtt{x}$,

(ii) $\mathcal{T}\,\mathtt{x}\,\sigma \leq d_{\mathtt{x}}$ *and*

(iii) $\forall \mathtt{u} \in \mathtt{stable} - c'\,.\ \forall \mathtt{v} \in \mathsf{dep}\ \sigma\,u.\ \mathtt{u} \in \mathtt{infl}\,\mathtt{v}$.

Then $\forall \mathtt{x} \in \mathtt{stable}' - c'\,.\ \mathcal{T}\,\mathtt{x}\,\sigma' \leq \sigma'\,\mathtt{x}$ *holds.*

3.2 Properties of the Warrowing Operator ⧄

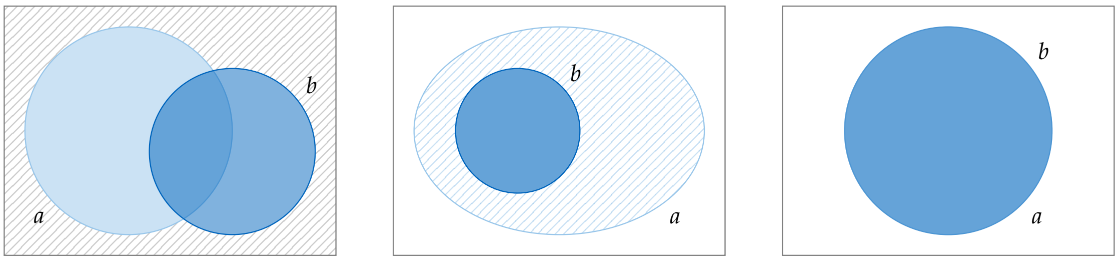
So far, the warrowing operator ⧄ was described by its defining suboperators ∇ and Δ. Properties of the latter are enforced by constraints, given in Equation 6 and 7. Recall the definition of the warrowing operator in (8):

$$a \boxslash b := \begin{cases} a \,\Delta\, b & \text{if } b \leq a \\ a \,\nabla\, b & \text{otherwise} \end{cases}$$

The case distinction in warrowing provides a further assumption per suboperator, namely $b \leq a$ for narrowing and $\neg\, b \leq a$ for widening[2]. This allows us to derive a unique characteristic of the warrowing operator, reflected in the statements below:

Lemma 2 (Warrowing Lower Bound)**.** *For arbitrary* $a, b \in \mathbb{D}$*, the second operand* b *is a lower bound of the operation* $a \boxslash b$*; that is,* $b \leq a \boxslash b$ *holds true.*

[2]Remember that $\mathbb{D}$ is required to be a join-semilattice. However, this by no means implies that elements in $\mathbb{D}$ are ordered linearly. Therefore $\neg\, b \leq a \not\equiv b > a$.



(a) Case $\neg\, b \leq a$: *Widening* **(b)** Case $b < a$: *Narrowing* **(c)** Case $b = a$: *Fixpoint*

Figure 2: For $a, b \in \mathbb{D}$, consider the three exhaustive cases (a) $\neg\, b \leq a$, (b) $b < a$ and (c) $b = a$. According to the definition of the operators ∇, Δ and $\boxslash$, warrowing is bounded by ■ $\leq a \boxslash b \leq$ ▨ in the diagrams above (disregarding the color). The warrowing operator behaves according to the intuition for (a) widening, (b) "true" narrowing and (c) the fixpoint case $b = a$, which is handled by the narrowing operator.

Proof. The lemma simply follows from the properties of widening and narrowing: In case $b \leq a$ the warrowing operator chooses narrowing. The fact $b \leq a$ also implies $a \sqcap b = b$. When paired with the narrowing operator's lower bound (Equation 7), this leads to the desired conclusion $b = a \sqcap b \leq a \,\Delta\, b$. In case $\neg\, b \leq a$, on the other hand, widening is applied. For widening, condition $b \leq a \,\Delta\, b$ is adhered to per definition 6. In consequence, b is a valid lower bound of the warrowing operation $a \boxslash b$. □

Lemma 3 (Warrowing Fixpoint Inequality)**.** *Let $f : \mathbb{D} \to \mathbb{D}, x \mapsto x \boxslash b$ for arbitrary but fixed $b \in \mathbb{D}$. Assume $a \boxslash b = a$ for a value $a \in \mathbb{D}$. In other words, a is a fixpoint of the warrowing operator application defined by f. Then $b \leq a$.*

Proof. The thesis is readily shown by contradiction: Assume $\neg\, b \leq a$. According to the definition of the warrowing operator, this implies the application of widening: $a = a \,\nabla\, b$. Combined with constraint $b \leq a \,\nabla\, b$ (Equation 6), $b \leq a$ follows. The inequality thus derived is in direct contradiction to the assumption, hence $b \leq a$. □

Lemma 3 reveals an interesting property of the warrowing TD: A chain of warrowing iterations always terminates with an application of the narrowing operator. This behavior corresponds to the initial concept of widening and narrowing proposed by P. Cousot and R. Cousot [14], with the final stage dedicated to an attempt to refine the results obtained:

Corollary 1 (Fixpoint Implies Narrowing)**.** *Assume $a = a \boxslash b$ for values $a, b \in \mathbb{D}$. The fixpoint is only reached if the warrowing operator dispatches narrowing, i.e., $a = a \,\Delta\, b$.*

A detailed analysis comparing the relation of input values a and b (see Figure 2) yields a final proposition regarding fixpoints, that is, solutions of the currently processed equation $\mathrm{x} \mapsto \mathcal{T}\,\mathrm{x}$:

Lemma 4 (Warrowing Preserves Fixpoints)**.** *Assume $a = b$ for values $a, b \in \mathbb{D}$. Then $a \boxslash b = a$, i.e., the warrowing operator preserves fixpoints $\mathcal{T}\,\mathrm{x}\,\sigma = \sigma\,\mathrm{x}$.*

Proof. Due to the assumption, we know warrowing dispatches narrowing, that is, $a \boxslash b = a \,\Delta\, b$ holds and hence $a \,\Delta\, b \leq a$ (Equation 7) must apply. Simultaneously, both values a and b are a lower bound of the warrowing operator application: $a = b \leq a \boxslash b$ (lemma 2). Both facts combined, $a \leq a \boxslash b \leq a$ holds and $a \boxslash b = a$ follows. □

3.3 Soundness Proof

Whereas Stade et al. [44] prove partial correctness of the vanilla TD by showing its equivalence to the plain TD, such an approach is not feasible for the warrowing TD. The results of the warrowing TD merely fulfill the post-solution criterion. In addition, termination properties may differ, so that the results of a run of the warrowing TD are defined, while those of the vanilla TD are not. We therefore choose to conduct the proof in a standalone fashion.

The proof goals consist of details concerning the implementation of the individual functions `query`, `iterate` and `eval`, respectively. Furthermore, all changes in the influence mapping and stable set must comply with the update relation. Parallel to the standalone verification of the plain TD [44], we require that the mapping σ is invariant for stable unknowns $\mathrm{u} \in \mathcal{U}$. By doing so, the fact is then available to us as an induction hypothesis, hence enabling us to make claims about the preservation of the invariant.

