TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK
Lehrstuhl für Sprachen und Beschreibungsstrukturen

Interprocedural Program Analysis:
Herbrand Equalities and Local Solvers

Stefan Schulze Frielinghaus

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr. Francisco Javier Esparza Estaun

Prüfer der Dissertation:

1. Prof. Dr. Helmut Seidl
2. Prof. Dr. Markus Müller-Olm

Die Dissertation wurde am 25.07.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 22.12.2017 angenommen.

Abstract

This thesis covers two aspects of interprocedural program analysis. In the first part we concentrate on pure syntactical relations between program variables and introduce a novel analysis in order to compute all interprocedural valid Herbrand equalities for programs where right-hand sides contain at most one program variable (which can occur several times). The novel analysis is based on procedure summaries representing the weakest preconditions for finitely many generic postconditions. In order to arrive at effective representations for all occurring weakest preconditions, we show for almost all run-time values possibly computed by the program, that they can be uniquely factorized into tree patterns and a terminating ground term. Moreover, we introduce an approximate notion of subsumption which is effectively decidable and ensures that finite conjunctions of equalities may not grow infinitely. Based on these technical results, we realize an effective fixed point iteration. Finally we show that a two-variable invariant candidate can be verified in time polynomial in the size of the program and in the size of the invariant candidate. Furthermore, we show that a multi-variable invariant candidate can be verified in time polynomial in the size of the program and in the size of the invariant candidate, and only exponentially in the number of variables of the invariant candidate.

In the second part of this thesis we perform interprocedural program analysis by means of partial tabulation of procedure summaries. Such an analysis might not terminate if a procedure is analyzed for infinitely many calling contexts or when the domain has infinite strictly ascending chains. As a remedy we provide two local solvers for general equation systems, be they monotone or not, and prove that these solvers only fail to terminate, if infinitely many variables are encountered during a run. We show that interprocedural analysis performed by these local solvers is guaranteed to terminate for all non-recursive programs. Moreover, for recursive programs we are still able to provide termination guarantees by over-approximating calling contexts such that the number of contexts for each procedure is kept finite.

Zusammenfassung

In dieser Thesis werden zwei Aspekte der interprozeduralen Programmanalyse behandelt. Im ersten Teil konzentrieren wir uns auf rein syntaktische Relationen zwischen Programmvariablen und führen eine neue Analyse ein, um alle interprozedural gültigen Herbrand-Gleichungen für Programme zu berechnen, bei denen rechte Seiten von Zuweisungen höchstens eine Programmvariable enthalten (die mehrmals vorkommen kann). Die neue Analyse basiert auf Prozedurzusammenfassungen, die die schwächsten Vorbedingungen für endlich viele generische Nachbedingungen darstellen. Um die schwächsten Vorbedingungen effektiv darzustellen, zeigen wir für fast alle Laufzeitwerte, dass diese eindeutig in Baummuster und einen abschließenden Term faktorisiert werden können. Darüber hinaus stellen wir eine Idee der approximierenden Subsumtion vor, die effektiv entscheidbar ist und sicherstellt, dass endliche Konjunktionen von Gleichungen nicht unendlich wachsen können. Basierend auf diesen Formalismen realisieren wir eine effektive Fixpunkt-Iteration. Schließlich zeigen wir, dass ein Zwei-Variablen-Invariantenkandidat in der Zeit polynomiell in der Größe des Programms und in der Größe des Invariantenkandidaten überprüft werden kann. Darüber hinaus zeigen wir, dass ein Mehr-Variablen-Invariantenkandidat in der Zeit polynomiell in der Größe des Programms und in der Größe des Invariantenkandidaten und nur exponentiell in der Anzahl der Variablen des Invariantenkandidaten überprüft werden kann.

Im zweiten Teil dieser Arbeit führen wir eine interprozedurale Programmanalyse basierend auf partieller Tabellierung von Prozedurzusammenfassungen durch. Eine solche Analyse terminiert nicht, wenn eine Prozedur für unendlich viele Aufrufkontexte analysiert wird oder wenn die Domäne unendlich streng aufsteigende Ketten hat. Als Lösung stellen wir zwei bedarfsgetriebene Gleichungslöser für allgemeine Gleichungssysteme vor, seien sie monoton oder nicht und zeigen, dass diese Gleichungslöser nur dann nicht terminieren, wenn während eines Durchlaufes unendlich viele Variablen vorkommen. Des Weiteren zeigen wir, dass die interprozedurale Analyse von nicht rekursiven Programmen, die anhand der neuen Gleichungslöser durchgeführt wird, immer terminiert. Darüber hinaus können wir für rekursive Programme durch Überapproximation von Aufrufkontexten Terminierung garantieren, sodass die Anzahl an Kontexten für jede Prozedur endlich ist.

List of Original Publications

This thesis includes the content of the following four publications:

Stefan Schulze Frielinghaus, Michael Petter, and Helmut Seidl
"Inter-procedural Two-Variable Herbrand Equalities"
In: *Proceedings of the 24th European Symposium on Programming, ESOP*.
Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science.
Springer, 2015, pp. 457–482. DOI: 10.1007/978-3-662-46669-8_19.

Stefan Schulze Frielinghaus, Michael Petter, and Helmut Seidl
"Inter-procedural Two-Variable Herbrand Equalities"
In: *Logical Methods in Computer Science* 13.2 (2017).
DOI: 10.23638/LMCS-13(2:5)2017.

Stefan Schulze Frielinghaus, Helmut Seidl, and Ralf Vogler
"Enforcing Termination of Interprocedural Analysis"
In: *Proceedings of the 23rd International Symposium on Static Analysis, SAS*.
Ed. by Xavier Rival. Vol. 9837. Lecture Notes in Computer Science.
Springer, 2016, pp. 447–468. DOI: 10.1007/978-3-662-53413-7_22.

Stefan Schulze Frielinghaus, Helmut Seidl, and Ralf Vogler
"Enforcing Termination of Interprocedural Analysis"
In: *Formal Methods in System Design* (2017).
DOI: 10.1007/s10703-017-0288-5.

Contents

1 Introduction

Static program analysis is the art of automated reasoning about programs without actually executing them. That means by analyzing a program the meaning or certain properties of a program are inferred. One goal of doing so is the optimization of a program by a compiler, e.g., to eliminate dead code, compute reachability information, minimization of registers, and so on. Other interesting goals are the detection of bugs and errors in general or even to verify the correctness of a program, i.e., proving the absence of errors. In contrast to program testing, where the programmer has to specify test cases manually in order to provide certain guarantees, program analysis is meant to be done automatically, i.e., by programs itself.

The limits of automatic reasoning about programs are already well known from the computability theory results from Turing [Tur37] and Rice [Ric53]. They showed that all non-trivial[1] semantic properties of programs written in a Turing complete programming language are mathematically undecidable. Still, static program analysis can reveal certain kinds of semantic properties by being more conservative, i.e., by taking all possible and also some impossible program executions into account—the latter clearly leads to some imprecision. For example, it is undecidable whether or not a program variable is constant at some program point. Still an effective analysis could compute *yes* or *no* for a program variable and program point or also *maybe*—indicating that a program variable may be constant or not. Therefore, by giving up precision it is still possible to come up with effective analyses.

In the following we consider two aspects of interprocedurel analysis. First we consider *Herbrand Equalities* and afterwards we have a look at *Local Solvers*.

Herbrand Equalities An alternative approach to introducing imprecision, is to consider only certain kinds of programs for which a precise analysis is possible, instead of considering general programs. Consider the program of Figure 1.1 which contains only right-hand sides that depend on at most one program variable. Inside procedure p the global variable $\mathbf{x}$ is at first assigned

[1] A property is termed as non-trivial if it is neither true for every program, nor for no program.

```
0:  global x, y;
1:  main() {
2:      x := f(a, a);
3:      y := a;
4:      p();
5:      assert (x = f(y, y));
6:  }

7:  p() {
8:      if (*) {
9:          x := g(x, x);
10:         x := f(x, x);
11:         p();
12:         y := f(y, y);
13:         y := g(y, y);
14:     }
15: }
```

Figure 1.1: A recursive program with binary operators f and g

the result of the binary operator g applied to $\mathbf{x}$. Subsequently the result of the binary operator f applied to $\mathbf{x}$ is assigned to $\mathbf{x}$. Then, after the recursive call, the global variable $\mathbf{y}$ is also assigned the results of the binary operators g and f but in reverse order, i.e., first operator f is applied and then g. The procedure p is called exactly one time in procedure *main* and is called arbitrarily often recursively in different calling contexts by making use of non-deterministic branching. At program point 5 we are faced with the question whether or not $\mathbf{x} = f(\mathbf{y}, \mathbf{y})$ is an invariant, i.e., does the equality hold for *all* program executions. The challenging problems about this question are that a recursive procedure—without any obvious termination argument—is involved and that the semantics of the operators g and f are unknown. Therefore we cannot assume that the operators satisfy any obvious algebraic laws and the equality must hold independent of the interpretation of the operators. Still the invariant can be shown if the program variables $\mathbf{x}$ and $\mathbf{y}$ are computed by means of syntactical identical terms of operator applications. Such equalities are called *Herbrand equalities* which we make use of in Chapter 2 of this thesis. There we introduce a novel analysis in order to compute all valid interprocedural Herbrand equalities for programs which contain only right-hand sides which depend on at most one program variable.

In order to arrive at this result we make use of *weakest precondition* computation which is related to the work of Hoare. In 1969, Hoare introduced a formal system which rigorously defined the semantics of an imperative programming language by introducing a set of logical deduction rules [Hoa69]. His work has been influenced by Floyd who published a similar system for flowcharts [Flo67]. Since then Hoare logic has been further developed. One famous notion are

predicate transformers which have been introduced by Dijkstra in 1975 [Dij75]. Predicate transformers assign meaning to each statement of an imperative programming language, i.e., a transformer is a total function over the state space of statements. In contrast to Hoare logic, which is a deductive system and closely related to axiomatic semantics, predicate transformers are a kind of denotational semantics. The difference between these two systems can also be seen as follows. Predicate transformers reduce the problem of proving a Hoare triple to the problem of proving a formula. That means, predicate transformers are a reformulation of Hoare logic. There exist two well known predicate transformers: *weakest precondition* and *strongest postcondition* transformer. The former has been introduced by Dijkstra which we make use of in Chapter 2 of this thesis.

The introduction of weakest precondition computation by Dijkstra was in the setting of intraprocedural analyses. These kinds of analyses are typically easier to formulate when there are no procedures or where procedure calls are handled as black boxes. In the latter case, an analysis interprets a procedure call as an unknown statement, and in order to be sound, all accumulated information before a call must be discarded after a call. Or in other words, all information is lost at procedure boundaries. For our example program such an analysis is not suitable in order to prove the invariant candidate at line 5. Therefore, in order to come up with an analysis for our problem, we still have to come up with a solution for procedures.

In contrast to intraprocedural analyses, *interprocedural* analyses take procedures into account. Some programs with multiple procedures can be massaged into programs with exactly one procedure where again techniques from intraprocedural analyses can be applied. Consider a program with non-recursive procedures and static procedure calls only, i.e., for each procedure call it can be statically determined which particular procedure is called. Then by inlining each procedure into its corresponding procedure call, the initial program is transformed into an equivalent program with exactly one procedure—although the size of the resulting program might by exponentially larger. For such programs, intraprocedural analyses might be suitable. Clearly, for our example recursive program this approach fails and more advanced interprocedural analyses are required.

Interprocedural analyses for imperative programs have been extensively studied since the 70s [All74; CC77c; Lom77; Bar77; Gal78; Ros79]. In 1981, Sharir and Pnueli introduced in their seminal paper [SP81] two approaches to interprocedural analysis. The first approach is called *functional* approach where procedures are interpreted as one huge block of statements. For each

such block, the input/output relation, i.e., the summary, is computed. Then, for each procedure call, the program state after the call is determined by the relation and the program state before the call. Technically speaking, the relation is determined by tabulating all input/output pairs. However, in general this is not effective. If the lattice $\mathbb{D}$, describing the program states, contains infinite elements, then the relation in $\mathbb{D} \to \mathbb{D}$ contains infinite strictly increasing chains. Moreover, even if an iterative solution converges, then there is still a space bound. While determining a fixed point, the intermediate solution in $\mathbb{D} \to \mathbb{D}$ is not applied to elements in $\mathbb{D}$ but is manipulated. Hence for an effective analysis, we require some compact representation of functions in $\mathbb{D} \to \mathbb{D}$.

Assume that our goal is to tabulate all weakest preconditions of all postconditions which occur while proving the invariant candidate at line 5 of the example given in Figure 1.1. While computing the weakest precondition of the postcondition $\mathbf{x} \doteq f(\mathbf{y}, \mathbf{y})$ for procedure p we observe that we have to compute the weakest precondition of every postcondition $\mathbf{x} \doteq f(\mathbf{y}, \mathbf{y})$ where $\mathbf{y}$ is arbitrarily often substituted by the term $g(f(\mathbf{y}, \mathbf{y}), f(\mathbf{y}, \mathbf{y}))$. Therefore, naive tabulation is ruled out since we would have to compute weakest preconditions for infinitely many postconditions. Still, we are able to circumvent this problem by using *generic* postconditions $A(\mathbf{x}) \doteq B(\mathbf{y})$ of which only finitely many exist. Basically, the idea is that A and B are placeholders for arbitrary terms which contain no program variables but holes into which the arguments get substituted. We then have to deal with equalities of the form $A(s) \doteq B(t)$ where s and t are terms containing at most one program variable. In Chapter 2 we provide an approximate notion of subsumption for such two-variable Herbrand equalities which is strong enough to guarantee that every occurring conjunction of two-variable Herbrand equalities is equivalent to a finite subset. By this subsumption and compactness result we arrive at an effective analysis and are ultimately able to determine for every program point *all* valid Herbrand equalities—at least for programs which contain at most one program variable in each right-hand side of assignments. This enables us to finally prove the invariant candidate $\mathbf{x} = f(\mathbf{y}, \mathbf{y})$ at program line 5 of the example given in Figure 1.1. A proof of the invariant candidate in the sense of a weakest precondition computation is presented in Section 2.7.

We study the complexity of the presented algorithms in Section 2.11 and observe that the terms during the weakest precondition computation may grow exponentially. Nevertheless, by compact representations of terms we are able to verify a two-variable invariant candidate in time polynomial in the size of the program and in the size of the invariant candidate. Furthermore, we show

that a multi-variable invariant candidate can be verified in time polynomial in the size of the program and in the size of the invariant candidate, and only exponentially in the number of variables of the invariant candidate.

Local Solvers For interprocedural analysis, however, it is often the case that effective representations of procedure summaries are not known. This is already the case, e.g., for Constant Propagation since the complete lattice there is infinite. For such a case Sharir and Pnueli introduce their second method termed the *call-string* approach. Basically, the idea is that data-flow values are tagged with the history of procedure calls in order to make interprocedural flow explicit. That means, whenever a procedure is entered or is returned from, the history of procedure calls—termed the call-string—is updated. By that, procedure calls and returns are treated like jump edges with the addition, that the call-string is updated, if we neglect parameters and local variables for the moment. One could also think of virtually copying a procedure into its corresponding procedure call. This approach fails for recursive programs. That is why Sharir and Pnueli propose the *call-string suffix approximation*, or also known as the k-call-string approach. In order to distinguish between procedure calls, a sequence of at most k call sites is kept. That means, two procedure calls to the same procedure are kept distinct, if their k-most recent call sites are different. By that interprocedurally invalid paths are possible, rendering the solution imprecise. However, the benefit is that only finitely many call-strings are possible, which makes this approach suitable even for recursive procedures.

Consider the example program given in Figure 1.2. In order to show that the assertion at line 4 holds, it is enough to perform interval analysis in conjunction with the k-call-string approach where $k = 0$. However, a priori it is unclear whether or not a $k > 0$ could improve precision. Even worse, for a $k > 1$ the procedure p is analyzed for $k + 1$ different call-strings, although

```
0:  global x;
1:  main() {
2:      x := 0;
3:      p();
4:      assert(0 ≤ x ≤ 1);
5:  }
6:  p() {
7:      if (x ≠ 0) { x := 0; }
8:      else       { x := 1; }
9:      if (∗)     { p(); }
10: }
```

Figure 1.2: Recursive procedure p is called in two different contexts only

the procedure is only called in two different contexts where the single global program variable **x** either equals 0 or 1. Therefore, the procedure is analyzed many times although the result of it only depends on two calling contexts. This is a general drawback of the call-string approach. Assume two different call sites to procedure p for the same calling context, then procedure p is analyzed two times—although the result will be the same for both call sites. Therefore, another approach is to separate data-flow values not by the history of procedure calls but by the context in which a procedure is called. By that we *partially* tabulate a procedure summary.

Following the idea presented in [CC77c] an analysis is compiled into a system of equations with variables in $\mathbb{N} \times \mathbb{D}$ where $\mathbb{N}$ is the set of program points and $\mathbb{D}$ is the domain of possible calling contexts. Then each variable $\langle u, d \rangle \in \mathbb{N} \times \mathbb{D}$ characterizes the value for the program point u, if the corresponding procedure is called in context d. This leads to a system of equations of the form

$$\langle u, d \rangle = f_{\langle u,d \rangle}, \qquad (u \in \mathbb{N}, d \in \mathbb{D})$$

where $f_{\langle u,d \rangle}$ is the defining right-hand side of a variable $\langle u, d \rangle$.

The domain $\mathbb{D}$ for an analysis can be determined by the idea of *Abstract Interpretation* which was introduced by the landmark paper [CC77a] from Patrick and Radhia Cousot in 1977. The basic idea is that an analysis executes a program, however, instead of computing with concrete values, the analysis computes with abstract values which *describe* concrete values. That means, in a first step, the *concrete semantics* is defined. In order to do so, the set of possible program states is determined which serves as the set of concrete values. The semantics of each program statement is then defined by a corresponding total function between concrete values. Though, the concrete semantics is in general not computable in finite time, i.e., the least or greatest fixed point cannot be computed of a given function of the concrete semantics in finite time. Therefore, in a second step, the *abstract semantics* is defined which introduces some kind of imprecision by over-approximating the concrete semantics. By the right choice of imprecision, the abstract semantics is computable in finite time. Similar to the concrete semantics, for the abstract semantics a set of abstract values is defined such that each abstract value describes one ore more concrete values. Then for each concrete function f a corresponding abstract function $f^\sharp$ between abstract values is defined such that whenever an abstract value d describes a concrete value c, then $f^\sharp(d)$ describes the concrete value $f(c)$. In other words, the abstract semantics simulates the concrete semantics.

If the domain $\mathbb{D}$ is infinite, then the system of equations is infinite. Thus well known fixed point algorithms like Round-Robin Iteration, Worklist Algorithm,

or Chaotic Iteration fail to compute a solution in finite time. However, an infinite system of equations often encodes procedure calls for certain contexts which never occur during any run of the program. For our example from Figure 1.2 we only have to consider the two calling contexts for procedure p where $\mathbf{x}$ is set to 0 or 1. In general, if a program is analyzed for a particular start context, only those contexts for procedure calls are of interest, which are *reachable* from program start. Therefore, instead of solving an infinite system of equations completely, basically the idea is to begin with a particular variable, as e.g. $\langle s_{main}, d_0 \rangle$ where s_{main} is the start node of the dedicated procedure *main* and $d_0 \in \mathbb{D}$ is an initial context for the program. Then new variables have to be solved which are encountered while evaluating right-hand sides. Hence, only those variables are solved which are necessary in order to solve the initial variable. Solvers for such infinite equation systems are known as *local solvers* [ASV13; Ama+16; Her+05].