For the remainder of this section, assume the common variables to be of appropriate types, i.e., let unknowns $\mathrm{x}, \mathrm{y} \in \mathcal{U}$, sets $c^*, \mathtt{stable}^*_{i?}, \mathtt{point}^*_{i?} \in \mathcal{P}(\mathcal{U})$, mappings $\mathtt{infl}^*_{i?} \colon \mathcal{U} \to \mathcal{P}(\mathcal{U})$, assignments $\sigma^*_{i?} \colon \mathcal{U} \to \mathbb{D}$ and values ${}^*d_{\mathrm{x}}, d_{\mathrm{y}} \in \mathbb{D}$. We then state:

Theorem 1 (Mutual Partial Correctness)**.** *The theorem shows:*

- *Assume* $\mathtt{query}\,\mathrm{y}\,\mathrm{x}\; c\; \mathtt{infl}\; \mathtt{stable}\; \mathtt{point}\; \sigma = (d_{\mathrm{x}}, \mathtt{infl}', \mathtt{stable}', \mathtt{point}', \sigma')$ *is defined and* $\mathsf{valid}\; c\; \sigma\; \mathtt{infl}\; \mathtt{stable}$ *holds. Then*
  (i) $\forall \mathrm{u} \in \mathtt{stable}\,.\, \sigma\,\mathrm{u} = \sigma'\,\mathrm{u}$ *and*
  (ii) $\mathsf{valid}\; c\; \sigma'\; \mathtt{infl}'\; \mathtt{stable}'$ *holds.*
  Furthermore, in this case (iii) $\mathsf{update}\; \mathrm{y}\; \mathtt{infl}\; \mathtt{stable}\; \mathtt{infl}'\; \mathtt{stable}'$ *holds true and (iv)* $\mathrm{y} \in \mathtt{infl}'\,\mathrm{x}$.

- *Assume* $\texttt{iterate}\,\mathrm{x}\;c\;\texttt{infl}\;\texttt{stable}\;\texttt{point}\;\sigma = (d_{\mathrm{x}},\texttt{infl}',\texttt{stable}',\texttt{point}',\sigma')$ *is defined,* $\mathrm{x} \in c$ *and* valid $c'\;\sigma\;\texttt{infl}\;\texttt{stable}$ *holds, where* $c' := c - \{\mathrm{x}\}$*. Then*
  (i) $\forall \mathrm{u} \in \texttt{stable}\,.\,\sigma\,\mathrm{u} = \sigma'\,\mathrm{u}$ *and*
  (ii) valid $c'\;\sigma'\;\texttt{infl}'\;\texttt{stable}'$ *holds.*
  Furthermore, in this case (iii) update $\mathrm{x}\;\texttt{infl}\;\texttt{stable}\;\texttt{infl}'\;\texttt{stable}'$ *holds true and (iv)* $\mathrm{x} \in \texttt{stable}'$.
- *Assume* $\texttt{eval}\,\mathrm{x}\,\mathrm{t}\;c\;\texttt{infl}\;\texttt{stable}\;\texttt{point}\;\sigma = (d_{\mathrm{x}},\texttt{infl}',\texttt{stable}',\texttt{point}',\sigma')$ *is defined,* $\mathrm{x} \in \texttt{stable}$ *and* valid $c\;\sigma\;\texttt{infl}\;\texttt{stable}$ *holds. Then*
  (i) $\forall \mathrm{u} \in \texttt{stable}\,.\,\sigma\,\mathrm{u} = \sigma'\,\mathrm{u}$ *and*
  (ii) valid $c\;\sigma'\;\texttt{infl}'\;\texttt{stable}'$ *holds.*
  Furthermore, in this case (iii) update $\mathrm{x}\;\texttt{infl}\;\texttt{stable}\;\texttt{infl}'\;\texttt{stable}'$ *holds, (iv)* $\mathrm{t}\,\sigma' = d_{\mathrm{x}}$*, and (v)* $\forall \mathrm{u} \in \mathrm{dep}\;\sigma'\;\mathrm{t}.\;\mathrm{x} \in \texttt{infl}'\,\mathrm{u}$.

Proof. The statements are shown by induction, following the induction scheme for mutually recursive functions generated by Isabelle. An induction step corresponds to a function call in the computation trace; accordingly, there are three *cases*. Individual instances of calls can be abstracted based on their execution path; we call them *subcases*. For the vanilla TD, Stade et al. [44] structure their inductive proof as follows:

Case 1 (Query). *Assume* $\texttt{query}\,\mathrm{y}\,\mathrm{x}\;c\;\texttt{infl}\;\texttt{stable}\;\sigma = (d_{\mathrm{x}},\texttt{infl}',\texttt{stable}',\sigma')$ *is defined. Depending on* $\mathrm{x} \in c$*, we distinguish:*
Subcase 1.1 (Lookup). *Assume* $\mathrm{x} \in c$*. Then return values are instantiated by* $(d_{\mathrm{x}},\texttt{infl}',\texttt{stable}',\sigma') = (\sigma\,\mathrm{x},\texttt{infl}',\texttt{stable},\sigma)$ *where* $\texttt{infl}' = \texttt{infl} \oplus \{\mathrm{x} \mapsto \texttt{infl}\;\mathrm{x} \cup \mathrm{y}\}$.
Subcase 1.2 (Iterate). *Assume* $\mathrm{x} \notin c$*. Then* $\texttt{iterate}\,\mathrm{x}\;c'\;\texttt{infl}\;\texttt{stable}\;\sigma = (d_{\mathrm{x}},\texttt{infl}_1,\texttt{stable}',\sigma')$ *is defined where* $c' := c - \{\mathrm{x}\}$ *and* $\texttt{infl}' = \texttt{infl}_1 \oplus \{\mathrm{x} \mapsto \texttt{infl}_1\;\mathrm{x} \cup \mathrm{y}\}$.

Case 2 (Iterate). *Assume* $\texttt{iterate}\,\mathrm{x}\;c\;\texttt{infl}\;\texttt{stable}\;\sigma = (d_{\mathrm{x}},\texttt{infl}',\texttt{stable}',\sigma')$ *is defined and* $x \in c$*. We distinguish 3 subcases:*
Subcase 2.1 (Stable). *Assume* $x \in \texttt{stable}$*. Then return values are instantiated by* $(d_{\mathrm{x}},\texttt{infl}',\texttt{stable}',\sigma') = (\sigma\,\mathrm{x},\texttt{infl},\texttt{stable},\sigma)$.
Subcase 2.2 (Fixpoint). *Assume* $\mathrm{x} \notin \texttt{stable}$*. Let* σ', d_{x} *such that* $\texttt{eval}\,\mathrm{x}\,(\mathcal{T}\,\mathrm{x})\;c\;\texttt{infl}\;(\texttt{stable} \cup \{\mathrm{x}\})\;\sigma = (d_{\mathrm{x}},\texttt{infl}',\texttt{stable}',\sigma')$ *is defined and* $\sigma'\,\mathrm{x} = d_{\mathrm{x}}$.
Subcase 2.3 (Continue). *Assume* $\mathrm{x} \notin \texttt{stable}$*. Let* $\sigma_1, 'd_{\mathrm{x}}$ *such that* $\texttt{eval}\,\mathrm{x}\,(\mathcal{T}\,\mathrm{x})\;c\;\texttt{infl}\;(\texttt{stable} \cup \{\mathrm{x}\})\;\sigma = ('d_{\mathrm{x}},\texttt{infl}_1,\texttt{stable}_1,\sigma_1)$ *is defined and* $\sigma_1\,\mathrm{x} \neq 'd_{\mathrm{x}}$*. Then* $\texttt{iterate}\;\mathrm{x}\;c\;\texttt{infl}_2\;\texttt{stable}_2\,(\sigma_1 \oplus \{\mathrm{x} \mapsto 'd_{\mathrm{x}}\}) = (d_{\mathrm{x}},\texttt{infl}',\texttt{stable}',\sigma')$ *is defined where* $(\texttt{infl}_2,\texttt{stable}_2) = \texttt{destab}\,\mathrm{x}\;\texttt{infl}_1\;\texttt{stable}_1$.