In the classical setting of abstract interpretation widening and narrowing are done in separate phases. This has been given up by the local solver SLR_3 [Ama+16] where the widening and narrowing operators are intertwined into one operator. By this approach the abstract semantics follows more closely the concrete semantics and precision is not unnecessarily given up which is difficult to recover later. The drawback of the solver SLR_3 is, that termination is not guaranteed for non-monotone equation systems where it is possible that for a variable of the equation system the algorithm switches infinitely often between widening and narrowing. However, interprocedural analysis in the style of [ASV12] where partial tabulation of procedure summaries is used, introduces non-monotone right-hand sides of equation systems.

In Chapter 3 of this thesis we introduce two novel local solvers which are guaranteed to terminate—regardless if right-hand sides of equation systems are monotone or not—as long as only finitely many variables are encountered during a run. The first solver TSTP strictly separates widening and narrowing into different phases. The novel point is that both iterations are performed in a demand-driven way so that also during the narrowing phase fresh variables may be encountered for which no sound over-approximation has yet been computed. Subsequently the solver TSMP intertwines widening and narrowing, similar as it is done for the solver SLR_3, but with additional logic in order to decide when widening or narrowing should be applied such that termination is guaranteed. At first we present the two local solvers in the setting where a Galois connection is present and show that if local solving terminates, then each computed (partial) solution is sound. Secondly we drop the requirement of a Galois connection, i.e., we consider the case when no abstraction function

α is available, and show that the solvers remain sound when widening and narrowing operators are used which are sound w.r.t. the concretization. Finally we show that interprocedural analysis in the style of [CC77c; ASV12] using the novel local solvers terminates for all non-recursive programs. For recursive programs we present an approach to keep the number of contexts for every procedure finite, similar to [PAH06], by over-approximating them such that termination is also guaranteed in this case.

2 Interprocedural Two-Variable Herbrand Equalities

How can we infer that an equality such as $\mathbf{x} \doteq f(\mathbf{y}, \mathbf{y})$ holds at some program point, if the operators by which the program variables $\mathbf{x}$ and $\mathbf{y}$ are computed, do not satisfy obvious algebraic laws? This is the case, e.g., when either very high-level operations such as sqrt, or very low-level operations such as bit-shift are involved or, generally, for floating-point calculations. Still, the equality $\mathbf{x} \doteq f(\mathbf{y}, \mathbf{y})$ can be inferred, if $\mathbf{x}$ and $\mathbf{y}$ are computed by means of *syntactically* identical terms of operator applications. The equality then is called *Herbrand* equality. The problem of inferring valid Herbrand equalities dates back to [CS70] where it was introduced as the famous *value numbering* problem. Since quite a while, algorithms are known which, in absence of procedures, infer *all* valid Herbrand equalities [Kil73; SKR90]. These algorithms can even be tuned to run in polynomial time, if only invariants of polynomial size are of interest [GN04]. Surprisingly, little is known about Herbrand equalities if recursive procedure calls are allowed. In [MSS05] it has been observed that the intraprocedural techniques can be extended to programs with local variables and *functions*—but without global variables. The ideas there are strong enough to generally infer all Herbrand *constants* in programs with procedures and both local and global variables, i.e., invariants of the form $\mathbf{x} \doteq t$ where t is ground. Another tractable case of invariants is obtained if only assignments are taken into account whose right-hand sides have at most *one occurrence* of a variable [Pet10]. Thus, assignment $\mathbf{x} := f(\mathbf{y}, a)$ is considered while assignments such as $\mathbf{x} := f(\mathbf{y}, \mathbf{y})$ or $\mathbf{x} := f(\mathbf{y}, \mathbf{z})$ are approximated with $\mathbf{x} := ?$, i.e., by an assignment of an unknown value to $\mathbf{x}$. The idea is to encode ground terms as numbers. Then Herbrand equalities can be represented as polynomial equalities with a fixed number of variables and of bounded degree. Accordingly, techniques from linear algebra are sufficient to infer all valid Herbrand equalities for such programs. As a special case, the class of programs from [Pet10] subsumes those programs where only *unary* operators are involved. Such programs have been considered by [GT07]. Interestingly, the latter paper arrives at decidability by a completely different line of argument, namely, by

exploiting properties of the free monoid generated from the unary operators. Another avenue to decidability is to restrict the control structure of programs to be analyzed. In [GT09], the restricted class of *Sloopy* Programs is introduced where the format of loop as well as recursion is drastically restricted. For this class an algorithm is not only provided to decide arbitrary equalities between variables but also disequalities.

On the other hand, when only affine numerical expressions as well as affine program invariants are of concern, the set of valid invariants at a program point form a *vector space* which can be effectively represented. This observation is exploited in [MS04] to apply methods from linear algebra to infer all valid affine program invariants. These methods later have been adapted to the case where values of variables are not from a field, but where integers will overflow at some power of 2, i.e., are taken from a modular ring. Note that in the latter structure, some number different from 0 may be a zero divisor and thus does not have a multiplicative inverse [MS07]. For some applications, an analysis of *general* equalities is not necessary. In applications such as coalescing of registers [MS08] or detection of local variables in low-level code [Fle+11], it suffices to infer equalities involving two variables only. In the affine case, algorithms for inferring all two-variable equalities can be constructed which have better complexities than the corresponding algorithms for general equalities [Fle+11].

The question whether or not *all* interprocedurally valid Herbrand equalities can be inferred, is still open. In this part of the thesis, we consider the case of Herbrand equalities containing two variables only. These are equalities such as $\mathbf{x} \doteq f(g(\mathbf{y}), \mathbf{y}, a)$, i.e., right-hand sides of equalities may contain only a single variable, but this multiple times. Accordingly, in programs only assignments are taken into account whose right-hand sides contain (arbitrarily many) occurrences of at most one variable. The main result is that under this provision, *all* interprocedurally valid two-variable Herbrand equalities can be inferred.

The novel analysis is based on calculating weakest preconditions for all occurring postconditions. Since there may be infinitely many potential postconditions for a called procedure, we rely on *generic* postconditions to obtain finite representations of procedure summaries. In a generic postcondition *second-order* variables are used as place-holders for yet unknown relationships between program variables. In the generic postcondition

$$A(\mathbf{x}) \doteq B(\mathbf{y})$$

the second-order variables A and B take as values terms with (possibly multiple

occurrences of) *holes* (which we call *templates*). As preconditions we then get conjunctions of the following form

$$\bigwedge_i A(s_i) \doteq B(t_i)$$

where each term s_i, t_i contains at most one variable which might occur multiple times. To realize the algorithm for inferring all interprocedurally valid two-variable equalities, we thus require

- a method to finitely represent all occurring conjunctions of equalities;
- a method for proving that one conjunction subsumes another conjunction, i.e., a method to detect when the greatest fixed point computation has terminated;
- a guarantee that a fixed point will be reached in finitely many steps.

Note here that the equalities occurring during the weakest precondition computation of a generic postcondition may contain occurrences of second-order variables. Thus, subsumption between conjunctions of equalities is subtly related to second-order unification [Gol81]. Second-order unification asks whether a conjunction of equalities possibly containing second-order variables is satisfiable. Since long, it is known that generally, second-order unification is undecidable [Gol81]. Undecidability of second-order unification even holds if only a single unary second-order variable is involved [LV00]. In contrast, the problem of *context* unification, i.e., the variant of second-order unification where second-order variables range over terms with single occurrences of holes only, has recently been proven to be decidable [Jeż14]. It is worth mentioning that neither of the two cases directly applies to this application, since here we consider unary second-order variables (as context unification) which range over terms with one or multiple occurrences of holes (differently from context unification). Furthermore, the occurring terms contain at most one first-order variable whereas in [LV00] multiple first-order variables may occur. To the best of our knowledge, decidability of satisfiability is still open for our case.

Example 2.0.1. In the case presented here, during the weakest precondition computation a conjunction of the following form might occur:

$$A(a) \doteq B(f(a,a)) \wedge A(b) \doteq B(f(b,b))$$

where a and b are atoms. The (unique) solution for the second-order variables A and B is then given as

$$A = B(f(\bullet,\bullet))$$

where • denotes the hole. Since the hole occurs two times in the solution, the conjunction is not satisfiable, if only context unification is considered. ■

In this thesis, the satisfiability problem for the given unification problem is not solved. Instead, we introduce two novel ideas to circumvent this problem and still infer all interprocedurally valid two-variable Herbrand equalities. First, we introduce a notion of *approximate* subsumption. This means that the algorithm does not allow to prove implications between all conjunctions of equalities—but at least sufficiently many so that accumulation of *infinite* conjunctions is ruled out. Second, we note that subsumption is not required for arbitrary valuations of program variables. Instead it suffices to consider values which may possibly be constructed by the program at run-time. For programs where every right-hand side of assignments contain occurrences of single variables only, we observe that the ground terms possibly occurring at run-time, have a specific structure, which allows for a *unique factorization* of these terms into irreducible templates—at least, if these ground terms are sufficiently *large*. Our factorization result applied to these kind of values, enables us to make use of the monoidal methods of [GT07]. This approach, which works for sufficiently large terms, then is complemented with a dedicated treatment of finitely many exceptional cases. By that, we ultimately succeed to construct an effective approximate subsumption algorithm which allows us to restrict the number of equalities in occurring conjunctions and to determine all valid two-variable Herbrand equalities.

In order to arrive at the key result, namely an algorithm to infer all valid interprocedural two-variable Herbrand equalities, we thus build on the following two novel technical constructions:

- a method to uniquely factorize the kind of values possibly occurring at run-time (except finitely many) of a given program;
- a notion of approximate subsumption which is decidable and still guarantees that every occurring conjunction of equalities is effectively equivalent to a finite conjunction.

Subsequently, we sketch how not only all two-variable equalities, but *all* interprocedurally valid Herbrand equalities can be inferred, if only all right-hand sides in assignments each contain occurrences of at most one variable.

Finally we show that the complexity of inferring all valid two-variable Herbrand equalities in *initialization-restricted* programs is polynomial and that for *unrestricted* programs, at least verifying a given equality can be performed in

polynomial time. This is remarkable in so far as the terms encountered during the weakest precondition computation may be exponentially deep. In order to obtain a polynomial time analysis, we therefore follow the ideas sketched in [GT07] and provide *compressed* representations for the occurring terms which support all basic term operations in polynomial time. Subsequently, we show that our notion of approximate subsumption is decidable in polynomial time. Furthermore, for the multi-variable case, we show that verifying an invariant candidate is polynomial as well, given that the number of occurrences of variables in the postcondition is bounded.

2.1 Programs

For the purpose of this part of the thesis, we consider imperative programs which consist of a finite set P of procedures such as presented in Figure 2.1. Instead of operating on the syntax of programs, we prefer to represent each

```
0:  global x, y;
1:  main() {
2:      x := a;
3:      y := a;
4:      p();
5:  }

6:  p() {
7:      if (*) {
8:          x := f(x, x);
9:          p();
10:         y := f(y, y);
11:     }
12: }
```

Figure 2.1: A recursive program with one binary operator f

procedure by a (non-deterministic) control flow graph. Figure 2.2 shows, e.g., the control flow graphs for the given example program. Formally, the control flow graph for a procedure p consists of:

- A finite set N_p of program points where $s_p, r_p \in N_p$ represent the unique start and return point of the procedure p;
- A finite set E_p of edges (u, s, v) where $u, v \in N_p$ are program points and s denotes a basic statement.

A program which consists of the set P of procedures is then represented by the tuple $(\mathbb{N}, \mathbb{E})$ where $\mathbb{N} := \biguplus_{p \in P} N_p$ and $\mathbb{E} := \biguplus_{p \in P} E_p$. Furthermore, each program contains a dedicated procedure termed *main*.

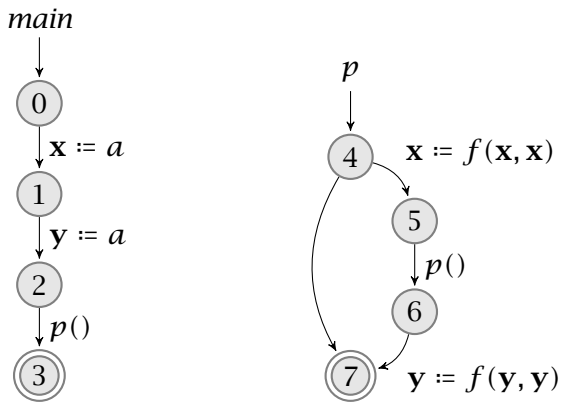


Figure 2.2: The corresponding CFGs for the program from Figure 2.1

For simplicity we assume that each program variable is initialized. Moreover, let us first proceed in the style of Sharir and Pnueli [SP81] and consider parameterless procedures which operate on global variables only. Later on, in Section 2.9, we lift this and also introduce procedures with local variables. In the following, $\mathbf{X}$ denotes the finite set of program variables. As *values*, we consider uninterpreted operator expressions only. Thus, values are constructed from atomic values by means of (uninterpreted) operator applications. Let Ω denote a finite signature containing a non-empty set of atomic values Ω_0 and sets Ω_k, $k > 0$, of constructors of rank k. Then $\mathcal{T}_\Omega$ denotes the set of all possible (ground) terms over Ω, and $\mathcal{T}_\Omega(\mathbf{X})$ the set of all possible terms over Ω and (possibly) occurrences of program variables from $\mathbf{X}$. In general, we will omit brackets around the argument of unary symbols. Thus, we may, e.g., write $h\mathbf{x}$ instead of $h(\mathbf{x})$.

As basic statements, we only consider assignments and procedure calls. An assignment $\mathbf{x} := ?$ non-deterministically assigns *any* value to the program variable $\mathbf{x}$, whereas an assignment $\mathbf{x} := t$ assigns the value constructed according to the right-hand side term $t \in \mathcal{T}_\Omega(\mathbf{X})$. A procedure call is of the form $p()$ for a procedure named p.

In this thesis, we only consider assignments whose right-hand sides contain occurrences of at most one program variable. The assignments occurring in the example program from Figure 2.2 fulfill this property. Note that this program does not fall into the class presented in [Pet10], since the right-hand sides of assignments contain more than one occurrence of a variable. In *general* programs with arbitrary assignments, the assignments with right-hand sides not conforming to the given restriction may, e.g., be abstracted by the non-deterministic assignment of *any* value.

2.2 Computing Weakest Preconditions

Our goal is to prove for a given assertion whether it is valid at a given program point or, better, to infer all invariants which are valid at that point. For that, we would like to calculate weakest preconditions of assertions, or, more generally, to determine for every program point the minimal assumptions to be met for the queried assertion to hold at the given program point. Since the program model makes use of non-deterministic branching, we may assume w.l.o.g. that every program point is *reachable*. In particular, this implies that no procedure is definitely non-terminating, i.e., that for every procedure p, there is at least one execution path from the start point of p reaching the end point of p.

Example 2.2.1. Consider the program from Figure 2.2. At program exit, the invariant $\mathbf{x} \doteq \mathbf{y}$ holds. In a proof of this fact by means of a weakest precondition computation, weakest preconditions must be provided for procedure p and all assertions $\mathbf{x} \doteq t_k$, $k \geq 0$, where $t_0 = \mathbf{y}$ and for $k > 0$, $t_k = f(t_{k-1}, t_{k-1})$. This set of postconditions is not only infinite, but also makes use of an ever increasing number of variable occurrences. Thus, an immediate encoding, e.g., into bounded degree polynomials as in [Pet10] is not obvious. ■

In order to summarize the effect of a procedure for multiple but similar postconditions, we tabulate the weakest preconditions for *generic* postconditions only. Generic postconditions are assertions which contain *template variables* which later may be instantiated differently in different contexts for arriving postconditions. This idea has been applied, e.g., for affine equalities [MS04; MS08; Fle+11], for polynomial equalities [MPS06; Pet10], or for Herbrand equalities with unary operators [GT07]. The generic postconditions which are of interest here, are of the forms

$$A\mathbf{x} \doteq C \quad \text{or} \quad A\mathbf{x} \doteq B\mathbf{y}$$

where $\mathbf{x}, \mathbf{y}$ are program variables, the *ground* template variable C is meant to receive a constant value, and the template variables A, B take *templates* as values, i.e., terms over the ranked alphabet Ω and having at least one occurrence of the (fresh) place holder variable $\bullet$. Computing weakest preconditions operates on assertions where an assertion is a possibly infinite conjunction of equalities. The equalities occurring during weakest precondition calculations are of the forms:

$$As \doteq C \quad \text{or} \quad As \doteq Bt$$

where s, t are terms possibly containing a program variable, i.e., $s, t \in \mathcal{T}_\Omega(\mathbf{X})$.

Consider a mapping σ which assigns *appropriate values* to the program variables from $\mathbf{X}$ as well as to the (non-ground or ground) template variables A, B, C. This means that σ assigns ground terms to the variables in $\mathbf{X} \cup \{C\}$ and templates to A, B. Such a mapping is called *variable assignment*. The variable assignment σ *satisfies* the equality $s \doteq t$ ($\sigma \models (s \doteq t)$ for short) iff $\sigma^*(s) = \sigma^*(t)$ where σ^* is the natural tree homomorphism corresponding to σ, which is the identity on all operators in Ω. The homomorphism σ^* maps, e.g., the application At of the template variable A to the term t into $\sigma(A)[\sigma^*(t)/\bullet]$, i.e., the *substitution* of the term $\sigma^*(t)$ into the occurrences of the dedicated variable $\bullet$ in the template $\sigma(A)$. Substitution into the dedicated variable $\bullet$ is an associative binary operation where the neutral element is the template consisting of $\bullet$ alone. In the following, we denote this operation by juxtaposition.

Consider, e.g., an assignment σ with $\sigma(A) = h(\bullet, \bullet)$, and $\sigma(B) = \bullet$, and $\sigma(\mathbf{x}) = a$. Then

$$\sigma^*(A\mathbf{x}) = h(\bullet, \bullet)\, a = h(a, a) = \bullet\, h(a, a) = \sigma^*(Bh(\mathbf{x}, a))$$

holds. Therefore, σ satisfies the equality $A\mathbf{x} \doteq Bh(\mathbf{x}, a)$. In the following, we will no longer distinguish between σ and σ^*.

The variable assignment σ satisfies the conjunction ϕ of equalities ($\sigma \models \phi$ for short), iff $\sigma \models e$ for all equalities $e \in \phi$.

In our application, it will be convenient not to consider arbitrary variable assignments, but only those which map program variables to *reasonable* values as shown in the following. For a subset $T \subseteq \mathcal{T}_\Omega$ of ground terms, we call a variable assignment σ a T-assignment, if σ maps program variables $\mathbf{x}$ to values $\sigma(\mathbf{x}) \in T$ only.

The conjunction ϕ then is called *T-satisfiable* if there is some T-assignment σ with $\sigma \models \phi$. Otherwise, it is *T-unsatisfiable*. Conjunctions ϕ, ϕ' are *T-equivalent* if for every T-assignment σ, $\sigma \models \phi$ iff $\sigma \models \phi'$. Obviously, an empty conjunction is satisfied by every variable assignment and therefore equal to $\top$ (true), while all T-unsatisfiable conjunctions are T-equivalent. As usual, these are denoted by $\bot$ (false). Finally, a conjunction ϕ' is *T-subsumed* by a conjunction ϕ, if ϕ is T-equivalent to $\phi \wedge \phi'$.

If the set T by which we have relativized the notions of satisfiability, equivalence and subsumption equals the full set $\mathcal{T}_\Omega$, we may also drop the prefixing with T. In particular, we have for any T that satisfiability, equivalence and subsumption imply T-satisfiability, T-equivalence and T-subsumption, while the reverse implication may not necessarily hold.