Case 3 (Eval). *Assume* $\texttt{eval}\,\mathrm{x}\,\mathrm{t}\;c\;\texttt{infl}\;\texttt{stable}\;\sigma = (d_{\mathrm{x}},\texttt{infl}',\texttt{stable}',\sigma')$ *is defined. Depending on the type of tree* t*, we distinguish:*
Subcase 3.1 (Answer). *Assume* $\mathrm{t} = \texttt{Answer}\,d_{\mathrm{x}}$*. Then return values are instantiated by* $(d_{\mathrm{x}},$

$\texttt{infl}', \texttt{stable}', \sigma') = (d_\text{x}, \texttt{infl}, \texttt{stable}, \sigma)$.

Subcase 3.2 (Query). *Assume* $\texttt{t} = \texttt{Query}\,\text{y}\,g$. *Then both* $\texttt{query}\,\text{x}\,\text{y}\;c\;\texttt{infl}\;\texttt{stable}\;\sigma = (d_\text{y}, \texttt{infl}_1, \texttt{stable}_1, \sigma_1)$ *and* $\texttt{eval}\,\text{x}\,(g\,d_\text{y})\;c\;\texttt{infl}_1\;\texttt{stable}_1\;\sigma_1 = (d_\text{x}, \texttt{infl}', \texttt{stable}', \sigma')$ *are defined.*

In order to lift their induction scheme and case rules to a scheme suitable for the warrowing TD, we need to extend it by statements that reflect the administration of set point. As already discussed above, none of the proof goals comprises facts about the set of potential warrowing points. It comes as no surprise, then, that the proof of Theorem 1 is conducted almost identical to those of Stade et al. [44] for all cases mentioned above. We therefore only focus on novel insights. They occure in case *Iterate*:

Case 2 (Iterate). *By case premise, invariant* valid *holds for the input arguments* c', σ, infl *and* stable, *where* $c' := c - \{\text{x}\}$. *This implies* **(A)** valid $c\;\sigma\;\texttt{infl}\;(\texttt{stable} \cup \{\text{x}\})$. *To satisfy the proof goal, we need to show* valid $c'\;\sigma'\;\texttt{infl}'\;\texttt{stable}'$ *holds. The invariant includes the proof obligation* $\forall x \in \texttt{stable}' - c'\,.\,\mathcal{T}\,\text{x}\,\sigma' \leq \sigma' x$, *i.e., we have to show that after iteration, the returned assignment* σ' *is still a post-solution for all truly stable unknowns* $\texttt{stable}' - c'$.

With the introduction of set point, *two new subcases arise where* $x \notin \texttt{stable}$ *and* $x \in \texttt{point}$. *In both subcases,* **(B)** *the induction hypothesis (IH) for* eval *applies as the necessary premises are fulfilled with (A). It follows that* $\texttt{eval}\,\text{x}\,(\mathcal{T}\,\text{x})\;c\;\texttt{infl}\;(\texttt{stable} \cup \{\text{x}\})\;\texttt{point}\;\sigma = ({}''d_\text{x}, \texttt{infl}_1, \texttt{stable}_1, \texttt{point}_1, \sigma_1)$ *is defined and* **(C)** $\mathcal{T}\,\text{x}\,\sigma_1 = {}''d_\text{x}$ *holds. Likewise,* **(D)** valid $c\;\sigma_1\;\texttt{infl}_1\;\texttt{stable}_1$ *is satisfied after the evaluation of* $\mathcal{T}\,\text{x}$ *and thus* **(E)** $\forall \text{u} \in \texttt{stable}_1 - c\,.\,\mathcal{T}\,\text{u}\,\sigma_1 \leq \sigma_1\,\text{u}$ *holds. Let* **(F)** ${}'d_\text{x} = \sigma_1\,\text{x} \boxtimes {}''d_\text{x}$.

Subcase 2.4 (Warrowing Continue). *Assume* ${}'d_\text{x} \neq \sigma_1\,\text{x}$. *Let* $\texttt{infl}_2, \texttt{stable}_2$ *such that* $(\texttt{infl}_2, \texttt{stable}_2) = \texttt{destab}\,\text{x}\;\texttt{infl}_1\;\texttt{stable}_1$ *and* $\sigma_1' := \sigma_1 \oplus \{\text{x} \mapsto {}'d_\text{x}\}$. *As in the subcase* Continue, *we want to proof that the premises of the IH for* iterate *are fulfilled (see also Figure 3). We thus need to show that the invariant* valid *in (D) is preserved by both the warrowing operator application and the ensuing destabilization. Accordingly, we aim to prove* valid $c'\;\sigma_1'\;\texttt{infl}_2\;\texttt{stable}_2$ *holds before the iteration continues. The warrowing application in (F) only concerns condition (ii) of the predicate* valid. *We therefore show* $\forall \text{u} \in \texttt{stable}_2 - c'\,.\,\mathcal{T}\,\text{u}\,\sigma_1' \leq \sigma_1'\,\text{u}$ *and refer to the formalization in Isabelle [18] for a thourough proof.*

The aforementioned condition $\forall \text{u} \in \texttt{stable}_2 - c'\,.\,\mathcal{T}\,\text{u}\,\sigma_1' \leq \sigma_1'\,\text{u}$ *is fulfilled by Lemma 1, if:*

- $\forall \text{u} \in \texttt{stable}_1 - c\,.\,\mathcal{T}\,\text{u}\,\sigma_1 \leq \sigma_1\,\text{u}$. *This corresponds to fact (E).*
- $\mathcal{T}\,\text{x}\,\sigma \leq {}'d_\text{x}$. *The inequality follows from* $\mathcal{T}\,\text{x}\,\sigma_1 = {}''d_\text{x}$ *(fact (C)) and* ${}''d_\text{x} \leq {}'d_\text{x}$ *(by Lemma 2 and the definition of* ${}'d_\text{x}$ *in (F)).*
- $\forall \text{u} \in \texttt{stable}_1 - c'.\;\forall \text{v} \in \texttt{dep}\;\sigma_1\,\text{u}.\;\text{u} \in \texttt{infl}_1\,\text{v}$. *By (D), a similiar statement* $\forall \text{u} \in \texttt{stable}_1 - c.\;\forall \text{v} \in \texttt{dep}\;\sigma_1\,\text{u}.\;\text{u} \in \texttt{infl}_1\,\text{v}$ *holds. According to the IH for* eval *(fact (B)),* $\texttt{stable} \cup \{\text{x}\} \subseteq \texttt{stable}_1$ *is valid and therefore* $\text{x} \in \texttt{stable}_1$. *For this reason,*

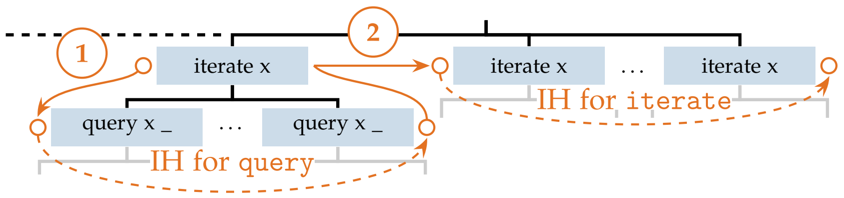


Figure 3: The figure illustrates an induction step of the warrowing TD for the subcase *Warrowing Continue*. Its implementation only differs to that of subcase *Continue* in that warrowing is applied after function `eval` returned a value. To fulfill all premises of the induction hypothesis (IH) for the recursive call `iterate`, we need to show that an application of the warrowing operator preserves the invariant valid. All other proof obligations follow analogously to subcase *Continue*. *Adapted from [44].*

the two statements differ. However, the IH for `eval` *also states that* $\forall \mathrm{v} \in \texttt{dep}\,\sigma_1\,\mathrm{x}.\ \mathrm{x} \in \texttt{infl}_1\,\mathrm{v}$ *is fulfilled. The assumption is thus satisfied.*

This establishes a proof state equivalent to the state before the recursive call of function `iterate` *in subcase* Continue. *The remaining proof follows simultaneously.*

Subcase 2.5 (Warrowing Fixpoint). *Assume* **(G)** $'d_\mathrm{x} = \sigma_1\,\mathrm{x}$. *Let* $\texttt{infl}' = \texttt{infl}_1$, $\texttt{stable}' = \texttt{stable}_1$, **(H)** $\sigma' = \sigma_1$ *and* $d_\mathrm{x} = {'d_\mathrm{x}}$ *(see also Figure 4). The IH for* `eval` *(fact (B)) states that the evaluation step satisfies the* update *relation, therefore* $\texttt{stable} \cup \mathrm{x} \subseteq \texttt{stable}'$ *holds. This implies* $\mathrm{x} \in \texttt{stable}'$. *To fulfill the proof obligation, and even more specifically the post-solution criterion, we thus need to show that* $\mathcal{T}\,\mathrm{x}\,\sigma' \le \sigma'\,\mathrm{x}$. *By the case premise of* `iterate`, $\mathrm{x} \in c$ *holds, hence the proposition is not covered by (E). Instead, combine the facts below:*

- $\mathcal{T}\,\mathrm{x}\,\sigma' = {''d_\mathrm{x}}$ *by (C)*
- *With (F) and (G),* $'d_\mathrm{x} = {'d_\mathrm{x}} \boxslash {''d_\mathrm{x}}$ *holds. Lemma 3 applies, hence* $''d_\mathrm{x} \le {'d_\mathrm{x}}$.
- $'d_\mathrm{x} = \sigma'\,\mathrm{x}$ *due to (G) and (H)*

All remaining proof obligations follow according to the reasoning in subcase Fixpoint.

□

Recall that function `solve` is defined as a wrapper function to `iterate`. Accordingly, a call `solve` x is equivalent to that of `iterate` x $\{\mathrm{x}\}$ $\texttt{infl}_0$ $\{\}$ $\{\}$ $\sigma_0 = (d_\mathrm{x}, \texttt{infl}', \texttt{stable}', \texttt{point}', \sigma')$. The initialization fulfills all required assumptions and Theorem 1 applies. We conclude that valid c' σ' $\texttt{infl}'$ $\texttt{stable}'$ holds and ultimately claim:

Corollary 2 (Partial Correctness of the warrowing TD). *Assume that a call to* `solve` x *terminates, i.e., the equation* $(\texttt{stable}, \sigma) = \texttt{solve}\,\mathrm{x}$ *is defined. Then* σ *is a partial post-solution for* `stable` *and* $\mathrm{x} \in \texttt{stable}$.

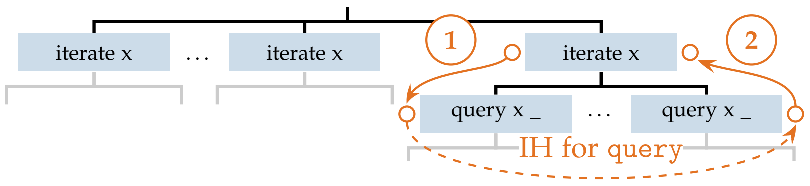
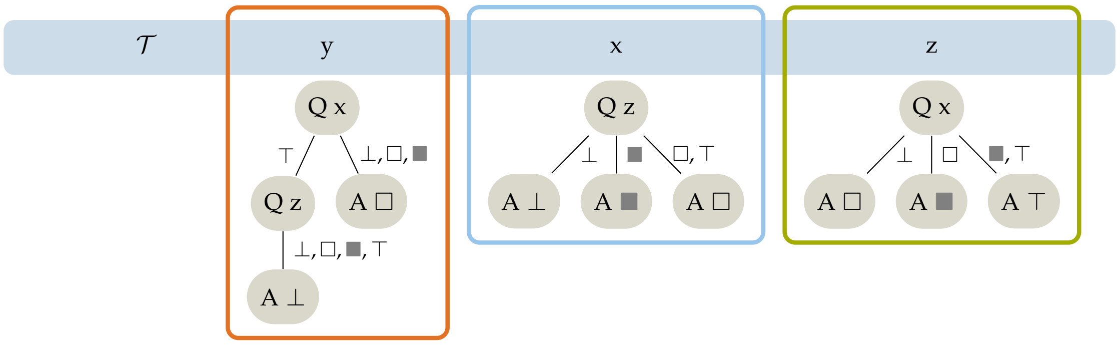


Figure 4: In the subcase *Warrowing Fixpoint*, the induction hypothesis (IH) for query implies that unknown x must be stable after its evaluation terminated. To conclude this subcase, we therefore need to show that the computed value of x, after being subject to warrowing, still fulfills the post-solution criterion for equation $x \mapsto \mathcal{T}\,x$.

3.4 Discussion

In an effort to implement warrowing as a modular feature, early attempts concerned the extension of the plain TD (i.e., without memoization) with widening and warrowing, respectively. However, it turns out that considering such an implementation, partial correctness cannot be proven. Even if only widening is used, the integrity of the solver is violated by the way it administers potential widening points in set `point`. The core problem is outlined in a small counterexample:

Example 5: Let $\mathbb{D} = \{\bot, \square, \blacksquare, \top\}$ where $\bot < \square < \blacksquare < \top$, i.e., the domain is ordered linearly. For unknowns $\mathcal{U} = \{x, y, z\}$, the system of equations $\mathcal{T}$ is defined by the following query-answer trees:



Assume that widening and narrowing operators are given as $a \mathbin{\nabla} b = \top$ and $a \mathbin{\Delta} b = a$ (which effectively disables narrowing) and we employ the plain TD without memoization. A call to `solve` y yields the computation trace illustrated in Figure 5 and returns the assignment

$$\begin{aligned} \square = \sigma\,x &\leq \mathcal{T}\,x\,\sigma = \square \\ \bot = \sigma\,y &\leq \mathcal{T}\,y\,\sigma = \square \\ \top = \sigma\,z &\not\leq \mathcal{T}\,z\,\sigma = \blacksquare \end{aligned}$$