In the following, we recall the ingredients of weakest precondition computation for assignments as well as for procedure calls as provided, e.g. in [Hoa69] or [Cou90]. The weakest preconditions of ϕ w.r.t. assignments are given by:

$$\begin{aligned} [\![\mathbf{x} \coloneqq t]\!]^\top\ \phi &= \phi[t/\mathbf{x}] \\ [\![\mathbf{x} \coloneqq ?]\!]^\top\ \phi &= \forall\, \mathbf{x}.\, \phi \end{aligned}$$

Thus, the weakest precondition for an assignment $\mathbf{x} \coloneqq t$ is given by substitution of the term t into all occurrences of the variable $\mathbf{x}$ in the postconditions, while the weakest precondition for a non-deterministic assignment $\mathbf{x} \coloneqq ?$ of any value is given by universal quantification. For Herbrand equalities, universal quantification can be computed as follows. Recall that universal quantification commutes with conjunction. Therefore, it suffices to consider single equalities e. If $\mathbf{x}$ does not occur in e, then $\forall\, \mathbf{x}.\, e$ is equivalent to e. If $\mathbf{x}$ occurs only on one side of e, then $\forall\, \mathbf{x}.\, e = \bot$. Now assume that $\mathbf{x}$ occurs on both sides of e. If e is of the form $s\mathbf{x} \doteq t\mathbf{x}$ for templates s, t (no template variables), then either $s = t$ and hence e as well as $\forall\, \mathbf{x}.\, e$ is equivalent to $\top$, or $s \neq t$, in which case $\forall\, \mathbf{x}.\, e$ equals $\bot$. If e is of the form $As\mathbf{x} \doteq Bt\mathbf{x}$ for templates s, t, then $\forall\, \mathbf{x}.\, e$ is equivalent to $As \doteq Bt$.

Every transformation f which is specified for generic postconditions to conjunctions of preconditions, can be uniquely extended to a transformation f^* of *arbitrary* postconditions by

$$f^*(\bigwedge E) \quad = \quad \bigwedge_{e \in E} f^*(e)$$

where the transformation f^* for an arbitrary equality e is defined as follows:

$$f^*(s \doteq t) = \begin{cases} f(A\mathbf{x} \doteq B\mathbf{y})[s'/A, t'/B] & \text{if } s = s'\mathbf{x},\ t = t'\mathbf{y} \\ f(A\mathbf{x} \doteq C)[s'/A, t/C] & \text{if } s = s'\mathbf{x},\ t \text{ ground} \\ f(A\mathbf{x} \doteq C)[s/C, t'/A] & \text{if } t = t'\mathbf{x},\ s \text{ ground} \\ s \doteq t & \text{otherwise} \end{cases}$$

Subsequently, the extended function f^* is denoted by f as well. The procedure summaries are then characterized by the constraint system $\mathbf{S}$:

$$\begin{array}{lcll} [\![r_p]\!]^\top & \Longrightarrow & \mathtt{Id} & \text{for each procedure } p \\ [\![u]\!]^\top & \Longrightarrow & [\![s_p]\!]^\top \circ [\![v]\!]^\top & \text{for each } (u, p(), v) \in \mathbb{E} \\ [\![u]\!]^\top & \Longrightarrow & [\![s]\!]^\top \circ [\![v]\!]^\top & \text{for each } (u, s, v) \in \mathbb{E}, \\ & & & s \text{ assignment} \end{array}$$

where $\circ$ means the composition of the weakest precondition transformers and Id is the identity transformer. Thus, accumulation of weakest preconditions for a generic postcondition e at procedure exit r_p starts with e and then propagates its preconditions backward to the start point of p by applying the transformations corresponding to the traversed edges. Here, the subsumption relation $\Longrightarrow$ as defined for conjunction of equalities, has silently been raised to the function level. Thus, $f \Longrightarrow g$ if $f(e)$ subsumes $g(e)$ for all generic postconditions e.

W.r.t. the ordering $\sqsubseteq$ given by $\Longrightarrow$, the weakest precondition transformer of procedure p is then obtained as the value for the variable corresponding to the start point s_p in the *greatest* solution to the constraint system **S**.

The weakest precondition transformers for all program points are characterized by the greatest solution of the constraint system **R**:

$$
\begin{array}{lcll}
[s_{main}]^\top & \Longrightarrow & \mathsf{Id} & \\
[s_p]^\top & \Longrightarrow & [u]^\top & \text{for each } (u, p(), \_) \in \mathbb{E} \\
[v]^\top & \Longrightarrow & [u]^\top \circ [\![s_p]\!]^\top & \text{for each } (u, p(), v) \in \mathbb{E} \\
[v]^\top & \Longrightarrow & [u]^\top \circ [\![s]\!]^\top & \text{for each } (u, s, v) \in \mathbb{E}, \\
 & & & s \text{ assignment}
\end{array}
$$

The value for $[v]^\top$ for program point v is meant to transform every assertion at program point v, into the corresponding weakest precondition at the start point of the program. Note that the constraint system for characterizing these functions makes use of the weakest precondition transformers of procedures as characterized by the constraint system **S**.

Assume that we are somehow given the greatest solution of the constraint system **R** where $[v]^\top$ is the corresponding transformation for program point v. In order to determine all one- or two-variable equalities which are valid when reaching the program point v, we conceptually proceed as follows:

One-variable Equality. For a program variable $\mathbf{x}$, let ψ denote the *universal closure* of $[v]^\top(A\mathbf{x} \doteq C)$. If $\psi = \bot$, then program variable $\mathbf{x}$ does not receive a constant value at program point v. Otherwise ψ is equivalent to an equality $As \doteq C$ where s is ground, i.e., $\mathbf{x} \doteq s$ is an invariant at v.

Two-variable Equality. For distinct program variables $\mathbf{x}$ and $\mathbf{y}$, let ψ denote the universal closure of $[v]^\top(A\mathbf{x} \doteq B\mathbf{y})$. If $\psi = \bot$, then no equality between $\mathbf{x}$ and $\mathbf{y}$ holds. Otherwise, ψ equals a conjunction $\bigwedge_i As_i \doteq Bt_i$ of equalities where for each i either $s_i, t_i \in \mathcal{T}_\Omega$ are ground or $s_i, t_i \in$

$\mathcal{T}_\Omega(\bullet) \setminus \mathcal{T}_\Omega$ are templates. Then $r_1\mathbf{x} \doteq r_2\mathbf{y}$ is an invariant at v iff $r_1 s_i = r_2 t_i$ for all i, i.e., any assignment σ with $\sigma(A) = r_1, \sigma(B) = r_2$ satisfies the conjunction.

Here, the *universal closure* of a conjunction ϕ is given by $\forall\, \mathbf{x}_1 \ldots \forall\, \mathbf{x}_n.\phi$, if the set of program variables equals $\mathbf{X} = \{\, \mathbf{x}_1, \ldots, \mathbf{x}_n \,\}$.

Example 2.2.2. Consider the main procedure of the program in Section 2.1, as defined by the control flow graph in Figure 2.2. The weakest precondition transformer $[3]^\top$ for the endpoint 3 of the main program is given by:

$$[3]^\top = [\![\mathbf{x} := a]\!]^\top \circ [\![\mathbf{y} := a]\!]^\top \circ [\![4]\!]^\top$$

where 4 is the entry point of the procedure p. Assume that

$$[\![4]\!]^\top(A\mathbf{x} \doteq B\mathbf{y}) = (A\mathbf{x} \doteq B\mathbf{y}) \wedge (Af(\mathbf{x}, \mathbf{x}) \doteq Bf(\mathbf{y}, \mathbf{y}))$$

holds. For the program variables $\mathbf{x}, \mathbf{y}$, we therefore obtain:

$$\begin{aligned}[3]^\top(A\mathbf{x} \doteq B\mathbf{y}) &= (A\mathbf{x} \doteq B\mathbf{y})[a/\mathbf{y}][a/\mathbf{x}] \\ &\quad \wedge (Af(\mathbf{x}, \mathbf{x}) \doteq Bf(\mathbf{y}, \mathbf{y}))[a/\mathbf{y}][a/\mathbf{x}] \\ &= (Aa \doteq Ba) \wedge (Af(a, a) \doteq Bf(a, a))\end{aligned}$$

This assertion does not contain occurrences of the program variables $\mathbf{x}, \mathbf{y}$. Therefore, it is preserved by universal quantification over program variables. Since $A = B = \bullet$ is a solution, $\mathbf{x} \doteq \mathbf{y}$ holds whenever program point 3 is reached. ■

In order to turn these definitions into an effective analysis algorithm, several obstacles must be overcome. So, it is not clear how general subsumption, as required in our characterization of the weakest precondition transformers, can be decided in presence of template variables. We observe, however, that instead of general subsumption, it suffices to rely on T-subsumption only—for a well-chosen subset $T \subseteq \mathcal{T}_\Omega$. Note that the smaller the set T is, the coarser is the subsumption relation. In particular for $T = \emptyset$, all conjunctions are T-equivalent. Since every assertion expresses a property of reaching program states, it suffices for our application to choose T as a superset of all run-time values of program variables.

The following wish list collects properties which enable us to construct an effective interprocedural analysis of all two-variable Herbrand equalities:

T-**Compactness.** Every occurring conjunction ϕ is T-subsumed by a conjunction of a *finite* subset of equalities in ϕ.

Effectiveness of subsumption. T-subsumption for *finite* conjunctions can be effectively decided.

Solvability of ground equalities. The set of solutions of finite systems of equalities with template variables only, i.e., *without* occurrences of program variables can be computed.

By the first assumption, a standard fixed point iteration for the constraint systems **S** and **R** will terminate after finitely many iterations (up to T-equivalence). By the second assumption, termination can effectively be detected, while the third assumption guarantees that for every program point and every program variable (or pair of program variables) the set of all valid invariants can be extracted out of the greatest solution of **R**. In total, we arrive at an effective algorithm for inferring all valid two-variable equalities.

The assumption on decidability of T-subsumption can be further relaxed. Instead, we provide an *approximate* notion of T-subsumption which is decidable. Our approximate T-subsumption implies T-subsumption. Moreover, it is still strong enough to guarantee that every occurring conjunction of equalities is approximately T-subsumed by a finite subset of the equalities. Notions for approximate T-subsumption are introduced in Sections 2.5 and 2.6.

For programs which operate on global as well as local variables, an extension of the program model and weakest precondition calculus is given in Section 2.9. There we introduce a program model which is general enough in order to model usual concepts of local variables together with call-by-value parameter parsing and returning of results in dedicated global variables. Furthermore, we extend the weakest precondition calculus in order to deal with generic postconditions which contain local program variables.

In the upcoming section, we recall basic properties of the set of terms, possibly containing the variable $\bullet$. These properties will allow us to deal with conjunctions of equalities where template variables are applied to ground terms only, i.e., the case of ground equalities.

2.3 Factorization of Terms

Let $\mathcal{T}_\Omega(\bullet)$ denote the set of terms constructed from the symbols in Ω, possibly together with the dedicated variable $\bullet$. In [Eng80], Engelfriet presents the following cancellation and factorization properties for terms in $\mathcal{T}_\Omega(\bullet)$:

Bottom Cancellation:
Assume that $t_1 \neq t_1'$. Then $s_1 t_1 = s_2 t_1$ and $s_1 t_1' = s_2 t_1'$ implies $s_1 = s_2$.

Top Cancellation:
Assume $\bullet$ occurs in s. Then $st_1 = st_2$ implies $t_1 = t_2$.

Factorization:
Assume $t_i \neq t_i'$ for $i = 1, 2$. Then $s_1 t_1 = s_2 t_2$ and $s_1 t_1' = s_2 t_2'$ implies that $s_1 r_1 = s_2 r_2$ for some r_1, r_2 each containing $\bullet$ where at least one of the r_i equals $\bullet$. In that case s_1 is a prefix of s_2 or vice versa. Therefore, by top cancellation we furthermore have that both $r_2 t_1 = r_1 t_2$ and $r_2 t_1' = r_1 t_2'$. We proof the latter as follows. Assume that $r_1 = \bullet$, i.e., $s_1 = s_2 r_2$, we then have $s_2 r_2 t_1 = s_2 r_1 t_2$ from which follows by top cancellation that $r_2 t_1 = r_1 t_2$. A similar argument holds for $r_2 t_1' = r_1 t_2'$ and for the case where $r_2 = \bullet$.

Using these cancellation properties, we obtain a complete method for equalities *without* occurrences of program variables.

For one-variable equalities alone, we have the following results concerning subsumption and compactness:

Theorem 2.3.1.

1. *A single equality $As \doteq C$ for some ground term s has exactly one solution where $A = \bullet$.*

2. *Consider the conjunction $As_1 \doteq C \wedge As_2 \doteq C$ for terms $s_1 \neq s_2$ containing the same variable $\mathbf{x}$. If the conjunction is satisfiable, then the value of $\mathbf{x}$ is uniquely determined.*

Proof. We prove the first assertion. If $A = \bullet$, then $As \doteq C$ is equivalent to $s \doteq C$ from which follows that s is the only solution for C.

We prove the second assertion. The conjunction $As_1 \doteq C \wedge As_2 \doteq C$ is equivalent to the conjunction $As_1 \doteq C \wedge s_1 \doteq s_2$. The most general unifier of s_1, s_2 maps $\mathbf{x}$ to a ground subterm of s_1, s_2 if the conjunction is satisfiable. □

As a consequence, we obtain:

Corollary 2.3.2. *Consider finite conjunctions of equalities of the form $As \doteq C$.*

1. *Subsumption for these is decidable.*

2. *Every satisfiable conjunction is equivalent to a conjunction of at most $n+1$ equalities where n is the number of program variables, when $A = \bullet$.*

Proof. Consider a satisfiable conjunction ϕ containing two equalities $As_1\mathbf{x} \doteq C$ and $As_2\mathbf{x} \doteq C$ for some program variable $\mathbf{x}$. Then from Theorem 2.3.1 the ground solution for variable $\mathbf{x}$ is uniquely determined and therefore also the unique solution for C. From the solution of C follows for any other equality $At\mathbf{y} \doteq C$ of conjunction ϕ the unique solution for each program variable $\mathbf{y}$. Hence, for any two equalities $At_1\mathbf{y} \doteq C$ and $At_2\mathbf{y} \doteq C$ of conjunction ϕ we have that $At_1\mathbf{y} \doteq C$ subsumes $At_2\mathbf{y} \doteq C$. Therefore, the conjunction ϕ is equivalent to a conjunction containing at most one equality for each program variable and only for one program variable a second equality is required. □

Since the weakest precondition of a generic one-variable equality consists of equalities of the form $As \doteq C$ only, Corollary 2.3.2 suffices to infer all interprocedurally valid one-variable equalities. In the following, we therefore concentrate on the two-variable case where the weakest precondition consists of conjunctions of equalities of the form $As \doteq Bt$. First, we observe:

Theorem 2.3.3.

1. *A single equality $As \doteq Bt$ for ground terms s, t has only finitely many solutions where at least one of the templates for A or B equals $\bullet$.*

2. *Consider the conjunction $As_1 \doteq Bt_1 \wedge As_2 \doteq Bt_2$ for ground terms $s_1 \neq s_2$ and $t_1 \neq t_2$. Then it has either no solution or there exists a unique solution $A = r_1, B = r_2$, where at least one of the templates r_1, r_2 equals $\bullet$. In the latter case the conjunction is equivalent to $Ar_1 \doteq Br_2$.*

3. *Consider the finite conjunction $\bigwedge_{i=1}^{k} As_i \doteq Bt_i$ for ground terms s_i, t_i. Then the set of all solutions can be effectively computed, where at least one of the templates for A or B equals $\bullet$.*

Proof. For a proof of the first statement, w.l.o.g. assume that s is at least as large as t. Then for size reasons, $r_1 = \bullet$ if r_1 is the template for A. This means that $s = r_2 t$ must hold if r_2 is the template for B. If t is not a subterm of s, there is no solution at all. Otherwise, i.e., if s contains occurrences of t, then every solution r_2 is obtained from s by replacing a non-empty set of occurrences of t with $\bullet$.

Now consider the second statement. If the pair of equalities is satisfiable then by factorization, there are templates r_1, r_2 of which at least one equals

$\bullet$ such that $Ar_1 \doteq Br_2$ holds. Since at the same time $r_2 s_i \doteq r_1 t_i$ holds, the equality $Ar_1 \doteq Br_2$ is equivalent to the conjunction. Moreover, there is exactly one solution $A = r_1', B = r_2'$ where at least one of the templates r_i' equals $\bullet$, namely, $r_1' = r_2$, $r_2' = r_1$.

Finally, consider the third statement. If $k = 1$, the assertion follows from statement 1. Therefore now let $k > 1$. First assume that for some i, j, $s_i \neq s_j$ and $t_i \neq t_j$. Then by statement 2, the conjunction is unsatisfiable or there is exactly one pair r_1, r_2 of templates one of which equals $\bullet$, such that $A = r_1, B = r_2$ is a solution of the conjunction $As_i \doteq Bt_i \wedge As_j \doteq Bt_j$. If in the latter case, $r_1 s_l \doteq r_2 t_l$ for all l, we have obtained a single solution. Otherwise, the conjunction is unsatisfiable. Now assume that no such i, j exists. Then either the conjunction is unsatisfiable or all equalities are syntactically equal. □

Example 2.3.1. Consider the two equalities:

$$Af(a, gb, gb) \doteq Bgb \qquad Af(a, gc, gb) \doteq Bgc$$

Then $A = \bullet$ and $B = f(a, \bullet, gb)$ is the only solution for A, B where at least one of the templates equals $\bullet$. ■

Applying the arguments which we used to prove Theorem 2.3.3, we obtain:

Corollary 2.3.4. *Consider a conjunction $\bigwedge_{i=1}^{n} As_i \doteq Bt_i$ with ground terms s_i, t_i.*

1. *If it is satisfiable, it is equivalent to the conjunction of at most two conjuncts.*
2. *If it is unsatisfiable, there are at most three conjuncts whose conjunction is unsatisfiable.* □

By Theorem 2.3.3, the assumption **solvability of ground equalities** from Section 2.2 is met. Thus, it remains to solve the constraint systems **S** and **R**, i.e., to construct an approximate T-subsumption relation which is both effective and guarantees that every conjunction is approximately T-subsumed by the conjunction of a finite subset of equalities. In order to construct such a relation, we require stronger insights into the structure of templates and their compositions. Let C_Ω denote the subset of all terms in $\mathcal{T}_\Omega(\bullet)$ which contain at least one occurrence of $\bullet$, i.e., $C_\Omega = \mathcal{T}_\Omega(\bullet) \setminus \mathcal{T}_\Omega$. The terms in C_Ω have also been called *templates*. The set C_Ω, equipped with substitution, is a *free monoid* with neutral element $\bullet$. This monoid consists of finite products of the irreducible elements in C_Ω. As usual, we call an element t *irreducible*

if t cannot be non-trivially decomposed into a product, i.e., $t = uv$ implies that $t = u$ with $v = \bullet$ or $t = v$ with $u = \bullet$. Note that there are *infinitely* many irreducible elements in C_Ω whenever Ω contains constructors of rank exceeding 1.

While templates can be uniquely factored, this is no longer the case for ground terms, i.e., terms without variable occurrences.