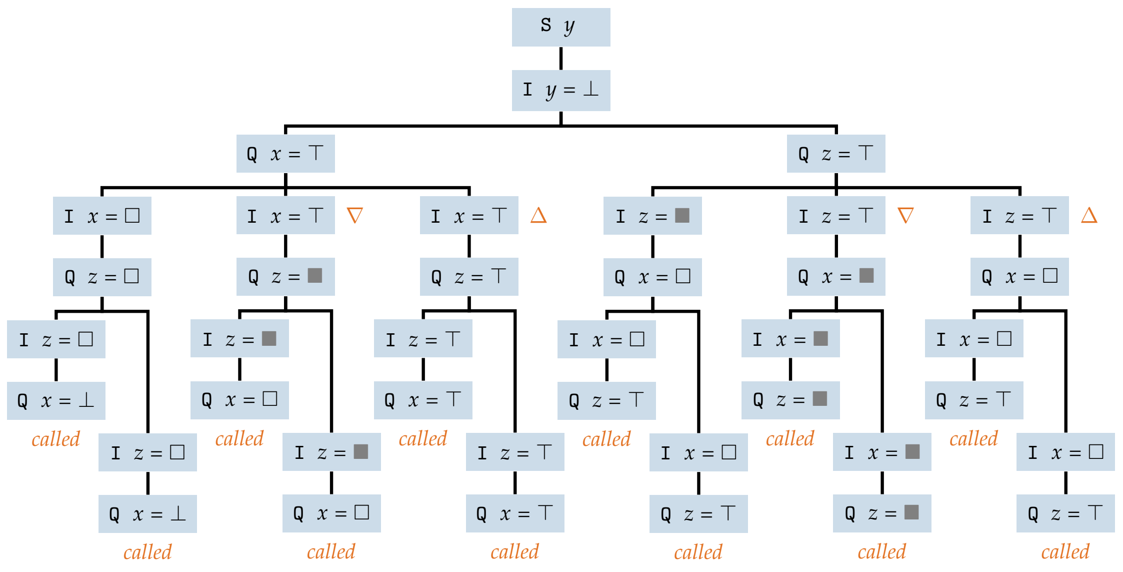
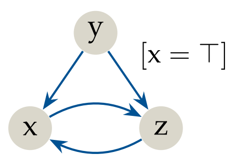


Figure 5: The computation trace of a call to `solve` y in Example 5. Note how the first half of leaf nodes consists of queries of x compared to nodes of the second half, which consists of queries of y. The application of warrowing is redirected correspondingly.

$\sigma_r = \{x \mapsto \square,\ \mathrm{y} \mapsto \bot,\ \mathrm{z} \mapsto \top\}$. However, when validating the returned assignment, it does not fulfill the post-solution criterion for unknown z.



The discordance of values occurs due to a cyclic dependency of unknowns x and z and the fact that `eval` y initially queries both unknowns – whereas an evaluation of the right-hand side of unknown y with the computed solution σ_r does not.

Reconsider the computation trace above. The left half of the tree, that is, all nodes below the topmost `query` of x, assumes unknown x to be the widening point of the aforementioned cycle. But the right half of the tree, that is, all nodes below the topmost `query` of z, recognizes unknown z as widening point. In consequence, the plain TD with warrowing immediately readjusts the value of x *without applying warrowing*. The solver thus loses the integrity of its internally managed assignment σ: While x is ultimately assigned the new value □, the value of y is still calculated based on the old value of x.

In essence, the discrepancy arises when during computation, the solver identifies differing unknowns of one and the same circle as suitable widening points. A similar effect is observable when unknowns $\mathrm{u} \in \mathcal{U}$ are added over-eagerly but mistakenly to set `point` in a first iteration phase and non-recursively re-queried later. Due to them having reached a fixpoint during their first iteration, they are assumed to be stable henceforth, that is, $\sigma\,\mathrm{u} = d_{\mathrm{u}}$ is invariant for all following assignments $\sigma\colon \mathcal{U} \to \mathbb{D}$ and $d_{\mathrm{u}} \in \mathbb{D}$ (c.f. Theorem 1 in [44]) . But on closer inspection, this does not hold: The solver plain TD is not able to reconstruct the information $\mathrm{u} \in$ `point` at a later point in time. The seemingly stable unknown u is therefore treated as a non-widening point $\mathrm{u}' \notin$ `point` and potentially updated to a non-widened value $d'_{\mathrm{u}} \in \mathbb{D}$. This update may, however, cause an inconsistency for all unknowns influenced by u with respect to the resulting assignment $\sigma^\star\colon \mathcal{U} \to \mathbb{D}$. If their calculation was based on the now outdated value d_{x}, the post-solution criterion need not apply. An implementation of the plain TD with widening (or warrowing) can thus not be proven correct.

We propose several remedies to circumvent the loss of integrity described above. The most straightforward solution is to reverse the proposed optimization by Tilscher et al. [46] and refrain from removing unknowns from `point` once they have been added to this set. As a result, the plain TD with widening computes partially correct results, but looses accuracy in the way shown by the authors.

Another, more brute-force approach specifies the inequality $'d_{\mathrm{x}} \leq \sigma\,\mathrm{x}$ as fixpoint condition. On termination of an iterate phase, the solver returns the looked-up value $d_{\mathrm{x}} := \sigma\,\mathrm{x}$. A fixpoint condition of this kind prevents any future improvement in the values of widening points and therefore guarantees partial correctness. Note that a proof along the lines of Theorem 1 by Stade et al. [44] introduces the necessity to prove $\mathrm{x} \notin \mathrm{s}$ for subcase *Widening Continue* in case *Iterate*. The solution criterion now relaxed to a post-solution criterion thereby requires for a stronger widening constraint: $b \leq a \implies a \nabla b = a$. We remark that such a condition is feasible for real-world applications and consistent with the initial intention of widening. As one would expect, however, this adaptation may also result in a drastic loss of precision and changes termination properties of the solver. In addition, the inequality of the fixpoint condition generally conflicts with an extension by the narrowing operator.

A more sophisticated implementation follows the idea underlying the verification of the plain TD by Stade et al. [44, Theorem 1 (Partial Correctness of the plain TD)]. The implicit stable set s utilized in their proof collects all unknowns occurring in previous iterations, as well as those currently under evaluation. This set s is now introduced as an explicit parameter of the solver. In consequence, the function `iterate` is adapted to:

```
iterate x $c$ s p $\sigma$ =
  if x $\notin$ s then
    let ($''d_{\mathrm{x}}$, s', p', $\sigma$) = eval x (T x) $c$ (s $\cup$ {x}) p $\sigma$ in
    let $'d_x$ = if x $\notin$ p then $''d_x$ else $\sigma$ x $\boxslash$ $''d_x$ in
    if $'d_x$ = $\sigma$ x then
      ($\sigma$ x, s', p' $-$ {x}, $\sigma$)
    else
      iterate x $c$ s p' ($\sigma \oplus$ {x $\mapsto$ $'d_x$})
  else ($\sigma$ x, s, p, $\sigma$)
```

Note how unknown x is permanently added to the implicit stable set s once a fixpoint is reached and hence never reevaluated later. As it turns out, the implementation is closely related to our first proposal, where unknowns are permanently added to `point`. This concludes our discussion on the extension of the solver plain TD with warrowing.