Example 2.3.2. Consider the ground term $t = h(f(h(1), h(1)))$, together with the templates $s_1 = h(f(\bullet, h(1)))$, $s_2 = h(f(h(1), \bullet))$ and $s_3 = h(f(\bullet, \bullet))$. All these three templates are distinct. Still,

$$t = s_1\ h(\bullet)\ 1 = s_2\ h(\bullet)\ 1 = s_3\ h(\bullet)\ 1 \qquad \blacksquare$$

Thus, unique factorization of arbitrary ground terms cannot be hoped for. Still, in the following we will observe that unique factorization for ground terms can be obtained—at least up to a fixed finite set of ground terms.

Let G denote a finite set of ground terms which is closed by subterms. Let M_G denote the sub-monoid of all templates in C_Ω whose ground subterms all are contained in G. Then we have:

Theorem 2.3.5. *Assume that $S \subseteq \mathcal{T}_\Omega$ which is closed by subterms. If $G \subseteq S$, then every ground term $t \in \mathcal{T}_\Omega \setminus S$, can be uniquely factored into $t = mx$ such that*

1. *$m \in M_G$ and $x \notin S$;*
2. *x is minimal with property 1, i.e., there exists no $x' \in \mathcal{T}_\Omega \setminus S$ such that $x = sx'$ for some $s \in M_G \setminus \{\bullet\}$.*

Proof. Since $M_G \subseteq C_\Omega$, every term in M_G is uniquely factorizable.

Let $t = m_1x_1 = m_2x_2$ with $m_i \in M_G$ and $x_i \in \mathcal{T}_\Omega \setminus S$ are minimal according to property 2 for $i = 1, 2$. Then either $m_1 = m_2m'$ or $m_2 = m_1m'$ for some $m' \in M_G$ holds. Otherwise, we have a contradiction to the assumption that $m_1x_1 = m_2x_2$ holds. Consider the case where $m_1 \neq m_2$, i.e., $m' \neq \bullet$. If $m_1 = m_2m'$, then we conclude that $m'x_1 = x_2$ holds. This means, that x_2 is not minimal according to property 2 which is a contradiction to our assumption. A similar argument holds for $m_2 = m_1m'$. Now consider the case where $m_1 = m_2$, then also $x_1 = x_2$ from which the assertion of the theorem follows. □

Example 2.3.3. Consider the term

$$t = f(h(f(2, h(1))), h(f(2, h(1))))$$

and assume that the set G of forbidden ground subterms is given by $G = \{h(1), 1\}$ and $S = G$. Then t can be decomposed into:

$$f(\bullet, \bullet)\ h(\bullet)\ f(\bullet, h(1))\ 2$$

If on the other hand, $S = G = \{2\}$, we obtain the decomposition:

$$f(\bullet, \bullet)\ h(\bullet)\ f(2, \bullet)\ h(\bullet)\ 1$$

If finally, S and G are empty, the term x of Theorem 2.3.5 is the minimal subterm such that the occurrences of x contains all ground leaves of t. This means that $x = f(2, h(1))$, and we obtain the decomposition:

$$f(\bullet, \bullet)\ h(\bullet)\ f(2, h(1))$$ ■

The unique decomposition of the ground term t claimed by Theorem 2.3.5, is constructed as follows. Let X denote the set of minimal subterms x' of t such that $x' \notin G$. Then we construct the least subterm $x \notin S$ of t such that all occurrences of subterms $x' \in X$ in t are contained in some occurrence of x. This subterm is uniquely determined. Then define m as the term obtained from t by replacing all occurrences of x with $\bullet$. This term m is also uniquely determined with $t = mx$. Moreover by construction, all ground subterms of m are contained in G.

Example 2.3.4. Consider the program from Example 2.2.1. In this program, no non-ground right-hand side contains ground subterms. Accordingly, the set G is empty. Since the only ground right-hand side equals the atom a, the decomposition Theorem 2.3.5 allows to uniquely decompose all run-time values of this program into right-hand sides of assignments. ■

Theorem 2.3.5 allows to extend the monoidal techniques of Gulwani et al. [GT07] for unary operators to programs where all run-time values can be uniquely factorized into right-hand sides. This extension is given in Section 2.5. The general case where unique factorization of all run-time values can no longer be guaranteed, subsequently is presented in Section 2.6. For completeness reasons, we also present simplified versions of the algorithms for monoidal equalities from [GT07] in the next section.

2.4 Equalities over a Free Monoid

Consider a free monoid M_Σ with set of generators Σ. As usual, the neutral element of M_Σ is denoted by ϵ. Let F_Σ be the corresponding free group. F_Σ

can be considered as the free monoid generated from $\Sigma \cup \Sigma^-$ (where $\Sigma^- = \{ a^- \mid a \in \Sigma \}$ is the set of formal inverses of elements in Σ with $\Sigma \cap \Sigma^- = \emptyset$) modulo exhaustive application of the cancellation rules $a \cdot a^- = a^- \cdot a = \epsilon$ for all $a \in \Sigma$. In particular, the neutral element of F_Σ is given by ϵ, and the inverse g^{-1} of an element $g = a_1 \cdots a_k$, $a_i \in \Sigma \cup \Sigma^-$, is given by $g^{-1} = a_k^{-1} \cdots a_1^{-1}$ where $x^{-1} = x^-$ and $(x^-)^{-1} = x$ for $x \in \Sigma$.

For every $w \in M_{\Sigma \cup \Sigma^-}$, the *balance* $|w|$ is the difference between the number of occurrences of positive and negative letters in w, respectively. Formally, the balance is inductively defined by

$$\begin{array}{lcll} |\epsilon| & = & 0 & \\ |aw| & = & |w| + 1 & \text{if } a \in \Sigma \\ |aw| & = & |w| - 1 & \text{if } a \in \Sigma^- \end{array}$$

Thus, $|aba^-b^-c| = 1$ and $|a^-b| = 0$. Note that the balance stays invariant under application of the cancellation rules. Also, $|uv| = |u| + |v|$ and $|u^{-1}| = -|u|$. Accordingly, the balance $|\cdot| \colon F_\Sigma \to \mathbb{Z}$ is a group homomorphism. Furthermore, we call w *non-negative* if $|w'| \geq 0$ for all prefixes w' of w. This property is also preserved by cancellation and concatenation but not by inverses. Instead, we have:

Lemma 2.4.1. *If both $u, v \in M_{\Sigma \cup \Sigma^-}$ are non-negative, and $|u| \geq |v|$ then also uv^{-1} is non-negative.*

Proof. Consider a prefix x of uv^{-1}. If x is a prefix of u, $|x| \geq 0$ since u is non-negative. Otherwise, $x = uv'^{-1}$ for some suffix v' of v. Then $|v'| \leq |v|$, since v is non-negative. Therefore, $|uv'^{-1}| = |u| - |v'| \geq |u| - |v| \geq 0$. □

We consider equalities of the form:

$$AuA^{-1} = Bu'B^{-1} \tag{2.1}$$

where A, B are variables which take values in M_Σ, and $u, u' \in M_{\Sigma \cup \Sigma^-}$ are maximally canceled. If the equality is satisfiable, then necessarily $|u| = |u'|$ holds. Assume from now on that u, u' are maximally canceled, and $|u| = |u'|$. Furthermore, we assume that u, u' are both non-negative. We then have:

Lemma 2.4.2. *If $|u| = |u'| = 0$, then the equality (2.1) is either trivial, is equivalent to an equality $As = B$ or an equality $A = Bs$ for some $s \in M_\Sigma$ or is contradictory.*

Proof. Assume $u = \epsilon$. Then by cancellation $AA^{-1} \doteq Bu'B^{-1}$ is equivalent to $\epsilon \doteq Bu'B^{-1}$. By right multiplication of B and again of cancellation we obtain $B = Bu'$. Thus either $u' = \epsilon$ and the equality is trivial, or $u' \neq \epsilon$ and the equality is contradictory.

Therefore, assume that $u \neq \epsilon \neq u'$. Then u and u' must be of the form $u = xyz^{-1}$, $u' = x'y'z'^{-1}$ for maximal $x, x', z, z' \in M_\Sigma$, i.e., y, y' each are either equal to ϵ or of the form $a^- wb$ for some $a, b \in \Sigma$. Then all x, x', z, z' are different from ϵ. Then equality (2.1) is equivalent to:

$$Ax = Bx' \wedge y = y' \wedge Az = Bz'$$

By bottom cancellation, these three equalities either are equivalent to one fixed relation between $As = B$ or $A = Bs$ for some $s \in M_\Sigma$, or to a contradiction. □

Example 2.4.1. Consider the equality

$$Affg^{-1}f^{-1}A^{-1} \doteq Bfg^{-1}B^{-1}$$

which is, according to Lemma 2.4.2, equivalent to

$$Aff \doteq Bf \wedge \epsilon \doteq \epsilon \wedge Afg \doteq Bg$$

By bottom cancellation, we conclude that the conjunction is equivalent to a solved equality $Af \doteq B$. ■

Now assume that there is another equality:

$$AvA^{-1} = Bv'B^{-1} \tag{2.2}$$

with non-negative v, v' where $|v| = |v'|$.

Theorem 2.4.3. *The two equalities* (2.1) *and* (2.2) *are effectively equivalent either to one solved equality, or to a single equality of the form* (2.1) *or are contradictory.*

Proof. We perform an induction on the sum of balances $|u| + |v|$. W.l.o.g. assume that $|u| \geq |v|$. If $|v| = 0$, then the assertion follows from Lemma 2.4.2. Therefore, assume that $|v| > 0$, and $r \geq 1$ is the maximal number such that $|v^r| = r \cdot |v| \leq |u|$. Then we construct the elements uv^{-r} and $u'v'^{-r}$, which are both non-negative by Lemma 2.4.1. Let w, w' be obtained from uv^{-r} and $u'v'^{-r}$ by exhaustively applying the cancellation rules. By construction, these are non-negative as well. Then we consider the equality:

$$AwA^{-1} = Bw'B^{-1} \tag{2.3}$$

which is implied by the two equalities (2.1) and (2.2).

If $w = \epsilon$, then either $w' = \epsilon$ holds and the equality (2.3) is trivial, or $w' \neq \epsilon$ and equality (2.3) is contradictory. In the first case, the equality (2.2) is implied by equality (2.1), while in the second case the two given equalities (2.1) and (2.2) are contradictory. The same argument applies when $w' = \epsilon$ with the roles of A, B exchanged. Therefore now assume that $w \neq \epsilon \neq w'$. Otherwise, the pair of equalities (2.1) and (2.2) is equivalent to the pair of equalities (2.2) and (2.3), where the sum of balances $|w| + |v| \leq |w| + r \cdot |v| = |u| < |u| + |v|$ has decreased. For these, the claim follows by inductive hypothesis. □

In [GT07] a similar argument is presented. The argument there together with the resulting algorithm has been significantly simplified by introducing the extra notion of *non-negativity*.

2.5 Initialization-restricted Programs

In the subsequent let R be the set of ground right-hand sides of assignments, and G be the set of ground subterms of non-ground right-hand sides of assignments of our program. Then generally, each value x possibly constructed at run-time by the program is of the form $x = mr$ where $m \in M_G$ and $r \in R$ or $r \in \mathbf{X}$. The latter case with $r \in \mathbf{X}$ indicates an unknown value where a variable was not initialized.

Example 2.5.1. Consider the following program consisting of three assignments:

$$\mathbf{x} := \mathbf{y};\ \mathbf{x} := f(\mathbf{x}, \mathbf{x});\ \mathbf{y} := f(\mathbf{y}, \mathbf{y})$$

Although variable $\mathbf{y}$ is not initialized before its first read and therefore an unknown value has to be assumed for it, the invariant $\mathbf{x} = \mathbf{y}$ holds at program exit. ■

We thus have:

Lemma 2.5.1. *Each program variable in* $\mathbf{X}$ *ranges over the set* $M_G R$ *or* $M_G \mathbf{X}$. □

This means that for preconditions ϕ possibly occurring in a weakest precondition computation for a program invariant, we are only interested in variable assignments σ which map each program variable $\mathbf{x}$ to a possible run-time value for $\mathbf{x}$, i.e., to a value from the set $M_G R$. In the subsequent let

$$T := M_G R \quad \text{and} \quad T' := M_G \mathbf{X}$$

then during the weakest precondition computation template variables are applied to ground terms in T and non-ground terms in T' only. Henceforth, we therefore no longer consider general satisfiability, equivalence and subsumption, but only T-satisfiability, T-equivalence and T-subsumption. This restriction is crucial for the generalization of the monoidal techniques from [GT07]. In the following, we first consider the sub-class of programs P where the set R of ground right-hand sides of P satisfies the two properties:

1. $R \cap G = \emptyset$.
2. The elements in R are mutually incomparable ground terms, i.e., for $r_1, r_2 \in R$, r_1 is a subterm of r_2 iff $r_1 = r_2$.

The program P then is called *initialization-restricted* (IR for short).

Example 2.5.2. Assume that the non-ground right-hand sides of assignments of a program are $f(\mathbf{x}, h(1))$ and $f(2, h(\mathbf{y}))$. Then the set G is given by $G = \{ 1, h(1), 2 \}$. A suitable set R of ground right-hand sides might be, e.g., $R = \{ 0, a \}$. ■

Our condition here is not as restrictive as it might seem. Programs where each variable is initialized by a non-deterministic assignment, are all IR. The same holds true for programs where all non-ground right-hand sides of assignments do not contain ground terms, and variables are initialized with atoms only. The latter property is met by our Example 2.2.1. By suitably massaging variable initializations, it also comprises all programs using monadic operators only (as in [GT07]).

We distinguish between two-variable equalities of the following formats:

$[F_{\mathbf{x},\mathbf{x}}]$	$As\mathbf{x}$	$\doteq$	$Bt\mathbf{x}$	where $s, t \in M_G$
$[F_{\mathbf{x},\mathbf{y}}]$	$As\mathbf{x}$	$\doteq$	$Bt\mathbf{y}$	where $s, t \in M_G$
$[F_{\cdot,\mathbf{x}}]$	As	$\doteq$	$Bt\mathbf{x}$	where $s \in T$ and $t \in M_G$
$[F_{\mathbf{x},\cdot}]$	$At\mathbf{x}$	$\doteq$	Bs	where $s \in T$ and $t \in M_G$

For each format separately, we observe:

Theorem 2.5.2.

***T*-subsumption.** *For finite sets E, E' of two-variable equalities of the same format it is decidable whether $\bigwedge E$ T-subsumes $\bigwedge E'$ or not.*

***T*-compactness.** *Every T-satisfiable conjunction of a set E of two-variable equalities of the same format is T-subsumed by a conjunction of a subset of at most three equalities in E.*

Proof. In order to prove the theorem we show that every T-satisfiable conjunction of equalities of the same format is effectively T-subsumed by a conjunction of at most three equalities. Furthermore, the proof indicates that, given three equalities, it can be effectively decided whether or not a fourth equality is T-subsumed or not. We consider one case of the assertion of the theorem after the other.

Same variable on both sides. Consider the two distinct equalities

$$As_1\mathbf{x} \doteq Bt_1\mathbf{x} \qquad As_2\mathbf{x} \doteq Bt_2\mathbf{x}$$

where $s_i, t_i \in M_G$, and assume that the conjunction of them is T-satisfiable. We claim that then $s_1\mathbf{x} \neq s_2\mathbf{x}$ and $t_1\mathbf{x} \neq t_2\mathbf{x}$—otherwise $\mathbf{x}$ would be a subterm of the terms s_i, t_i from M_G which is a contradiction to our assumption. If $s_1 = s_2$, then from $As_1\mathbf{x} \doteq Bt_1\mathbf{x} \wedge As_1\mathbf{x} \doteq Bt_2\mathbf{x}$ follows that $t_1\mathbf{x} = t_2\mathbf{x}$ must hold which means that $\mathbf{x}$ is a subterm of t_i which is again a contradiction to our assumption. A similar argument applies for the case if $t_1 = t_2$. Therefore, $s_1 \neq s_2$ and $t_1 \neq t_2$ must hold. Then for a contradiction, assume that $s_1\mathbf{x} \doteq s_2\mathbf{x}$. Since $s_1 \neq s_2$, their unifier must map $\mathbf{x}$ to a ground term of s_1 and s_2. These ground terms are all contained in G, whereas we only consider values for $\mathbf{x}$ in M_GR, which is disjoint from G. A similar argument also shows that $t_1\mathbf{x} \neq t_2\mathbf{x}$ holds. Thus by factorization, $Ar_1 \doteq Br_2$ must hold for some $r_1, r_2 \in M_G$ of which at least one equals $\bullet$. Due to unique factorization, we then may cancel $\mathbf{x}$ on both sides, resulting in the equalities $As_1 \doteq Bt_1$ and $As_2 \doteq Bt_2$. These can be simplified to one equality $Ar_1 \doteq Br_2$ for some $r_1, r_2 \in M_G$ where $r_i = \bullet$ for at least one i. Hence, the second equality is T-subsumed by the first one.

Different variables on both sides. Consider the three distinct equalities

$$As_1\mathbf{x} \doteq Bt_1\mathbf{y} \qquad As_2\mathbf{x} \doteq Bt_2\mathbf{y} \qquad As_3\mathbf{x} \doteq Bt_3\mathbf{y}$$

for distinct program variables $\mathbf{x}, \mathbf{y}$ where $s_i, t_i \in M_G$, and assume that the conjunction of them is T-satisfiable. As before, we argue that $s_i\mathbf{x} \neq s_j\mathbf{x}$, $t_i\mathbf{y} \neq t_j\mathbf{y}$ for all $i \neq j$ must hold. Then by factorization, A is a prefix of B or vice versa. But then, due to unique factorization, also As_1 is a prefix of Bt_1 or vice versa. This means that there are $u, v \in M_G$ of which one equals $\bullet$ such that $As_1u \doteq Bt_1v$, which (by top cancellation) implies that $v\mathbf{x} = u\mathbf{y}$ holds. From that, we conclude that $As_iu \doteq Bt_iv$ for all i. Assume again w.l.o.g. that the balance of s_1 is less or equal to the balances of s_2 and s_3. We then proceed as in the last case to obtain the T-equivalent three equalities:

$$As_1u \doteq Bt_1v \qquad As_2s_1^{-1}A^{-1} \doteq Bt_2t_1^{-1}B^{-1} \qquad As_3s_1^{-1}A^{-1} \doteq Bt_3t_1^{-1}B^{-1}$$

where $s_2 s_1^{-1}$, $t_2 t_1^{-1}$, $s_3 s_1^{-1}$, $t_3 t_1^{-1}$ are non-negative. According to Theorem 2.4.3, the latter two equalities again are T-equivalent to an equality $Ar_1 \doteq Br_2$ for templates r_1, r_2 of which at least one equals $\bullet$, or are T-equivalent to each other, and the assertion follows.