In the following, recall that warrowing was introduced in Section 2.6 as a more robust operator than narrowing, in that the former guarantees partial correctness even in the presence of non-monotonic systems of equations. Non-monotonic equations are indispensable for a number of static analyses problems, notably context-sensitive analysis such as interprocedural analysis [15] or the analysis of concurrent systems [48]. Yet at the same time, the warrowing operator exhibits a major weakness in terms of its termination characteristics: If the dynamic selection of its suboperators ∇ and Δ leads to a cycle of calculated values, then the solver may not terminate.

Example 6 (adapted from [42]): Let $\mathcal{U} = \{x\}$ and $\mathbb{D} = \mathbb{N} \cup \{\infty\}$ with the bottom element $\bot_\mathbb{D} := 0$ and a dedicated top element $\top_\mathbb{D} := \infty$. For values $a, b \in \mathbb{D}$, let

$$\nabla := \begin{cases} \infty & \text{if } a < b \\ a \sqcup b & \text{otherwise} \end{cases} \qquad \text{and} \qquad \Delta := \begin{cases} b & \text{if } a = \infty \\ a \sqcap b & \text{otherwise.} \end{cases}$$

Consider the single equation

$$\mathsf{x} = (\mathsf{if}\ \mathsf{x} = 0\ \mathsf{then}\ 1\ \mathsf{else}\ 0).$$

Obviously, any assignment $\sigma\colon \mathcal{U} \to \mathbb{D}$, $\sigma := \{x \mapsto d\}$ where $d \geq 1$ is a valid post-solution to the present system of equations. An iteration on x started by the warrowing TD first computes the values $\bot_\mathbb{D}$ and $\top_\mathbb{D}$, where the latter constitutes a correct result value. But in an attempt to improve the overly general value ∞, the solver will then apply narrowing. This yields the cyclical behavior

$$0 \to \infty \to 0 \to \infty \to 0 \to \infty \to \ldots$$

and the solver thus never terminates.

Still, termination of the warrowing TD can be enforced even for non-monotonic equations if a few assumptions are met. An inherently inevitable condition of top-down solving is that *(i)* only finitely many unknowns may be encountered during the evaluation of strategy trees. In case of an infinite domain $\mathbb{D}$, both the widening and narrowing operator must ensure *(ii)* arbitrary values in $\mathbb{D}$ are only in- or decreased finitely often. Such a restriction is made with regard to Kleene's Fixpoint Theorem [28]. Let $f\colon \mathbb{D} \to \mathbb{D}$ denote a monotonic function whose iterative application results in a finite ascending chain of values. Then a fixpoint is reached in a finite amount of function applications $n \in \mathbb{N}$ [28, 14]. With this in mind, we *(iii)* also restrict the number of switches between widening and narrowing phases. For example, an alternative implementation of the TD [42] features a single widening phase per iteration, followed by a single narrowing phase. Another possibility involves the introduction of a counter variable that terminates the solver after a defined number of switches. Without proof, we claim that such a solver terminates and returns partially correct results if

(iv) There exists a maximum element of the domain, i.e., $\top_{\mathbb{D}}$ is defined. Note that this proposition directly follows from assumption *(ii)*.

(v) In case the termination is triggered by the counter variable exceeding the threshold of maximum switches, the solver must terminate with a completed widening phase. A widening phase is completed if the solver would apply narrowing in the next iteration step.

That being said, we remark that for verifying soundness, that is, the correctness of the solver conditioned on termination, assumptions on the domain of values $\mathbb{D}$ can be further relaxed. A close inspection of Theorem 1 reveals that its proof is based only on parts of the assumptions about the narrowing operator, namely

$$b \leq a \implies b < a \,\Delta\, b \tag{9}$$

for $a, b \in \mathbb{D}$. Above condition is applied in Lemma 2 and Lemma 3. Due to the design of the warrowing operator, narrowing is only dispatched to if $b \leq a$. Thus $b = a \sqcap b$ holds and the requirement for the existence of a pairwise supremum operator is obsolete. The second part of the assumptions $a \,\Delta\, b \leq a$ (see definition 7), on the other hand, is only referred to once in the proof of Lemma 4. The preservation of fixpoints stated in Lemma 4 finds application in proofs concerning the precision of results and termination properties of the solver, but is of no relevance to our work. In our formalization in Isabelle [18], we therefore assume $\mathbb{D}$ to form a join-semilattice and fix $\bot_{\mathbb{D}}$ as dedicated bottom element. In addition, the choice of narrowing operator is solely limited by inequality 9.

4 Related Work

Our work is closely related to that of Stade et al. [44], whose verification we extend by widening and narrowing. The changes made can be broken down into three core measures: First, we combine their standalone correctness proof of the plain TD with their equivalence proof of the plain TD and the vanilla TD. This gives us a standalone proof of the partial correctness of the vanilla TD. We then generalize the proof such that the correctness criterion is already fulfilled if the returned assignment is a post-solution, hence accounting for the solver to overapproximate solutions. Lastly, the algorithm of the fixpoint engine TD is extended by warrowing. The proof of soundness is adjusted where necessary, namely by the extension with two new subcases *Warrowing Continue* and *Warrowing Fixpoint* (see also Section 3.3).

It is particularly interesting to note that, parallel to the implementation of the vanilla TD by Stade et al. [44], local parameters such as the `point` set could be realized by references to mutable data structures. The warrowing TD as presented in Chapter 3 can therefore be implemented in a very memory-efficient way.

With regard to generic fixpoint engines, correctness and termination of various algorithms have been verified by machine-checked proofs. This includes round-robin iteration [7], variations of worklist iteration [35, 11], [5] (correctness only) and the local generic fixpoint solver RLD [24]. The solver RLD is closely related to the TD, but destabilizes directly influenced unknowns locally. Additionally, affected unknowns are reevaluated immediately by means of a local worklist. A major weakness of the RLD, however, is that multiple queries of the same unknown within a single right-hand side can result in different values [44]. The undesired behavior can be avoided by implementing extra measures [4]. Nevertheless, this characteristic leads to the realization that the RLD is not a chaotic iteration solver [6] and thus hardly compatible with widening and narrowing.

One of the most recent formalizations of a local generic solver was published by Vilhena et al. [47], who prove their solver's partial correctness using Iris. In retracing the reasoning in Section 2.2, we note that: While the representation as strategy trees is minimalistic and straightforward, the constraint of purity does not allow for right-hand sides to cause any side effects. This effectively deprives the TD of its ability to handle internal side effects, such as logging or the support of concurrency. Vilhena et al. [47] address the issue by proposing the concept of *apparent purity*: If two identical calls to a function terminate, they must return the same result. Yet functions can also have an internal state as long as it is properly encapsulated. In consequence, their evaluation of two semantically equivalent expressions may differ observationally. In comparison to the top-down solver TD, however, their implementation also loses general applicability due to a restriction to systems of equations with monotonic right-hand sides.

As an alternative to generic fixpoint engines, static analysis tools that focus on the abstract interpretation of a specific programming language often apply *syntax-directed* fixpoint iterators. In this context, a number of non-generic fixpoint engines have been verified [8, 9, 26, 17]. A formalization of particular interest for our thesis is that of a minimalistic abstract interpreter developed by Nipkow [34]. It is conceived for educational purposes and targeted at a simple while-language. Even though this solver is syntax-directed, their verification is closely related to ours in that they use Isabelle as their proof assistant and include a basic support of widening and narrowing. The original publication [34] formally verified correctness and supplied a technical (i.e. pen-and-paper-style) proof of termination. A machine-checked termination proof including widening and narrowing was supplemented later [36].

Despite its indispensability for certain abstract domains, solvers implementing widening and narrowing are rarely verified and if so, then only to a limited extent. Limited, as they either do not consider any guarantees regarding termination or exclusively apply an ascending acceleration, that is, widening [39, 26, 17]. An exception is the work of Pichardie (et al.): With a special interest in termination, they proposed various attempts of a certified abstract interpreter, including such that implement widening and narrowing [38, 9]. Regarding verifications of *generic* solvers with widening and narrowing, there exist several technical formalizations [3, 1, 2, 40, 42]. *Machine-checked* formalizations of generic solvers used in practice, on the other hand, are rare. To our knowledge, the warrowing TD constitutes one of the first implementations of a certified generic abstract interpretation framework implementing widening and narrowing.

5 Conclusion

In our thesis, we provided an extension of the vanilla TD presented by Stade et al. [44] with the warrowing operator and demonstrated how their proof structure can be extended to consider post-solutions. We then conducted an in-depth analysis of the warrowing operator and identified characteristic properties. Building on the existing verification, we ultimately showed the partial correctness of the warrowing TD.

Whereas memoization can be implemented to preserve the semantics of a solver, this is not the case with widening and narrowing. The warrowing TD does generally not compute the least solution – even for extended assumptions on widening and narrowing operators (Equation 6, 7) and monotonic systems of equations (see also Example 4). In consequence, derived claims on program states might be too weak to be of reasonable use for the overall problem statement. A detailed analysis and comparison of the quality of results under different parameters is advisable and remains for future research. Equally important, the termination properties discussed above should be formally verified as well. For monotonic systems of equations, this is possible without further adjustments. As an alternative, we outlined an adapted implementation in Section 3.4, which enforces termination for arbitrary equation systems by restricting the number of phase switches.

The consideration of warrowing as an independent operator in Section 3.2 presents opportunities for the further improvement of fixpoint engines. In combination with an identification of the minimal set of required properties in Section 3.3, it is possible to refine the warrowing operator in accordance with the application scenario. For example, a dynamic warrowing operator can be defined which, in the case of oscillating behavior of the solver, introduces additional thresholds and thereby reduces the deviation from the smallest solution (c.f. Example 6). The incomplete propagation of the `point` set in an implementation without memoization, on the other hand, indicates that the management of potential warrowing points may need to be revised. Inspired by the discussion in Section 3.4, a more sophisticated distinction could be made between genuine warrowing points and unknowns that were over-eagerly added to `point` in early iteration steps. If realizable, this can also improve the quality of results, as the deployment of accelerated but imprecise widening is further restricted.

Bibliography

[1] G. Amato, F. Scozzari, H. Seidl, K. Apinis, and V. Vojdani. "Efficiently intertwining widening and narrowing." In: *Science of Computer Programming* 120 (2016), pp. 1–24. ISSN: 0167-6423. DOI: `10.1016/j.scico.2015.12.005`.

[2] K. Apinis, H. Seidl, and V. Vojdani. "Enhancing Top-Down Solving with Widening and Narrowing." In: *Semantics, Logics, and Calculi: Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*. Ed. by C. W. Probst, C. Hankin, and R. R. Hansen. Cham: Springer International Publishing, 2016, pp. 272–288. ISBN: 978-3-319-27810-0. DOI: `10.1007/978-3-319-27810-0_14`.

[3] K. Apinis, H. Seidl, and V. Vojdani. "How to combine widening and narrowing for non-monotonic systems of equations." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 377–386. ISBN: 9781450320146. DOI: `10.1145/2491956.2462190`.

[4] A. Bauer, M. Hofmann, and A. Karbyshev. "On Monadic Parametricity of Second-Order Functionals." In: *Foundations of Software Science and Computation Structures*. Ed. by F. Pfenning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 225–240. ISBN: 978-3-642-37075-5.

[5] Y. Bertot, B. Grégoire, and X. Leroy. "A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis." In: *Types for Proofs and Programs*. Ed. by J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 66–81. ISBN: 978-3-540-31429-5.

[6] F. Bourdoncle. "Efficient chaotic iteration strategies with widenings." In: *Formal Methods in Programming and Their Applications*. Ed. by D. Bjørner, M. Broy, and I. V. Pottosin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 128–141. ISBN: 978-3-540-48056-3.

[7] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. "Extracting a Data Flow Analyser in Constructive Logic." In: *Programming Languages and Systems*. Ed. by D. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 385–400. ISBN: 978-3-540-24725-8.

[8] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. "Certified Memory Usage Analysis." In: *FM 2005: Formal Methods*. Ed. by J. Fitzgerald, I. J. Hayes, and A. Tarlecki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 91–106. ISBN: 978-3-540-31714-2.

[9] D. Cachera and D. Pichardie. "A Certified Denotational Abstract Interpreter." In: *Interactive Theorem Proving*. Ed. by M. Kaufmann and L. C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 9–24. ISBN: 978-3-642-14052-5.

[10] B. L. Charlier and P. Van Hentenryck. *A Universal Top-Down Fixpoint Algorithm*. Technical Report CS-92-25. Providence, Rhode Island, USA, May 1992.

[11] S. Coupet-Grimal and W. Delobel. "A Uniform and Certified Approach for Two Static Analyses." In: *Types for Proofs and Programs*. Ed. by J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 115–137. ISBN: 978-3-540-31429-5.

[12] P. Cousot. "Abstracting Induction by Extrapolation and Interpolation." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by D. D'Souza, A. Lal, and K. G. Larsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 19–42. ISBN: 978-3-662-46081-8.

[13] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks." In: *Journal of Logic and Computation* 2.4 (Aug. 1992), pp. 511–547. ISSN: 0955-792X. DOI: `10.1093/logcom/2.4.511`.

[14] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: `10.1145/512950.512973`.

[15] P. Cousot and R. Cousot. "Static determination of dynamic properties of recursive procedures." English (US). In: *IFIP Conference on Formal Description of Programming Concepts, St. Andrews, N.B., Canada*. Ed. by E. Neuhold. North-Holland Publishing Company, 1977, pp. 237–277.

[16] C. Fecht and H. Seidl. "A faster solver for general systems of equations." In: *Science of Computer Programming* 35.2 (1999), pp. 137–161. ISSN: 0167-6423. DOI: `10.1016/S0167-6423(99)00009-X`.

[17] L. Franceschino, D. Pichardie, and J.-P. Talpin. "Verified Functional Programming of an Abstract Interpreter." In: *Lecture Notes in Computer Science*. Springer International Publishing, 2021, pp. 124–143. ISBN: 9783030888060. DOI: `10.1007/978-3-030-88806-0_6`.

[18] A. Graß. *Partial Correctness of the Warrowing TD*. Available at `https://github.com/AlexandraGrass/warrowing-td-verification`. 2024.