One-sided single variable. Consider the three distinct equalities

$$As_1 \doteq Bt_1\mathbf{x} \qquad As_2 \doteq Bt_2\mathbf{x} \qquad As_3 \doteq Bt_3\mathbf{x}$$

where $s_i \in M_G R$ and $t_i \in M_G$, and assume that the conjunction of them is T-satisfiable. Again, we argue that all s_i must be distinct as well as all $t_i\mathbf{x}$. Then again by factorization, $Ar_1 \doteq Br_2$ for some templates r_1, r_2 of which at least one equals $\bullet$. By unique factorization, $s_1 = s_1' r$ for some $s_1' \in M_G$ and $r \in R$. Therefore, again by unique factorization, the value for $\mathbf{x}$ also must terminate in the term r, i.e., is of the form $\mathbf{x} = x' r$ for some $x' \in M_G$. Accordingly, also s_2, s_3 can be factored as $s_i = s_i' r$ for suitable $s_i' \in M_G$. Canceling out the ground terms r, we obtain the monoid equalities:

$$As_1' \doteq Bt_1 x' \qquad As_2' \doteq Bt_2 x' \qquad As_3' \doteq Bt_3 x'$$

Assume w.l.o.g., that the balance of s_1 is less or equal to the balances of s_2 and s_3. Then the conjunction of the three equalities is T-equivalent to:

$$As_1' \doteq Bt_1 x' \qquad As_2' s_1'^{-1} A^{-1} \doteq Bt_2 t_1^{-1} B^{-1} \qquad As_3' s_1'^{-1} A^{-1} \doteq Bt_3 t_1^{-1} B^{-1}$$

where $s_2' s_1'^{-1}, t_2 t_1^{-1}, s_3' s_1'^{-1}, t_3 t_1^{-1}$ all are non-negative. According to Theorem 2.4.3, the two last equalities are either T-equivalent to each other, which means that the initial conjunction is T-equivalent to the conjunction of the two equalities

$$As_1 \doteq Bt_1\mathbf{x} \qquad As_2 \doteq Bt_2\mathbf{x}$$

and the assertion follows. Otherwise, they are T-equivalent to an equality $Ar_1 \doteq Br_2$ for templates r_1, r_2 of which at least one equals $\bullet$. A fourth equality is then either T-subsumed or falsifies the conjunction of equalities. A similar argument applies to equalities of the form $At_i\mathbf{x} \doteq Bs_i$. This completes the proof. □

Since T-subsumption is decidable, at least for equalities of the same format, we define an approximate T-subsumption for conjunctions of equalities $\bigwedge E$ and $\bigwedge E'$ as follows. Let E_F and E_F' denote the subsets of equalities of the same format F in E and E', respectively. Then $\bigwedge E$ approximate T-subsumes $\bigwedge E'$ iff $\bigwedge E_F$ T-subsumes $\bigwedge E_F'$ for all formats F. Hence, by Theorem 2.5.2, we obtain:

Corollary 2.5.3. *Assume that n is the number of program variables.*

Approximate T-subsumption. *For finite sets E, E' of two-variable equalities, it is decidable whether $\bigwedge E$ approximately T-subsumes $\bigwedge E'$ or not.*

Approximate T-compactness. *Every T-satisfiable conjunction of a set E of two-variable equalities is approximately T-subsumed by a conjunction of a subset of at most $\mathcal{O}(n^2)$ equalities in E.* □

Overall, we therefore conclude for IR programs:

Theorem 2.5.4. *For every program point u of an IR program, the set of all two-variable Herbrand equalities can be determined that are valid when reaching program point u.*

Proof. By Corollary 2.5.3, the greatest solutions of the constraint systems **S** and **R** can be effectively computed. Let $[u]^\top$, u program point, denote the greatest solution of the system **R**. Then the set of valid equalities $s\mathbf{x} \doteq t\mathbf{y}$ between program variables $\mathbf{x}$, $\mathbf{y}$ is given by the set of solutions to a system of ground equalities which are obtained by universal quantification over all program variables of the conjunction of equalities $[u]^\top(A\mathbf{x} \doteq B\mathbf{y})$. By Theorem 2.3.3, a representation of the set of solutions for the template variables A, B in this conjunction can be explicitly computed.

Likewise, the set of valid equalities $\mathbf{x} \doteq t$ for program variable $\mathbf{x}$ and ground term t can be extracted from the universal quantification over all program variables of the conjunction of equalities $[u]^\top(A\mathbf{x} \doteq C)$. The resulting conjunction may either equal $\bot$ (meaning no constant value for $\mathbf{x}$) or contain only the variable C. Consequently, the possible constant value for $\mathbf{x}$ and program point u can also be effectively computed. This completes the proof. □

Example 2.5.3. According to our constructions presented in Section 2.2 and Theorem 2.3.3, the set of all interprocedurally valid assertions can be obtained from the greatest solutions to the constraint systems **S** and **R**. Consider, e.g., the constraint system **R** for the recursive procedure p from Section 2.1, as defined by the control flow graph of Figure 2.2. If round-robin iteration is applied to calculate the transformers $[\![u]\!]^\top$ for the program points $u = 4, 5, 6, 7$, we obtain for the generic postcondition $A\mathbf{x} \doteq B\mathbf{y}$ the result depicted by Table 2.1 where in the ith column, we only displayed preconditions which have additionally been attained in the ith iteration for the program points 7, 6, 5 and 4, respectively. For convenience, we displayed the terms in equalities according

Table 2.1: Round-robin iteration for procedure p from Figure 2.2

	1	2	3
7	$A\mathbf{x} \doteq B\mathbf{y}$		
6	$A\mathbf{x} \doteq Bf(\bullet,\bullet)\mathbf{y}$		
5	$\top$	$A\mathbf{x} \doteq Bf(\bullet,\bullet)\mathbf{y}$	$Af(\bullet,\bullet)\mathbf{x} \doteq Bf(\bullet,\bullet)f(\bullet,\bullet)\mathbf{y}$
4	$A\mathbf{x} \doteq B\mathbf{y}$	$Af(\bullet,\bullet)\mathbf{x} \doteq Bf(\bullet,\bullet)\mathbf{y}$	$Af(\bullet,\bullet)f(\bullet,\bullet)\mathbf{x} \doteq Bf(\bullet,\bullet)f(\bullet,\bullet)\mathbf{y}$

to their unique factorizations. For program point 4, the two equalities after the second iteration, imply:

$$Af(\bullet,\bullet)A^{-1} \doteq Bf(\bullet,\bullet)B^{-1}$$

The second equality for program point 4 together with this identity imply that

$$Af(\bullet,\bullet)A^{-1}Af(\bullet,\bullet)\mathbf{x} \doteq Bf(\bullet,\bullet)B^{-1}Bf(\bullet,\bullet)\mathbf{y}$$

from which the third equality for program point 4 as provided by the third iteration follows. Thus, round-robin fixed point iteration reaches the greatest fixed point after the third iteration. ■

2.6 Unrestricted Programs

Our analysis of IR programs relied on the fact that all run-time values of program variables can be uniquely factorized. This was possible since in IR programs the "bottom end" of values can be uniquely identified by means of the ground right-hand sides from R. In general, though, ground right-hand sides could very well also occur as subterms of other right-hand sides in the program. In this case, we can no longer assume that R serves as such a handy set of end marker terms. At first sight, therefore, the monoidal method seems no longer applicable. A second look, however, reveals that the monoidal method essentially fails only, where program variables take *small* values. Again, let R and G denote the set of all ground right-hand sides and the set of all ground subterms of non-ground right-hand sides of assignments in the program, respectively. We call a term in $M_G R$ *small* if it is a ground subterm of a right-hand side of an assignment. Let us denote the finite set of all small terms by S. Thus in particular, $R \subseteq S$. The terms in $M_G R$ which are not small, are called *large*, i.e., we then have:

$$T \coloneqq M_G R = S \uplus L$$

Example 2.6.1. Consider the program fragment consisting of the statements:

$$\mathbf{x}_1 := a; \quad \mathbf{x}_2 := f(\mathbf{x}_1, a); \quad \mathbf{x}_3 := g(\mathbf{x}_2, f(a,a))$$

Then a is a ground right-hand side, and $f(a,a)$ is a ground subterm of a non-ground right-hand side, i.e., $a \in R$ and $f(a,a) \in G$. Since the term $f(a,a)$ is also contained in $M_G R$, it is *small*. ■

Let $\bar{R}$ be the set of *minimal* elements in $M_G R$ which are large, i.e., not contained in S. Then by Theorem 2.3.5, every large term t can be uniquely factored such that $t = t'r$ where $t' \in M_G$ and $r \in \bar{R}$. We then have for small and large terms:

$$S := M_G R \cap (R^* \cup G) \quad \text{and} \quad L := M_G \bar{R}$$

where R^* is the subterm closure of R. For small terms, i.e., for terms in S, on the other hand, we cannot hope for unique factorizations. Since there are finitely many small terms only, we take care of small terms by two means:

- We restrict the formats $[F_{\mathbf{x},\cdot}]$ and $[F_{\cdot,\mathbf{x}}]$ from the last section to the case where the occurring ground terms are large and introduce dedicated sub-formats $[F_{\mathbf{x},s}]$ and $[F_{s,\mathbf{x}}]$ for each small term s in the equalities.
- For T-subsumption, we single out the case of subsumption w.r.t. assignments of large terms only and treat subsumption w.r.t. assignments assigning small terms separately.

The set of non-ground terms is again given as $T' := M_G\mathbf{X}$. Thus, we now consider the following formats of two-variable equalities:

$[F_{\mathbf{x},\mathbf{x}}]$	$As\mathbf{x}$	$\doteq$	$Bt\mathbf{x}$	where $s,t \in M_G$
$[F_{\mathbf{x},\mathbf{y}}]$	$As\mathbf{x}$	$\doteq$	$Bt\mathbf{y}$	where $s,t \in M_G$
$[F_{\cdot,\mathbf{x}}]$	As	$\doteq$	$Bt\mathbf{x}$	where $s \in L$ and $t \in M_G$
$[F_{s,\mathbf{x}}]$	As	$\doteq$	$Bt\mathbf{x}$	where $s \in S$ and $t \in M_G$
$[F_{\mathbf{x},\cdot}]$	$At\mathbf{x}$	$\doteq$	Bs	where $s \in L$ and $t \in M_G$
$[F_{\mathbf{x},s}]$	$At\mathbf{x}$	$\doteq$	Bs	where $s \in S$ and $t \in M_G$

In the following, let us call a substitution σ of program variables *small*, if for every program variable $\mathbf{x}$, $\sigma(\mathbf{x})$ either equals $\mathbf{x}$ or is a small ground term. The notions of satisfiability, equivalence and subsumption restricted to the set T can be inferred by means of the corresponding notions restricted to the set L of large terms only. We have:

- A conjunction ϕ of equalities is T-satisfiable iff there is a small substitution σ such that $\sigma(\phi)$ is L-satisfiable.
- A conjunction ϕ of equalities T-subsumes an equality e, iff for every small substitution σ, $\sigma(\phi)$ L-subsumes $\sigma(e)$.

According to this observation, it seems plausible to consider the analogue of Theorem 2.5.2 for L-subsumption and L-compactness only. We obtain:

Theorem 2.6.1.

*L-**subsumption.** For finite sets E, E' of two-variable equalities of the same format it is decidable whether $\bigwedge E$ L-subsumes $\bigwedge E'$ or not.*

*L-**compactness.** Every L-satisfiable conjunction of a set E of two-variable equalities of the same format is L-subsumed by a conjunction of a subset of at most three equalities in E.*

Proof. For equalities of the formats $[F_{\mathbf{x},\mathbf{y}}]$, $[F_{\mathbf{x},\cdot}]$, $[F_{\cdot,\mathbf{x}}]$ the proofs are analogous to the corresponding proofs for Theorem 2.5.2 where the set T is replaced with the set $L = M_G\bar{R}$, i.e., instead of the set R we rely on the set $\bar{R}$ of unique end marker terms.

Now consider equalities of the format $[F_{s,\mathbf{x}}]$ for a small term $s \in S$. W.l.o.g. let $As \doteq Bt\mathbf{x}$ and $As \doteq Bt'\mathbf{x}$ be two equalities of this format. If $t \neq t'$, then their conjunction is either contradictory, or $t\mathbf{x}, t'\mathbf{x}$ have a ground unifier which maps $\mathbf{x}$ to a value from G—in contradiction to the assumption that $\mathbf{x}$ takes values from L only.

Therefore, each conjunction of a set E of equalities of the format $[F_{s,\mathbf{x}}]$ either is L-equivalent to $\bot$ or to a single equality in E, and the assertion of the theorem follows. The same argument also applies for the format $[F_{\mathbf{x},s}]$. □

Given that L-subsumption is decidable, at least for equalities of the same format, and that also L-compactness holds, we define in the following an approximate T-subsumption of conjunctions of equalities $\bigwedge E$ and $\bigwedge E'$. Let E_F and E'_F denote the subsets of equalities of format F, in E and E', respectively. Then $\bigwedge E$ approximately T-subsumes $\bigwedge E'$ iff for all small substitutions σ, $\bigwedge \sigma(E_F)$ L-subsumes $\bigwedge \sigma(E'_F)$ for all formats F. As a consequence of Theorem 2.6.1, we obtain:

Theorem 2.6.2. *Assume that n is the number of program variables and m is the cardinality of the set S of small terms.*

Approximate T-subsumption. *For finite sets E, E' of two-variable equalities, it is decidable whether $\bigwedge E$ approximately T-subsumes $\bigwedge E'$ or not.*

Approximate T-compactness. *Every T-satisfiable conjunction of a set E of two-variable equalities is approximately T-subsumed by a conjunction of a subset of at most $\mathcal{O}(n^2 \cdot m^2)$ equalities in E.*

Proof. In the following we consider equalities of formats which contain either one or two program variables.

One program variable. Let E' denote a subset of equalities of E of the same format which contains only the program variable $\mathbf{x}$. Then for every small term $c \in S$ we construct a subset $E'_c \subseteq E'$ such that $\bigwedge E'_c[c/\mathbf{x}]$ T-subsumes $\bigwedge E'[c/\mathbf{x}]$. Furthermore, we construct a subset $E'_L \subseteq E'$ which L-subsumes E'. Then the conjunction of $\bigcup_{c\in S} E'_c \cup E'_L$ T-subsumes the conjunction of E'.

For each set E'_c we require at most two equalities according to Corollary 2.3.4, while for the set E'_L we require at most three equalities, according to Theorem 2.6.1. Thus, overall, at most $2m + 3$ equalities are required.

Two program variables. Let E' denote a subset of equalities of E of format $[F_{\mathbf{x},\mathbf{y}}]$ which contains only the distinct program variables $\mathbf{x}, \mathbf{y}$. We proceed as follows.

1. For every $c \in S$, we construct a set $E'_{c,\mathbf{y}} \subseteq E'$ such that $\bigwedge E'_{c,\mathbf{y}}[c/\mathbf{x}]$ T-subsumes $\bigwedge E'[c/\mathbf{x}]$.
2. For every $c \in S$, we construct a set $E'_{\mathbf{x},c} \subseteq E'$ such that $\bigwedge E'_{\mathbf{x},c}[c/\mathbf{y}]$ T-subsumes $\bigwedge E'[c/\mathbf{y}]$.
3. Finally, we construct a set $E'_L \subseteq E'$ such that $\bigwedge E'_L$ L-subsumes $\bigwedge E'$.

Then the conjunction of $\bigcup_{c\in S} E'_{\mathbf{x},c} \cup E'_{c,\mathbf{y}} \cup E'_L$ T-subsumes the conjunction of E'.

For each set $E'_{\mathbf{x},c}$ resp. $E'_{c,\mathbf{y}}$ we require at most $2m+3$ equalities. While for the set E'_L we require at most three equalities according to Theorem 2.6.1. Thus, overall, at most $4m^2 + 6m + 3$ equalities are required for E'.

For each program variable $\mathbf{x}$ we distinguish between $2m + 3$ different formats ($[F_{\mathbf{x},s}]$, $[F_{s,\mathbf{x}}]$, $s \in S$, and $[F_{\mathbf{x},\mathbf{x}}]$,$[F_{\mathbf{x},\cdot}]$, and $[F_{\cdot,\mathbf{x}}]$) of equalities. While for two distinct program variables we only have one format $[F_{\mathbf{x},\mathbf{y}}]$ of equalities. Hence

we conclude that every conjunction E is T-subsumed by a conjunction of a subset of E which contains at most

$$n \cdot (2m+3) \cdot (2m+3) + n \cdot (n-1) \cdot (4m^2+6m+3) \quad \in \quad \mathcal{O}(n^2 \cdot m^2)$$

equalities. This completes the proof. □

Due to Theorem 2.6.2, representations of the greatest solutions of the constraint systems **S** and **R** can be effectively computed. By that, we arrive at our main result:

Theorem 2.6.3. *Assume that all right-hand sides of assignments of a program contain at most one variable. Then all interprocedurally valid two-variable Herbrand equalities can be inferred.*

The proof is analogous to the proof of Theorem 2.5.4—only that Theorem 2.6.2 is used instead of Corollary 2.5.3.

2.7 Revisiting the Introductory Example Program

Let us again consider the program given in Figure 1.1 from the introductory chapter. The corresponding control-flow graphs of the program are depicted in Figure 2.3. The program is not initialization-restricted since the ground right-hand side a is a subterm of the ground right-hand side $f(a,a)$. Hence we consider an unrestricted program for which we have that the set of small terms $S = \{\, a, f(a,a) \,\}$ and the set of minimally large terms $\bar{R} \supseteq \{\, g(a,a), g(f(a,a),f(a,a)) \,\}$.

2.7.1 Computing the Summary for Procedure p

We compute the summary of procedure p by round-robin iteration. Let us denote the term $f(\bullet,\bullet)$ and $g(\bullet,\bullet)$ by $\mathbf{f}$ and $\mathbf{g}$ respectively, i.e.,

$$\mathbf{f} \coloneqq f(\bullet,\bullet) \qquad \text{and} \qquad \mathbf{g} \coloneqq g(\bullet,\bullet).$$

If started with the generic post-condition $A\mathbf{x} \doteq B\mathbf{y}$ at program point 10 we obtain for the program point 5 in the third iteration the following equalities:

$$A\mathbf{x} \doteq B\mathbf{y} \tag{2.4}$$

$$A\mathbf{fgx} \doteq B\mathbf{gfy} \tag{2.5}$$

$$A\mathbf{fgfgx} \doteq B\mathbf{gfgfy} \tag{2.6}$$

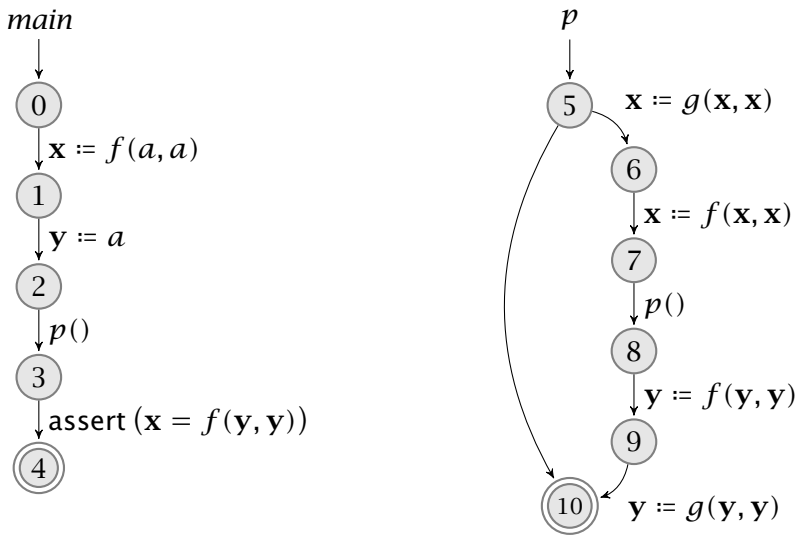


Figure 2.3: The corresponding CFGs for the program from Figure 1.1

For all small substitutions σ of the form $\{ \mathbf{x} \mapsto \mathbf{x}, \mathbf{y} \mapsto s \}$ or $\{ \mathbf{x} \mapsto s, \mathbf{y} \mapsto \mathbf{y} \}$ the equality $\sigma(A\mathbf{x}) \doteq \sigma(B\mathbf{y})$ is of format $[F_{\mathbf{x},s}]$ or $[F_{s,\mathbf{y}}]$. Therefore, in order to decide approximate T-subsumption, the equality has to be kept separately. A fixed point is still not reached since for the remaining two equalities (2.5) and (2.6) approximate T-subsumption is not decidable—we require at least three equalities. In the fourth iteration the additional equality

$$A\mathbf{fgfgfgx} \doteq B\mathbf{gfgfgfy} \tag{2.7}$$

is obtained. In the following we show that for all small substitutions, the equality (2.7) is L-subsumed by the conjunction of the three equalities (2.4), (2.5), and (2.6).

Small substitution $\sigma = \{ \mathbf{x} \mapsto a, \mathbf{y} \mapsto a \}$
If the small substitution is applied to the four initial equalities, then the following four equalities are obtained:

$$Aa \doteq Ba \tag{2.8}$$
$$A\mathbf{f}ga \doteq B\mathbf{gf}a \tag{2.9}$$
$$A\mathbf{fgf}ga \doteq B\mathbf{gfgf}a \tag{2.10}$$
$$A\mathbf{fgfgfg}a \doteq B\mathbf{gfgfgf}a \tag{2.11}$$

The conjunction of the distinct equalities (2.8) and (2.9) is equivalent to a solved equality $Ar_1 \doteq Br_2$ with $r_1, r_2 \in M_G$ where at least one of them equals $\bullet$, if the conjunction is satisfiable. Hence from equality (2.8) we conclude that $r_2 a \doteq r_1 a$ must hold. If $r_1 = \bullet$, then $r_2 = \bullet$ must hold and we conclude that $A \doteq B$ must hold. For $A \doteq B$ the equality (2.9) is unsatisfiable. Likewise, if $r_2 = \bullet$, then $r_1 = \bullet$ must hold and the same argument applies. Since the conjunction of the two equalities (2.8) and (2.9) is unsatisfiable and therefore equivalent to false it subsumes the equalities (2.10) and (2.11). Therefore, the equalities (2.4) and (2.5) L-subsume the equality (2.7) w.r.t. small substitution σ.

Small substitution $\sigma = \{ \mathbf{x} \mapsto f(a,a), \mathbf{y} \mapsto a \}$
If the small substitution is applied to the four initial equalities, then the following four equalitites are obtained:

$$A\mathbf{f}a \doteq Ba \tag{2.12}$$
$$A\mathbf{fgf}a \doteq B\mathbf{gf}a \tag{2.13}$$
$$A\mathbf{fgfgf}a \doteq B\mathbf{gfgf}a \tag{2.14}$$
$$A\mathbf{fgfgfgf}a \doteq B\mathbf{gfgfgf}a \tag{2.15}$$

Again as for the previous case we have that from the first two equalities (2.12) and (2.13) follows that $r_2\mathbf{f}a \doteq r_1 a$ must hold, if the conjunction is satisfiable. If $r_1 = \bullet$, then the equality $r_2\mathbf{f}a \doteq a$ is unsatisfiable. Otherwise, if $r_2 = \bullet$, then $\mathbf{f}a \doteq r_1 a$ is satisfiable with $r_1 = \mathbf{g}$. As a solution we have $A\mathbf{g} \doteq B$ which satisfies all four equalities. Therefore, the equalities (2.4) and (2.5) L-subsume the equality (2.7) w.r.t. the small substitution σ.

Small substitution $\sigma = \{ \mathbf{x} \mapsto a, \mathbf{y} \mapsto f(a,a) \}$
This case is similar to the case of the small substitution $\{ \mathbf{x} \mapsto a, \mathbf{y} \mapsto a \}$, i.e., the conjunction of the equalities (2.4) and (2.5) is unsatisfiable w.r.t. the small substitution σ. Therefore, the equalities (2.4) and (2.5) L-subsume the equality (2.7) w.r.t. the small substitution σ.

Small substitution $\sigma = \{ \mathbf{x} \mapsto \mathbf{x}, \mathbf{y} \mapsto a \}$
For the equality $A\mathbf{x} \doteq B\mathbf{y}$ which has been obtained in the first iteration we have that $\sigma_5(A\mathbf{x}) \doteq \sigma_5(B\mathbf{y})$ is an equality of format $[F_{\mathbf{x},a}]$. Therefore, the equality has to be kept separately. Let us now consider the remaining three equalities of iteration 4. If the small substitution σ is applied to these, then

the following three equalities are obtained:

$$A\mathbf{fgx} \doteq Bg(f(a,a),f(a,a)) \tag{2.16}$$

$$A\mathbf{fgfgx} \doteq B\mathbf{gf}g(f(a,a),f(a,a)) \tag{2.17}$$

$$A\mathbf{fgfgfgx} \doteq B\mathbf{gfgf}g(f(a,a),f(a,a)) \tag{2.18}$$

Note that every large term is represented by its unique factorization. For example, by applying the substitution σ to the term $\mathbf{gfgfy}$ the resulting term $\mathbf{gfgf}a$ is uniquely factorized by $\mathbf{gf}g(f(a,a),f(a,a))$ where $g(f(a,a),f(a,a))$ is a unique end-marker term in $\bar{R}$ and $\mathbf{gf} \in M_G$.

For L-subsumption we have that for each equality (2.16), (2.17), and (2.18) $\mathbf{x} = x'g(f(a,a),f(a,a))$ for some $x' \in M_G$ must hold. By bottom cancellation of the term $g(f(a,a),f(a,a))$ we receive the following three equalities where each term contains $\bullet$, i.e., each term can be uniquely factorized:

$$A\mathbf{fg}x' \doteq B \tag{2.19}$$

$$A\mathbf{fgfg}x' \doteq B\mathbf{gf} \tag{2.20}$$

$$A\mathbf{fgfgfg}x' \doteq B\mathbf{gfgf} \tag{2.21}$$

We have that the balance $|\mathbf{fg}x'| \leq |\mathbf{fgfg}x'|$ and $|\mathbf{fg}x'| \leq |\mathbf{fgfgfg}x'|$ holds. Hence the equalities (2.19) and (2.20) imply the equality

$$A\mathbf{fgfg}x'x'^{-1}\mathbf{g}^{-1}\mathbf{f}^{-1}A^{-1} \doteq B\mathbf{gf}B^{-1}$$

which is equivalent to the maximally canceled equality

$$A\mathbf{fg}A^{-1} \doteq B\mathbf{gf}B^{-1}. \tag{2.22}$$

Similarly we have that the equalities (2.19) and (2.21) imply the equality

$$A\mathbf{fgfgfg}x'x'^{-1}\mathbf{g}^{-1}\mathbf{f}^{-1}A^{-1} \doteq B\mathbf{gfgf}B^{-1}$$

which is equivalent to the maximally canceled equality

$$A\mathbf{fgfg}A^{-1} \doteq B\mathbf{gfgf}B^{-1}. \tag{2.23}$$

For the equalities (2.22) and (2.23) we have that the balance $|\mathbf{fgfg}| = 2 \cdot |\mathbf{fg}|$. Therefore, the two equalities imply the equality

$$A\mathbf{fgfgg}^{-1}\mathbf{f}^{-1}\mathbf{g}^{-1}\mathbf{f}^{-1}A^{-1} \doteq B\mathbf{gfgff}^{-1}\mathbf{g}^{-1}\mathbf{f}^{-1}\mathbf{g}^{-1}B^{-1}$$

which is equivalent to the maximally canceled equality

$$A\epsilon A^{-1} \doteq B\epsilon B^{-1}.$$

We conclude that the equality (2.18) is subsumed by the two equalities (2.16) and (2.17). Therefore, the equalities (2.5) and (2.6) L-subsume the equality (2.7) w.r.t. the small substitution σ.

Small substitutions
$\{ \mathbf{x} \mapsto \mathbf{x}, \mathbf{y} \mapsto f(a,a) \}$, $\{ \mathbf{x} \mapsto f(a,a), \mathbf{y} \mapsto \mathbf{y} \}$, **and** $\{ \mathbf{x} \mapsto a, \mathbf{y} \mapsto \mathbf{y} \}$
These cases are all similar to the case of the small substitution $\{ \mathbf{x} \mapsto \mathbf{x}, \mathbf{y} \mapsto a \}$, i.e., the equalities (2.5) and (2.6) L-subsume the equality (2.7) w.r.t. the corresponding small substitution.

Small substitution $\sigma = \{ \mathbf{x} \mapsto \mathbf{x}, \mathbf{y} \mapsto \mathbf{y} \}$
In this case we have that σ is the identity substitution. Let us consider the following three equalities:

$$A\mathbf{fgx} \doteq B\mathbf{gfy} \tag{2.24}$$
$$A\mathbf{fgfgx} \doteq B\mathbf{gfgfy} \tag{2.25}$$
$$A\mathbf{fgfgfgx} \doteq B\mathbf{gfgfgfy} \tag{2.26}$$

Since all three equalities are distinct, according to unique factorization A must be a prefix of B or vice versa. Therefore, $A\mathbf{fg}u \doteq B\mathbf{gf}v$ must hold for some $u, v \in M_G$ where at least one equals $\bullet$. By top cancellation this implies that $v\mathbf{x} \doteq u\mathbf{y}$ must hold. Hence we conclude that the three equalities (2.24), (2.25), and (2.26) imply the following three equalities where each term contains $\bullet$, i.e., each term can be uniquely factorized:

$$A\mathbf{fg}u \doteq B\mathbf{gf}v \tag{2.27}$$
$$A\mathbf{fgfg}u \doteq B\mathbf{gfgf}v \tag{2.28}$$
$$A\mathbf{fgfgfg}u \doteq B\mathbf{gfgfgf}v \tag{2.29}$$

We have that the balance $|\mathbf{fg}u| \leq |\mathbf{fgfg}u|$ and $|\mathbf{fg}u| \leq |\mathbf{fgfgfg}u|$ holds. Hence the equalities (2.27) and (2.28) imply the equality

$$A\mathbf{fgfg}uu^{-1}\mathbf{g}^{-1}\mathbf{f}^{-1}A^{-1} \doteq B\mathbf{gfgf}vv^{-1}\mathbf{f}^{-1}\mathbf{g}^{-1}B^{-1}$$

which is equivalent to the maximally canceled equality

$$A\mathbf{fg}A^{-1} \doteq B\mathbf{gf}B^{-1}. \tag{2.30}$$

Similarly we have that the equalities (2.27) and (2.29) imply the equality

$$A\mathbf{fgfgfg}uu^{-1}\mathbf{g}^{-1}\mathbf{f}^{-1}A^{-1} \doteq B\mathbf{gfgfgf}vv^{-1}\mathbf{f}^{-1}\mathbf{g}^{-1}B^{-1}$$

which is equivalent to the maximally canceled equality

$$A\mathbf{fgfg}A^{-1} \doteq B\mathbf{gfgf}B^{-1}. \tag{2.31}$$

From the previous case we already know that the equalities (2.30) and (2.31) imply the equality $A\epsilon A^{-1} \doteq B\epsilon B^{-1}$. Therefore, the equalities (2.5) and (2.6) L-subsume the equality (2.7) w.r.t. the small substitution σ.

Conclusion
In total we have that for all small substitutions the equality (2.7) is L-subsumed by the conjunction of the three equalities (2.4), (2.5) and (2.6). Therefore, fixed point iteration terminates and as a summary for procedure p we have that

$$[\![5]\!]^\top (A\mathbf{x} \doteq B\mathbf{y}) = (A\mathbf{x} \doteq B\mathbf{y} \wedge A\mathbf{fgx} \doteq B\mathbf{gfy} \wedge A\mathbf{fgfgx} \doteq B\mathbf{gfgfy}) .$$

2.7.2 Computing Reachability Information

Assume we want to compute the reachability information at program end, i.e., at program point 4. In the course of weakest precondition computation we have that an assert statement has the same semantics as a skip statement. Therefore, for the reachability information at program point 4 the transformer $[4]^\top = [3]^\top$. For the generic postcondition $A\mathbf{x} \doteq B\mathbf{y}$ we have:

$$\begin{aligned}
&[3]^\top (A\mathbf{x} \doteq B\mathbf{y}) \\
&= ([0]^\top \circ [1]^\top \circ [2]^\top) (A\mathbf{x} \doteq B\mathbf{y}) \\
&= ([\![\mathbf{x} \coloneqq \mathbf{f}a]\!]^\top \circ [\![\mathbf{y} \coloneqq a]\!]^\top \circ [\![5]\!]^\top) (A\mathbf{x} \doteq B\mathbf{y}) \\
&= ([\![\mathbf{x} \coloneqq \mathbf{f}a]\!]^\top \circ [\![\mathbf{y} \coloneqq a]\!]^\top) (A\mathbf{x} \doteq B\mathbf{y} \wedge A\mathbf{fgx} \doteq B\mathbf{gfy} \wedge A\mathbf{fgfgx} \doteq B\mathbf{gfgfy}) \\
&= (A\mathbf{f}a \doteq Ba \wedge A\mathbf{fgf}a \doteq B\mathbf{gf}a \wedge A\mathbf{fgfgf}a \doteq B\mathbf{gfgf}a)
\end{aligned}$$

The resulting three ground equalities are equivalent to one solved equality

$$A\mathbf{f} \doteq B$$

according to Theorem 2.3.3. From the solved equality we derive all valid two-variable Herbrand equalities of the form $\mathbf{x} \doteq t\mathbf{y}$ or $t\mathbf{x} \doteq \mathbf{y}$. If $B = \bullet$, then the equality $A\mathbf{g} \doteq B$ is not satisfiable. Otherwise, if $A = \bullet$, then the equality $A\mathbf{f} \doteq B$ has exactly one solution with $B = \mathbf{f}$. Therefore, we conclude that the equality

$$\mathbf{x} \doteq \mathbf{f}\mathbf{y}$$

is the only valid two-variable Herbrand equality between the program variables $\mathbf{x}$ and $\mathbf{y}$ at the program points 3 and 4. Consequently we have proven that the assertion $\mathbf{x} = f(\mathbf{y}, \mathbf{y})$ at program line 5 of Figure 1.1 always holds.

2.8 Multi-variable Equalities

In this section, we extend our methods to arbitrary equalities such as

$$\mathbf{x} \doteq f(g\mathbf{y}, \mathbf{z})$$

where, w.l.o.g., the left-hand side is a plain program variable while the right-hand side is a term possibly containing occurrences of more than one variable. Note that an equality where the left-hand side is not a plain program variable is either contradictory as e.g. $g\mathbf{x} \doteq f(\mathbf{y}, \mathbf{z})$ or can be massaged via top cancellation into an equivalent conjunction of equalities where each left-hand side of an equality is a plain variable. Still, we consider programs where each right-hand side of an assignment contains occurrences of at most one variable only. Here, we indicate how for any program point u and any given candidate Herbrand equality $\mathbf{x} \doteq t$, we verify whether or not the equality is valid whenever u is reached. There are only constantly many candidate equalities of this form, namely, all equalities which hold for a variable assignment σ_u computed by a single run of the program reaching u. Since such a single run can be effectively computed before-hand, we conclude:

Theorem 2.8.1. *Assume that all right-hand sides of assignments of a program contain at most one variable. Then all interprocedurally valid Herbrand equalities can be inferred.*

Now consider the single Herbrand equality $\mathbf{x} \doteq t$, where t contains occurrences of the program variables $\mathbf{y}_1, \ldots, \mathbf{y}_k$. Then we construct new generic postconditions as follows. First, we consider all substitutions σ which map each variable $\mathbf{y}_i$ in t either to a fresh template variable C_i or an expression $A_i\mathbf{y}_i'$ for a fresh template variable A_i and any program variable $\mathbf{y}_i'$. Then the new generic postconditions are of the form $\mathbf{x}' \doteq t'$ where $\mathbf{x}'$ is any program variable, and t' is a subterm of $t\sigma$. Note that this set may be large but is still finite. In a practical implementation, we may, however, tabulate for each procedure the weakest preconditions only for those postconditions which are really required. Since we envision that for realistic programs, only few of these equalities for each procedure will be necessary to prove the queried assertion at target point u, the potential exponential blow-up is hopefully not an obstacle.

Example 2.8.1. Assume the equality we are interested in is $\mathbf{x} \doteq f(g\mathbf{y}, \mathbf{z})$, then, e.g.,

$$\mathbf{x} \doteq f(gA_1\mathbf{y}, A_2\mathbf{z}) \qquad \mathbf{y} \doteq f(gA_1\mathbf{x}, A_2\mathbf{z})$$

are new generic postconditions to be considered, as well as

$$\mathbf{z} \doteq f(gC, A\mathbf{y}) \qquad \mathbf{y} \doteq f(gA\mathbf{z}, C)$$ ■

Starting from a new generic postcondition $\mathbf{x} \doteq p$, repeatedly computing weakest preconditions w.r.t. assignments may result in conjunctions of equalities which can be simplified by top cancellation to one of the following forms:

- $s \doteq C_i$ or $s \doteq A_i t_i$ where s and t_i contain occurrences of at most one program variable each;
- $\mathbf{y} \doteq p'$, i.e., the left-hand side is a plain program variable, and the right-hand side p' is obtained from a subterm of p by substituting each occurrence of a program variable $\mathbf{y}_i$ with some term t_i containing occurrences of at most one program variable each.

Example 2.8.2. Consider, e.g., the generic postcondition $\mathbf{x} \doteq f(gA_1\mathbf{y}, A_2\mathbf{z})$. Then

$$\begin{aligned}[\![\mathbf{x} := f(\mathbf{x}, h\mathbf{x})]\!]^\top(\mathbf{x} \doteq f(gA_1\mathbf{y}, A_2\mathbf{z})) &= f(\mathbf{x}, h\mathbf{x}) \doteq f(gA_1\mathbf{y}, A_2\mathbf{z}) \\ &= (\mathbf{x} \doteq gA_1\mathbf{y}) \wedge (h\mathbf{x} \doteq A_2\mathbf{z})\end{aligned}$$

which means that we equivalently obtain two two-variable equalities. Likewise, for an assignment to one of the program variables on the right, we have:

$$[\![\mathbf{y} := f(b, \mathbf{y})]\!]^\top(\mathbf{x} \doteq f(gA_1\mathbf{y}, A_2\mathbf{z})) = (\mathbf{x} \doteq f(gA_1 f(b, \mathbf{y}), A_2\mathbf{z}))$$

which is an equality of the form described in the second item. ∎

The equalities from the first item contain at most one program variable on each side. They can be dealt with in the same way as we did for plain two-variable equalities. They are even somewhat simpler, in that only one template variable occurs (instead of two). The equalities of the second item, on the other hand, we may group into equalities which agree in the variable on the left as well as in the constructor applications outside the template variables A_i. Of each such group it suffices to keep exactly one equality. Any conjunction with another equality from the same group will allow us to simplify the second equality to a conjunction of equalities with at most one program variable on each side.

Example 2.8.3. Assume that we are given the conjunction of the two equalities:

$$\mathbf{x} \doteq f(gA_1\mathbf{y}, A_2\mathbf{z}) \qquad \mathbf{x} \doteq f(gA_3h\mathbf{y}, A_4g\mathbf{z})$$

This conjunction is equivalent to the first equality together with:

$$f(gA_1\mathbf{y}, A_2\mathbf{z}) \doteq f(gA_3h\mathbf{y}, A_4g\mathbf{z})$$

The latter equality, now, is equivalent to the conjunction of:

$$A_1\mathbf{y} \doteq A_3h\mathbf{y} \qquad A_2\mathbf{z} \doteq A_4g\mathbf{z}$$

which is a finite conjunction of two-variable equalities. ∎

Thus, in the course of weakest precondition computation for any of the new generic postconditions, we obtain conjunctions which (up to finitely many exceptions) consists of two-variable equalities only, to which we can apply our methods from Section 2.6. In summary, we thus find that it can be effectively verified whether or not a general Herbrand equality is interprocedurally valid at a given program point u.