[19] M. S. Hecht and J. D. Ullman. "A Simple Algorithm for Global Data Flow Analysis Problems." In: *SIAM Journal on Computing* 4.4 (1975), pp. 519–532. DOI: `10.1137/0204044`.

[20] K. S. Henriksen and J. P. Gallagher. "Abstract Interpretation of PIC Programs through Logic Programming." In: *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. 2006, pp. 184–196. DOI: `10.1109/SCAM.2006.1`.

[21] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. "An overview of Ciao and its design philosophy." In: *Theory and Practice of Logic Programming* 12.1–2 (2012), pp. 219–252. DOI: `10.1017/S1471068411000457`.

[22] M. Hermenegildo. "Parallelizing irregular and pointer-based computations automatically: Perspectives from logic and constraint programming." In: *Parallel Computing* 26.13 (2000). Parallel Computing for Irregular Applications, pp. 1685–1708. ISSN: 0167-8191. DOI: `10.1016/S0167-8191(00)00051-X`.

[23] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. "Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor)." In: *Science of Computer Programming* 58.1 (2005). Special Issue on the Static Analysis Symposium 2003, pp. 115–140. ISSN: 0167-6423. DOI: `10.1016/j.scico.2005.02.006`.

[24] M. Hofmann, A. Karbyshev, and H. Seidl. "Verifying a Local Generic Solver in Coq." In: *Static Analysis*. Ed. by R. Cousot and M. Martel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 340–355. ISBN: 978-3-642-15769-1.

[25] M. Hofmann, A. Karbyshev, and H. Seidl. "What Is a Pure Functional?" In: *Automata, Languages and Programming*. Ed. by S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, and P. G. Spirakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 199–210. ISBN: 978-3-642-14162-1. DOI: `10.1007/978-3-642-14162-1_17`.

[26] J.-H. Jourdan. "Verasco: a Formally Verified C Static Analyzer." Theses. Universite Paris Diderot-Paris VII, May 2016.

[27] G. A. Kildall. "A unified approach to global program optimization." In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '73. Boston, Massachusetts: Association for Computing Machinery, 1973, pp. 194–206. ISBN: 9781450373494. DOI: `10.1145/512927.512945`.

[28] S. C. Kleene. *Introduction to Metamathematics*. Groningen: P. Noordhoff N.V., 1952.

[29] M. Méndez-Lojo, J. Navas, and M. V. Hermenegildo. "A Flexible, (C)LP-Based Approach to the Analysis of Object-Oriented Programs." In: *Logic-Based Program Synthesis and Transformation*. Ed. by A. King. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 154–168. ISBN: 978-3-540-78769-3.

[30] A. Miné. "The octagon abstract domain." In: *Proceedings Eighth Working Conference on Reverse Engineering*. 2001, pp. 310–319. DOI: `10.1109/WCRE.2001.957836`.

[31] K. Muthukumar and M. Hermenegildo. "Compile-time derivation of variable dependency using abstract interpretation." In: *The Journal of Logic Programming* 13.2 (1992), pp. 315–347. ISSN: 0743-1066. DOI: `10.1016/0743-1066(92)90035-2`.

[32] K. Muthukumar and M. V. Hermenegildo. *Deriving a fixpoint computation algorithm for top-down abstract interpretation of logic programs*. Technical Report. Madrid, Spain: Informatica, Apr. 1990.

[33] K. Muthukumar and M. V. Hermenegildo. "Determination of variable dependence information through abstract interpretation." In: *Logic Programming, Proceedings of the North American Conference 1989*. MIT Press, Oct. 1989. ISBN: 0262620642.

[34] T. Nipkow. "Abstract Interpretation of Annotated Commands." In: *Interactive Theorem Proving*. Ed. by L. Beringer and A. Felty. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 116–132. ISBN: 978-3-642-32347-8.

[35] T. Nipkow. "Verified Bytecode Verifiers." In: *Foundations of Software Science and Computation Structures*. Ed. by F. Honsell and M. Miculan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 347–363. ISBN: 978-3-540-45315-4.

[36] T. Nipkow and G. Klein. "Abstract Intepretation." In: *Concrete Semantics with Isabelle/HOL*. Springer Cham, Dec. 2014, pp. 219–280. ISBN: 978-3-319-10541-3. DOI: `10.1007/978-3-319-10542-0`.

[37] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer Berlin, Heidelberg, 2002. ISBN: 3-540-43376-7. DOI: `10.1007/3-540-45949-9`.

[38] D. Pichardie. "Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés." Thèse de doctorat. Universite de Rennes 1-Rennes I, 2005.

[39] V. Robert and X. Leroy. "A formally-verified alias analysis." In: *Proceedings of the Second International Conference on Certified Programs and Proofs*. CPP'12. Kyoto, Japan: Springer-Verlag, 2012, pp. 11–26. ISBN: 9783642353079. DOI: `10.1007/978-3-642-35308-6_5`.

[40] S. Schulze Frielinghaus, H. Seidl, and R. Vogler. "Enforcing Termination of Interprocedural Analysis." In: *Static Analysis*. Ed. by X. Rival. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 447–468. ISBN: 978-3-662-53413-7.

[41] M. Schwarz, S. Saan, H. Seidl, K. Apinis, J. Erhard, and V. Vojdani. "Improving Thread-Modular Abstract Interpretation." In: *Static Analysis*. Ed. by C. Drăgoi, S. Mukherjee, and K. Namjoshi. Cham: Springer International Publishing, 2021, pp. 359–383. ISBN: 978-3-030-88806-0.

[42] H. Seidl and R. Vogler. "Three Improvements to the Top-Down Solver." In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. PPDP '18. Frankfurt am Main, Germany: Association for Computing Machinery, 2018. ISBN: 9781450364416. DOI: `10.1145/3236950.3236967`.

[43] M. Sintzoff. "Calculating properties of programs by valuations on specific models." In: *Proceedings of ACM Conference on Proving Assertions about Programs*. Las Cruces, New Mexico, USA: Association for Computing Machinery, 1972, pp. 203–207. ISBN: 9781450378918. DOI: `10.1145/800235.807086`.

[44] Y. Stade, S. Tilscher, and H. Seidl. "The Top-Down Solver Verified: Building Confidence in Static Analyzers." In: *Computer Aided Verification*. To be published. 2024.

[45] P. Thomson. "Static Analysis: An Introduction: The fundamental challenge of software engineering is one of complexity." In: *Queue* 19.4 (Sept. 2021), pp. 29–41. ISSN: 1542-7730. DOI: `10.1145/3487019.3487021`.

[46] S. Tilscher, Y. Stade, M. Schwarz, R. Vogler, and H. Seidl. "The Top-Down Solver—An Exercise in A^2I." In: *Challenges of Software Verification*. Ed. by V. Arceri, A. Cortesi, P. Ferrara, and M. Olliaro. Singapore: Springer Nature Singapore, 2023, pp. 157–179. ISBN: 978-981-19-9601-6. DOI: `10.1007/978-981-19-9601-6_9`.

[47] P. E. de Vilhena, F. Pottier, and J.-H. Jourdan. "Spy game: verifying a local generic solver in Iris." In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: `10.1145/3371101`.

[48] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. "Static race detection for device drivers: the Goblint approach." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE '16. Singapore: Association for Computing Machinery, 2016, pp. 391–402. ISBN: 9781450338455. DOI: `10.1145/2970276.2970337`.