2.9 Global and Local Program Variables

So far we considered programs with global variables only. In this section we extend the program model from Section 2.1 in order to also deal with programs which contain global as well as local variables. Assume that the finite set of program variables $\mathbf{X}$ contains a subset $\mathbf{L} \subseteq \mathbf{X}$ consisting of *local* program variables, while the remaining variables are considered as *global*. The scope of local variables is meant to be restricted to the body of the current procedure. At the start of a procedure call, the fresh local variables are assumed to be *uninitialized*, i.e., have any value, whereas at procedure exit, the current locals are abandoned while the locals of the calling procedure are recovered. By means of global variables, this simple model already allows to realize call-by-value variable passing as well as the returning of functional results.

In order to deal with a non-empty set of locals, we enhance the weakest precondition calculus by an operator $\mathcal{H}$ which takes the weakest precondition transformation realized by the body of a procedure as an argument, and returns the weakest precondition transformation of the procedure call. For a given weakest precondition transformation f, the weakest precondition transformation $\mathcal{H}(f)$ is defined as follows.

$$
\begin{array}{lcll}
\mathcal{H}(f)(A\mathbf{x} \doteq C) & = & \forall \mathsf{L}.\, f(A\mathbf{x} \doteq C) & \mathbf{x} \text{ global} \\
\mathcal{H}(f)(A\mathbf{x} \doteq B\mathbf{y}) & = & \forall \mathsf{L}.\, f(A\mathbf{x} \doteq B\mathbf{y}) & \mathbf{x}, \mathbf{y} \text{ global} \\
\mathcal{H}(f)(A\mathbf{x} \doteq B\mathbf{y}) & = & (\forall \mathsf{L}.\, f(A\mathbf{x} \doteq C))[B\mathbf{y}/C] & \mathbf{x} \text{ global}, \mathbf{y} \text{ local} \\
\mathcal{H}(f)(A\mathbf{x} \doteq B\mathbf{y}) & = & (\forall \mathsf{L}.\, f(A\mathbf{y} \doteq C))[B/A, A\mathbf{x}/C] & \mathbf{x} \text{ local}, \mathbf{y} \text{ global} \\
\mathcal{H}(f)(e) & = & e & e \text{ contains no globals}
\end{array}
$$

where L is the sequence of local variables in $\mathbf{L}$. Accordingly, the constraints for call edges in the constraint systems $\mathbf{S}$ and $\mathbf{R}$ must be changed into:

$$[\![u]\!]^\top \implies \mathcal{H}([\![s_p]\!]^\top) \circ [\![v]\!]^\top \qquad \text{for each } (u, p(), v) \in \mathbb{E}$$

and

$$[v]^\top \implies [u]^\top \circ \mathcal{H}([\![s_p]\!]^\top) \qquad \text{for each } (u, p(), v) \in \mathbb{E}$$

respectively.

2.10 Soundness of the Analysis

In order to prove the soundness of every solution to the constraint systems **S** and **R** we first introduce some basic concepts. A path π is a finite sequence $k_1\, k_2 \cdots k_n$ of edges $k_i \in \mathbb{E}$ with $k_i = (u, \_, v)$ and $k_{i+1} = (v, \_, w)$. The semantics $[\![k]\!]$ of an edge $k = (u, lab, v)$ is defined by $[\![k]\!] = [\![lab]\!]$. The semantics $[\![\pi]\!]$ of a path π is then given as the composition of edge effects

$$[\![\pi]\!] = [\![k_n]\!] \circ \cdots \circ [\![k_1]\!]$$

and likewise in the course of weakest precondition computation

$$[\![\pi]\!]^\top = [\![k_1]\!]^\top \circ \cdots \circ [\![k_n]\!]^\top$$
$$[\pi]^\top = [k_n]^\top \circ \cdots \circ [k_1]^\top.$$

An *interprocedural valid path* is a path where for each return of a procedure p a corresponding enter to the procedure p exists, such that in-between these pairs no enter to a procedure p' without a corresponding return occurs. Consider the context-free grammar with non-terminals $A_{u,v}$ where $u, v \in \mathbb{N}$ are program points, start non-terminal $A_{s_{main}, r_{main}}$, and production rules

$$\begin{array}{ll} A_{u,u} \to \epsilon & \\ A_{u,w} \to (u, s, v)\, A_{v,w} & (u, s, v) \in \mathbb{E},\ s \text{ statement} \\ A_{u,w} \to (u, \langle p \rangle, s_p)\, A_{s_p, r_p}\, (r_p, \langle / p \rangle, v)\, A_{v,w} & (u, p(), v) \in \mathbb{E},\ p \text{ procedure} \end{array}$$

where ϵ denotes the empty path and the labels $\langle p \rangle$ and $\langle / p \rangle$ denote the entering and returning of a procedure call, respectively. We silently lift the set $\mathbb{E}$ of edges to the set containing additionally edges $(u, \langle p \rangle, s_p)$ and $(r_p, \langle / p \rangle, v)$ for each edge $(u, p(), v) \in \mathbb{E}$. Let us then denote by $\Pi(u, v)$ the set of paths which can be derived from the non-terminal $A_{u,v}$, i.e.,

$$\Pi(u, v) \coloneqq \left\{ \pi \in \mathbb{E}^* \;\middle|\; A_{u,v} \to^* \pi \right\}$$

for $u \neq v$. Note that if $\Pi(u, v) = \emptyset$ holds, then there exists no interprocedurally valid path between the program points u and v. Otherwise, if $\Pi(u, v) \neq \emptyset$ holds, then $\Pi(u, v)$ contains all interprocedurally valid paths between the program points u and v.

Here we follow the idea presented in [SWH12] and term any path π in $\Pi(u, v)$ as a *same-level computation*, i.e., for each opening parenthesis $\langle f \rangle$ occurring in a path π, there is also a corresponding closing parenthesis $\langle / f \rangle$ in π such

that π is well-parenthesized. Furthermore, let us term any prefix of a path in $\Pi(s_{main}, r_{main})$ ending at node u as a *u-reaching computation*, i.e., each u-reaching computation π is of the form

$$\pi = \langle s_{main} \rangle \pi_0 \langle p_1 \rangle \pi_1 \cdots \langle p_k \rangle \pi_k (\_, \_, u)$$

for procedures p_i and same-level computations π_i. The set of all u-reaching computations is then given by

$$\Pi(u) \coloneqq \{\ \pi \mid \pi\pi' \in \Pi(s_{main}, r_{main}), \pi = \pi_0(\_, \_, u)\ \}$$

for $u \neq s_{main}$. The semantics of a path is then defined as follows

$$\begin{aligned} [\![k\pi]\!] &= [\![\pi]\!] \circ [\![k]\!] && \text{for a normal edge } k \\ [\![\langle p \rangle \pi \langle / p \rangle \pi']\!] &= [\![\pi']\!] \circ \mathcal{H}([\![\pi]\!]) && \text{for a procedure } p \text{ and} \\ & && \text{same-level computation } \pi \end{aligned}$$

By that we arrive at the following theorem:

Theorem 2.10.1. *Let $[\![\cdot]\!]^\top_*$ respectively $[\cdot]^\top_*$ be solutions to the constraint systems **S** respectively **R**. We then have*

$$\begin{aligned} [\![s_p]\!]^\top_* &\implies \bigwedge \left\{ [\![\pi]\!]^\top \;\middle|\; \pi \in \Pi(s_p, r_p) \right\} \\ [u]^\top_* &\implies \bigwedge \left\{ [\pi]^\top \;\middle|\; \pi \in \Pi(u) \right\} \end{aligned}$$

for all procedures p and all program points u.

Proof. We remark that composition of monotone funcions yields a monotone function. In particular we have for monotone functions f, f', with $f \implies f'$ and g, g', with $g \implies g'$ that $(f \circ f') \implies (g \circ g')$ holds.

We proceed by structural induction over the path π. For the induction base let $\pi = (u, s, r_p)$ where s is a statement. Then $[\![u]\!]^\top_* \implies [\![s]\!]^\top \circ [\![r_p]\!]^\top_* = [\![s]\!]^\top \circ \mathsf{Id} = [\![s]\!]^\top = [\![\pi]\!]^\top$. Now assume that for any path $\pi' = (v, \_, \_)\pi'''$ with $\pi''\pi' \in \Pi(s_p, r_p)$ the claim $[\![v]\!]^\top_* \implies [\![\pi']\!]^\top$ holds. As the induction hypothesis we have for all procedures f that $[\![s_f]\!]^\top_* \implies [\![\pi_f]\!]^\top$ for all paths $\pi_f \in \Pi(s_f, r_f)$. Let $\pi = (u, s, v)\pi'$ where s is a statement. Then $[\![u]\!]^\top_* \implies [\![s]\!]^\top \circ [\![v]\!]^\top_*$. By assumption $[\![v]\!]^\top_* \implies [\![\pi']\!]^\top$ holds and therefore $[\![u]\!]^\top_* \implies [\![\pi]\!]^\top$ from which the assertion of this claim follows. Now let $\pi = (u, f(), v)\pi'$ where f is a procedure. Then $[\![u]\!]^\top_* \implies [\![s_f]\!]^\top_* \circ [\![v]\!]^\top_*$. Since by induction hypothesis $[\![s_f]\!]^\top_* \implies [\![\pi_f]\!]^\top$ for any path $\pi_f \in \Pi(s_f, r_f)$ and $[\![v]\!]^\top_* \implies [\![\pi']\!]^\top$, we have that $[\![u]\!]^\top_* \implies [\![\pi]\!]^\top$ and the assertion of this claim also follows.

This proves the assertion for the constraint system **S**. A similar argument holds for the solution of the constraint system **R**. □

From Theorem 2.10.1 we conclude that each solution to the constraint systems **S** resp. **R** is sound. In the following we reason about the precision of the greatest solutions to the constraint systems. Note here the intraprocedural coincidence theorem from Kildall [Kil73] who pointed out that if every function $[\![\cdot]\!]$ of a constraint system corresponding to an intraprocedural program is $\sqcap$-distributive, then the greatest solution $[\![\cdot]\!]_*$ to the constraint system equals the meet-over-all-paths, i.e., $[\![u]\!]_* = \bigsqcap \{ [\![\pi]\!] \mid \pi \in \Pi(u) \}$ for all u.

Lemma 2.10.2. *For every statement s, its weakest precondition transformer $[\![s]\!]^\top$ is $\sqcap$-distributive. The transformer $\mathcal{H}$ is $\sqcap$-distributive, if applied to a $\sqcap$-distributive transformer.*

Proof. Let σ be a variable assignment and ϕ_1, ϕ_2 be conjunctions of equalities, respectively. We then have that $\sigma(\phi_1 \sqcap \phi_2) = \sigma(\phi_1) \sqcap \sigma(\phi_2)$ holds. Since a weakest precondition transformer for a statement s equals a variable assignment the transformer $[\![s]\!]^\top$ is also $\sqcap$-distributive. Furthermore, we have that the composition of $\sqcap$-distributive functions yields again a $\sqcap$-distributive function, and the universal closure as well as any substitution commutes with conjunction. Hence, the transformer $\mathcal{H}(f)$ is also $\sqcap$-distributive. □

Theorem 2.10.3 (Seidl et al. [SWH12])**.** *Let us assume that for each procedure p there exists at least one same-level computation from the entry point s_p to every program point u of p, i.e., $\Pi(s_p, u) \neq \emptyset$. Let us further assume that all transformer $[\![\cdot]\!]^\top$ of normal edges and the transformer $\mathcal{H}$ are $\sqcap$-distributive. We then have for the greatest solution $[\![\cdot]\!]^\top_*$ to the constraint system **S** and for each procedure p*

$$[\![s_p]\!]^\top_* = \bigwedge \left\{ [\![\pi]\!]^\top \;\middle|\; \pi \in \Pi(s_p, r_p) \right\}. \qquad \square$$

Theorem 2.10.4 (Seidl et al. [SWH12])**.** *Let us assume that for each program point u there exists at least one u-reaching computation, i.e., $\Pi(u) \neq \emptyset$. Let us further assume that all transformer $[\![\cdot]\!]^\top$ of normal edges and the transformer $\mathcal{H}$ are $\sqcap$-distributive. We then have for the greatest solution $[\cdot]^\top_*$ to the constraint system **R** and for each program point u*

$$[u]^\top_* = \bigwedge \left\{ [\pi]^\top \;\middle|\; \pi \in \Pi(u) \right\}. \qquad \square$$

From the theorems 2.10.3 and 2.10.4 we conclude that the greatest solutions to the constraint systems **S** and **R** equal their corresponding meet-over-all-paths, respectively.

2.11 Analysis of Computational Complexity

In the following we indicate how the presented algorithms for inferring interprocedurally valid Herbrand equalities can be realized in polynomial time. Crucial for the complexity is the size of representations of occurring terms. Note that already the factorization of a term results in a succinct representation by sharing isomorphic subtrees. Still, the *depth* of occurring terms may grow exponentially in a program with procedures.

Example 2.11.1. Consider the following program fragment consisting of procedures p_n and two global variables $\mathbf{x}$ and $\mathbf{y}$:

$$\begin{array}{ll} p_i & \{\ p_{i-1}();\ p_{i-1}();\ \} \\ p_0 & \{\ \mathbf{x} := f(\mathbf{x},\mathbf{x});\ \mathbf{y} := f(\mathbf{y},\mathbf{y});\ \} \end{array}$$

The weakest precondition of a generic postcondition $A\mathbf{x} \doteq B\mathbf{y}$ for a procedure p_n is then given by a single equality $Af(\bullet,\bullet)^{2^n}\mathbf{x} \doteq Bf(\bullet,\bullet)^{2^n}\mathbf{y}$ with exponentially deep terms on both sides of the equality. ■

Hence, in order to arrive at polynomial algorithms, polynomially sized representations must be provided for all occurring terms which additionally support the required operations on terms in polynomial time. For trees, *tree straight-line programs* (TSLP, for short) have been proposed which efficiently represent trees by context-free *tree* grammars (see [Sch13; Loh15] for recent overviews). Polynomial algorithms for equality of the represented trees, however, are only known in case that the tree grammars in question are *linear*—meaning that each parameter of a rule occurs in the corresponding right-hand side at most once. Our factorizations of trees, however, may easily introduce *non-linear* terms. Therefore, we apply compression only to elements from the free monoid M_G. We use ordinary straight-line programs (SLP for short)—but with the understanding that individual letters are irreducible trees. For plain symbols (corresponding to unary constructors only), algorithms based on such a representation have been sketched in [GT07]. Thus in our application, an SLP P of *size* k consists of a sequence of definitions

$$X_i \to \alpha_i \qquad i = 1, \ldots, k$$

where either $k = 1$ and $\alpha_i = \bullet$, or each right-hand side α_i is either of the form $X_j X_l$ for unknowns X_j, X_l with $i < j, l$ or a single irreducible term $t \in M_G$. Given a suitable ordering on the unknowns together with an initial unknown, we may consider P also as a *set* of definitions of unknowns.

Beyond the size, we are also interested in the *depth*, i.e., the length h of the longest chain of unknowns $Y_1, \dots, Y_h$ in P such that $Y_1 \to \alpha_1 Y_2 \alpha'_1, \dots, Y_{h-1} \to \alpha_{h-1} Y_h \alpha'_{h-1}$ occur among the definitions in P for suitable α_i, α'_i.

An SLP can also be considered as a context-free grammar (in Chomsky Normal Form) generating a single term in M_G. Formally, the term $[\![P]\!]$ represented by P is defined by $[\![P]\!] = [\![X_1]\!]_P$ where

$$\begin{aligned} [\![X_i]\!]_P &= [\![X_j]\!]_P [\![X_l]\!]_P && (X_i \to X_j X_l) \in P \\ [\![X_i]\!]_P &= t && (X_i \to t) \in P \text{ and } t \in M_G \end{aligned}$$

We remark that in linear time in the size of P, we can determine the *length* of the represented element in M_G, which is defined by:

$$\begin{aligned} \|X_i\|_P &= \|X_j\|_P + \|X_l\|_P && (X_i \to X_j X_l) \in P \\ \|X_i\|_P &= 0 && (X_i \to \bullet) \in P \\ \|X_i\|_P &= 1 && (X_i \to t) \in P \text{ and } t \in M_G \setminus \{\bullet\} \end{aligned}$$

An SLP in Chomsky Normal Form of size k cannot produce a word larger than 2^k. Therefore, the length of each word which it generates can be described by k bits. For such numbers, basic operations as equality and addition can be done in linear time in k.

In order to avoid repeated computation of lengths, we assume that every unknown occurring during the analysis will once for all be annotated with its length. For later use, we collect a set of basic algorithms for SLPs (see, e.g., [Loh12]).

Theorem 2.11.1. *The following tasks can be realized in polynomial time:*

1. *Given an SLP P representing a term $t = t_1 \cdots t_k \in M_G$. Determine an SLP Q for the reverse $t' = t_k \cdots t_1$ of t such that Q has the same size and depth as P.*

2. *Given an SLP P representing a term $t = t_1 \cdots t_k \in M_G$ of some length k, and some number $0 \le h \le k$. Determine an SLP Q for the prefix $t_1 \cdots t_h$ (suffix $t_{k-h} \cdots t_k$) of t of length h. The number of new definitions in Q is bounded by the depth of P, and the depth of Q is not increased.*

3. *Given SLPs P and Q for terms $t, t' \in M_G$. Determine whether or not $t = t'$.*

4. *Given SLPs P and Q for terms $t, t' \in M_G$. Determine the length of the longest common prefix (suffix) of t, t', i.e., let $t = t_1 \cdots t_n$ and $t' = t'_1 \cdots t'_m$ where each t_i, t'_i is irreducible, then the longest common prefix is of length k if $t_i = t'_i$ for $1 \leq i \leq k$ and $t_{k+1} \neq t'_{k+1}$ and likewise the longst common suffix is of length k if $t_{n-i} = t'_{m-i}$ for $0 \leq i < k$ and $t_{n-k} \neq t'_{m-k}$.*

5. *Given SLPs P and Q for terms $t, t' \in M_G$. Determine an SLP for tt'. At most one new definition is introduced and also the depth is increased at most by one.*

Proof. An SLP for the reverse of t is obtained from P by introducing a fresh copy of unknowns X' for every unknown X in P together with a definition $X' \to f$ if $X \to f$ with $f \in M_G$, and a definition $X' \to Z'Y'$ if P has a definition $X \to YZ$. This new SLP clearly generates the reverse of the SLP P—proving assertion 1.

For a proof of assertion 2, we only consider the construction of an SLP for the prefix of t of length h. The case where $h = 0$ is trivial. Therefore, assume that $h > 0$. We construct the new SLP by successively introducing fresh unknowns X' for the unknowns X on a path in P. in order to do so, we maintain the sum of the lengths l of the unknowns to the left of the path. We start with the initial unknown X_1 of P where $l = 0$ with corresponding fresh unknown X'_1. In general, assume that $l < h$, and we have reached an unknown X with corresponding fresh unknown X'. First assume that the definition of X in P is given by $X \to f$ for some irreducible term $f \in M_G$. In this case, $h = l + 1$, and we set the definition of X' to $X' \to f$. Then assume that the definition of X in P is given by $X \to YZ$. If $h \leq l + \|Y\|_P$, then we introduce a fresh copy Y' for Y and the definition $X' \to Y'$ for X', and proceed with Y'. If $l + \|Y\|_P < h$, then we introduce a fresh copy Z' for Z and the definition $X' \to YZ'$ for X' and proceed with Z'. The resulting set of definitions, though, may not meet our assumptions on SLPs. The definitions with single unknowns in their right-hand sides, can however, be removed in polynomial time by a technique similar to the removal of chain rules in context-free grammars.

Polynomial time algorithms for deciding equivalence of SLPs were independently discovered by Hirshfeld et al. [HJM96], Mehlhorn et al. [MSU97], and Plandowski [Pla94] proving assertion 3. The algorithms can be applied to obtain a polynomial time algorithm for determining the length of longest common prefixes of elements in a free monoid as claimed in assertion 4. First, the algorithm from assertion 3 can be extended to decide whether or not t is a *prefix* of t' by first determining the lengths h and h' of t and t', respectively. If $h > h'$, t is not a prefix of t'. Otherwise, we may determine an SLP Q' of Q

representing the prefix of t' of length h which then is checked for equivalence with P. In the next step, that algorithm is extended to the case where t is not necessarily a prefix of t' by performing binary search on the prefixes of t.

Finally, consider assertion 5. If t or t' equals $\bullet$, the concatenation is trivial. So assume that neither t nor t' equal $\bullet$, and that the initial unknowns of the SLPs P and Q equal X_1 and Y_1, respectively. Let X_0 denote a fresh unknown. Then the term tt' can be represented by the SLP $P \cup Q$ together with the initial definition $X_0 \to X_1 Y_1$. □

The size of a term $t \in \mathcal{T}_\Omega(\mathbf{X}) \cup \mathcal{T}_\Omega(\bullet)$ is given by $\mathsf{size}(t)$ which is recursively defined as follows:

$$\begin{array}{lcll} \mathsf{size}(t) & = & 1 + \Sigma_{i=1}^{k} \mathsf{size}(t_i) & \text{if } t = f(t_1, \ldots, t_k) \text{ and } f \in \Omega_k \\ \mathsf{size}(t) & = & 1 & \text{if } t \in \mathbf{X} \cup \{\bullet\} \end{array}$$

In the following we define the size of a program. As mentioned in Section 2.1 we do not operate on the syntax of a program directly but on the corresponding control flow graph. The size of a program is then given as the sum of the number of nodes, the number of edges, and the sum of the sizes of terms of right-hand sides of assignments.

A non-ground term $t = t'\mathbf{x}$ containing occurrences of the variable $\mathbf{x}$ is then succinctly represented by the pair $(P, \mathbf{x})$ where P is an SLP for t'. Ground terms in T may be factorized differently for initialization-restricted or unrestricted programs. In the following, we first consider initialization-restricted programs, and subsequently unrestricted programs.

2.11.1 Polynomial Time Algorithms for IR Programs

For *initialization-restricted* programs, every ground term t possibly produced at run-time, can be uniquely factored into $t = t'r$ for $t' \in M_G$ and a ground term $r \in R$ occurring as a right-hand side in the program. Such a term t is represented by a pair (P, r) where P is an SLP for t'. We remark that the size of the term r is bounded by the size of the program.

In a succinct representation of a postcondition ϕ, every occurring term in $T \cup T'$ (recall that $T = M_G R$ and $T' = M_G \mathbf{X}$) is represented by such a pair where the different SLPs need not necessarily be disjoint but may share unknowns together with their definitions. The weakest precondition of a postcondition ϕ w.r.t. a non-ground assignment $\mathbf{x} := t\mathbf{y}$ is given as $\phi[t\mathbf{y}/\mathbf{x}]$. This means that $t\mathbf{y}$ must be substituted into each term $s\mathbf{x}$, $s \in M_G$ occurring in ϕ. If s or t equals $\bullet$, the substitution is trivial. So assume that neither s nor t equal $\bullet$.

Then by Theorem 2.11.1 an SLP P for st can be constructed from the SLPs for s and t by adding one fresh unknown together with its definition, so that the depth of the involved SLPs increases at most by one—even if the depth of the resulting term may be doubled. The resulting term of the substitution is then represented by the pair $(P, \mathbf{y})$.

Now consider a substitution $\phi[t/\mathbf{x}]$ for a ground term $t = t'r$ where $t' \in M_G$ and $r \in R$ is a ground term of some assignment. This means that t must be substituted into each term $s\mathbf{x}$ occurring in ϕ. If s equals $\bullet$, the substitution is trivial. Therefore, assume that s does not equal $\bullet$. Then by Theorem 2.11.1 an SLP P for st' can be determined from the SLPs for s and t' in polynomial time. The resulting term of the substitution is then represented by the pair (P, r). We thus have proven:

Lemma 2.11.2. *Consider a single equality $As_1 \doteq Bs_2$ or $As \doteq C$ where the terms $s_1, s_2, s \in T \cup T'$ are succinctly represented. Then a succinct representation of the weakest precondition of the equality w.r.t. an assignment $\mathbf{x} := t$ can be determined in time polynomial in the size of t. Furthermore, for each succinctly represented term at most one fresh unknown is introduced such that the depth of an SLP increases at most by one.* □

The weakest precondition of a postcondition $As\mathbf{x} \doteq Bt\mathbf{y}$ w.r.t. a procedure call $p()$ is given as $\phi' = \phi[As/A, Bt/B]$ if the weakest precondition of the generic postcondition $A\mathbf{x} \doteq B\mathbf{y}$ w.r.t. a procedure call $p()$ is given as ϕ. This case is similar to the case of (non-)ground program variable assignments. That means that, instead of a program variable two template variables are substituted. In order to obtain succinct representations for the terms in ϕ', we again can apply our techniques for computing succinct representations for the result of the substitution of terms.

Lemma 2.11.3. *Consider a single equality $As\mathbf{x} \doteq Bt\mathbf{y}$ (resp. $As \doteq Bt\mathbf{x}$, $As\mathbf{x} \doteq Bt$, or $As\mathbf{x} \doteq C$) where the occurring terms $s, t \in M_G$ are succinctly represented. Moreover, assume that each term of type $T \cup T'$ occurring in the weakest precondition ϕ of a generic postcondition $A\mathbf{x} \doteq B\mathbf{y}$ (resp. $A\mathbf{x} \doteq C$) w.r.t. a procedure call $p()$ is also succinctly represented. Then a succinct representation of the weakest precondition of the equality w.r.t. a procedure call $p()$ can be computed in time polynomial in the number of equalities in ϕ. Furthermore, for each succinctly represented term at most one fresh unknown is introduced such that the depth of an SLP increases at most by one.* □

From Lemmas 2.11.2 and 2.11.3 we conclude that the sizes and depths of occurring SLPs during the whole fixed point computation for determining

the weakest precondition transformers for procedures as well as the weakest precondition transformers for reachability, remains polynomial in the size of the program and the numbers of equalities occurring in preconditions. Accordingly, a polynomial time algorithm for inferring valid Herbrand equalities is obtained whenever we are given polynomial time algorithms for

- solving systems of ground equalities, as well as for
- approximate T-subsumption.

Consider a satisfiable equality of the form $As \doteq Bt$ where $s, t \in T$ are ground. Let $A = \bullet$, then the finite set of all solutions for B equals the set

$$\{\, uw \in C_\Omega \mid s = uvt \text{ and } u, v \in M_G \text{ and } v \text{ is irreducible and } wt = vt \,\}.$$

In the set above, each w equals v where some occurrences of $\bullet$ are substituted by t. That means, once the decomposition of s into uvt is known, then all solutions can be trivially derived. Still there exist $2^i - 1$ many solutions if $\bullet$ occurs i times in the term v. Let $u = [\![P]\!]$ be represented by some SLP P and $t = [\![Q]\!]r$ be represented by some SLP Q and $r \in R$. Then the set of all solutions for B is *succinctly represented* by the tuple

$$\langle P, v, Q, r \rangle. \tag{2.32}$$

Similarly, the finite set of all solutions for the template variable A is succinctly represented by a tuple of the form (2.32), if $B = \bullet$.

Theorem 2.11.4. *In the following consider only equalities of the form $As \doteq Bt$ where $s, t \in T$ are ground and succinctly represented.*

1. *It is decidable in polynomial time whether or not the equality $As \doteq Bt$ is satisfiable where A or B receives the value $\bullet$. Furthermore, if it is satisfiable, then a succinct representation of the form (2.32) of the set of all solutions for A (resp. B) can be determined in polynomial time.*
2. *It is decidable in polynomial time whether or not the conjunction of the two distinct equalities $As_1 \doteq Bt_1$ and $As_2 \doteq Bt_2$ is satisfiable where A or B receives the value $\bullet$. Furthermore, if it is satisfiable, then a succinct representation of the unique solution can be determined in polynomial time.*

Proof.

1. Let $A = \bullet$, i.e., we then consider $s \doteq Bt$. If the equality is satisfiable, then $s = t't$ for some $t' \in M_G$ must hold. Whether or not t is a suffix of s is decidable in polynomial time.

   Assume that the equality is satisfiable. Then each solution of B equals s where some occurrences of t are substituted by $\bullet$. Let $s = uvt$ for some $u, v \in M_G$ and v is an irreducible element in M_G. A succinct representation Q of the prefix u of s of length $\|s\| - (\|t\| + 1)$ can be determined in polynomial time. Likewise, the irreducible element v occurring in the unique factorization of s can be determined in polynomial time. Assume that t is succinctly represented by the tuple (P, r). Then the set of all solutions for B is succinctly represented by the tuple $\langle Q, v, P, r\rangle$ of the form (2.32), from which the assertion of this part follows.

2. Let $A = \bullet$, i.e., we then consider $s_1 \doteq Bt_1$ and $s_2 \doteq Bt_2$. If the conjunction of the two equalities is satisfiable, then $s_1 = tt_1$ and $s_2 = tt_2$ for some $t \in M_G$ must hold, i.e., $B = t$ is then a solution. From the succinctly represented term s_i a succinct representation of the prefix u_i of length $\|s_i\| - \|t_i\|$ and the suffix v_i of length $\|t_i\|$ can be determined in polynomial time for $i = 1, 2$. If $u_1 = u_2$ and $v_1 = t_1$ and $v_2 = t_2$ holds, then the conjunction is satisfiable and u_1 is a solution for B. This is decidable in polynomial time.

   According to Theorem 2.3.3, t is a unique solution, i.e., there exists no other solution $t' \neq t$. A similar argument holds for the case $B = \bullet$. □

Assume that we are given a conjunction of ground equalities arising from the analysis. Clearly, it allows to efficiently *test* any candidate templates whether or not they constitute a solution. In light of Theorem 2.11.4, the conjunction allows to *infer* a succinct representation of all valid equalities in polynomial time.

Theorem 2.11.5. *T-subsumption for equalities of the form $As \doteq C$ where $s \in T \cup T'$ are succinctly represented is decidable in polynomial time.*

Proof. Consider two equalities $As\mathbf{x} \doteq C$ and $At\mathbf{x} \doteq C$ with $s, t \in M_G$ (resp. $As \doteq C$ and $At \doteq C$ with $s, t \in T$). The conjunction of the two equalities is T-unsatisfiable, if $s \neq t$ holds which is decidable in polynomial time. Otherwise, if $s = t$ holds, then one equality is subsumed by the other. □

In the following we show that approximate T-subsumption of two-variable equalities is decidable in polynomial time, too. In order to do so we first extend the idea of succinctly represented terms in M_G to terms in the corresponding free group F_G. That means that definitions of an SLP representing a term in F_G are now either of the form $X \to YZ$ for suitable unknowns Y, Z or $X \to f$ where f is an irreducible term in F_G. The length $\|t\|$ of a term $t \in F_G$ can be determined in time linear in the size of the SLP representing t similar to any term $s \in M_G$. The balance $|t|$ of a term $t \in F_G$ which is represented by the SLP P can be determined in linear time in the size of P as follows:

$$\begin{array}{ll} |X_i|_P = |X_j|_P + |X_l|_P & (X_i \to X_j X_l) \in P \\ |X_i|_P = 0 & (X_i \to \bullet) \in P \\ |X_i|_P = 1 & (X_i \to f) \in P \text{ and } f \in M_G \setminus \{\bullet\} \\ |X_i|_P = -1 & (X_i \to f^-) \in P \text{ and } f \in M_G \setminus \{\bullet\} \end{array}$$

An SLP in Chomsky normal form of size k cannot produce a word larger than 2^k. Therefore, the balance of each word which it generates can be described by $k+1$ bits. For such numbers, basic operations as equality, addition and subtraction can be done in time linear in k—even if only single bit operations are considered as constant time.

Lemma 2.11.6. *Assume that all terms are succinctly represented and let F_G be the corresponding free group of M_G. Then the following tasks can be realized in polynomial time:*

1. *All tasks described in Theorem 2.11.1 can also be realized for terms in F_G.*
2. *Given a term $w \in F_G$, determine the term $w^{-1} \in F_G$.*
3. *Given two maximally canceled terms $u, v \in F_G$, determine $w = uv$ such that w is maximally canceled.*
4. *Given a term $w \in F_G$, determine the term w^r, $r \geq 1$.*

Proof. For the tasks described in Theorem 2.11.1 it is irrelevant from which algebraic structure an element f in a definition $X \to f$ comes. That means, it does not matter if $f \in M_G$ or $f \in F_G$ proving assertion 1.

Given an SLP P representing some term $w \in F_G$, the SLP P' representing the term w^{-1} can be constructed as follows. If the definition $X \to YZ$ is included in P, then let $X' \to Z'Y'$ be included in P'. Otherwise, if the definition $X \to f$, $f \in F_G$ is included in P, then let $X' \to f^{-1}$ be in P'. The size and the depth of P and P' are the same proving assertion 2.

Assume that the SLPs P and Q represent the terms u and v from F_G, respectively. By assertion 2, an SLP for u^{-1} can be determined in polynomial time. Furthermore, by Theorem 2.11.1, the length k of the longest common prefix of u^{-1} and v can be determined in polynomial time. Again by Theorem 2.11.1, an SLP Q for the prefix u' of u of length $\|u\| - k$ can be determined in polynomial time. Similarly, an SLP Q' for the suffix v' of v of length $\|v\| - k$ can be determined in polynomial time. Finally, an SLP for the term $w = u'v'$ can be determined in polynomial time. Since w is maximally canceled, this proves assertion 3.

The last assertion 4 can be proven as follows. The case where $r = 1$ is trivial. Therefore, assume that $r > 1$. Let the term w be represented by the SLP P with initial unknown X_0 and size s_P. The term w^{2^k}, $k \geq 1$ is then represented by the SLP Q_k with initial unknown N_{k+1} and the following definitions (for fresh unknowns N_k):

$$\begin{array}{lcll} N_1 & \to & X_0X_0 & \\ N_{i+1} & \to & N_iN_i & 1 \leq i \leq \log_2(k) \end{array}$$

Assume that the binary representation of r equals $b_{\log_2(r)} \dots b_0$ where b_0 is the least significant bit and let $j_1 < \cdots < j_n$ equal the list of indices j where $b_j = 1$. Then we introduce the SLP Q with initial unknown M_1 and the following fresh definitions:

$$\begin{array}{lcll} M_k & \to & N_{j_k}M_{k+1} & \text{for } 1 \leq k < n \\ M_n & \to & N_{j_n} & \end{array}$$

Thus, the SLP Q represents w^r. The size of Q is in $\mathcal{O}(\log_2(r) + s_P)$ from which the assertion follows. □

A term uv which is not maximally canceled, may only be constructed during checks of subsumption when two terms $u, v \in F_G$ are concatenated. According to Lemma 2.11.6, however, a maximally canceled term corresponding to uv can be determined in polynomial time. Therefore, in the following we assume that each succinctly represented term occurring during subsumption checks are maximally canceled.

Lemma 2.11.7. *Assume that all occurring terms are succinctly represented and maximally canceled. Then the assertion of Lemma 2.4.2 is decidable in polynomial time, i.e., the question whether for an equality of the form $AuA^{-1} \doteq Bu'B^{-1}$ with $u, u' \in F_G$ and $|u| = |u'| = 0$, it is decidable in polynomial time, whether it is trivial, is equivalent to an equality $As \doteq B$ or $A \doteq Bs$ for some $s \in M_G$, or is contradictory.*

Proof. Assume that u and u' are represented by the SLPs P and Q, respectively. The equality is trivial iff $[\![P]\!] = \bullet = [\![Q]\!]$ which can be checked in constant time since we assumed that succinctly represented terms are maximally canceled. If $[\![P]\!] = \bullet \neq [\![Q]\!]$, or $[\![P]\!] \neq \bullet = [\![Q]\!]$ holds, then the equality is contradictory. The latter can also be checked in constant time.

Otherwise, we proceed as follows. The length $n \leq \|u\|$ of the longest positive prefix of u can be determined similarly to the length $\|u\|$ of u, and thus can be determined in time linear in the size of P. Likewise, the length $m \leq \|u\|$ of the longest negative suffix of u can be determined in polynomial time, by first computing the inverse of u, i.e., u^{-1} and then determining the longest positive prefix of u^{-1}. We then proceed by determining SLPs for the prefix x of u of length n and the remaining suffix w of u of length $\|u\| - n$. From the SLP representing w we then derive SLPs for the prefix y of length $\|w\| - m$ and suffix of length m of w such that $u = xyz^{-1}$. This can be done in polynomial time.

Similarly, we determine succinct representations for the longest positive prefix x' of u', longest negative suffix z'^{-1} and y' such that $u' = x'y'z'^{-1}$.

Overall this means that the equivalent simplified conjunction $Ax \doteq Bx' \wedge y \doteq y' \wedge Az \doteq Bz'$ can be determined in polynomial time. Since $y, y' \in M_G$, their equality can be checked in polynomial time. If the conjunction is satisfiable then it is equivalent to a solved equality $As \doteq B$ or $A \doteq Bs$ which means that either $x = sx'$ and $z = sz'$, or $x' = sx$ and $z' = sz$ holds which can be checked in polynomial time. □

Lemma 2.11.8. *Assume that all occurring terms are succinctly represented and maximally canceled. Then the assertion of Theorem 2.4.3 is decidable in polynomial time, i.e., it is decidable in polynomial time whether the conjunction of the two equalities $AuA^{-1} \doteq Bu'B^{-1}$ and $AvA^{-1} \doteq Bv'B^{-1}$ with $u, u', v, v' \in F_G$ is equivalent to one solved equality, or to a single equality, or are contradictory.*

Proof. W.l.o.g. assume that $|u| \geq |v|$. If $|v| = 0$, then from Lemma 2.11.7 follows that $AvA^{-1} \doteq Bv'B^{-1}$ is either trivial, i.e., the conjunction of the two initial equalities is equivalent to $AuA^{-1} \doteq Bu'B^{-1}$, or is contradictory, i.e., the conjunction of the two initial equalities is equivalent to $AvA^{-1} \doteq Bv'B^{-1}$, or the equality is equivalent to one solved equality $As \doteq B$ (resp. $A \doteq Bs$). In the latter case either holds $u = su's^{-1}$ (resp. $u' = sus^{-1}$) and the conjunction of the two equalities is equivalent to $AvA^{-1} \doteq Bv'B^{-1}$ or the conjunction is contradictory. According to Theorem 2.11.1 and Lemma 2.11.6 the equality check $u = su's^{-1}$ (resp. $u' = sus^{-1}$) can be done in polynomial time—from which the assertion of this part follows.

Otherwise, if $|v| > 0$, then let $r = |u| \bmod |v|$ and we derive a third equality $AwA^{-1} \doteq Bw'B^{-1}$ such that $w = uv^{-r}$ and $w' = u'v'^{-r}$. According to Lemma 2.11.6 the terms w, w' can be determined in polynomial time. We then start allover by considering the two equalities $AvA^{-1} \doteq Bv'B^{-1}$ and $AwA^{-1} \doteq Bw'B^{-1}$ where $|v| \geq |w|$ holds. This algorithm is a generalization of *Euclid's algorithm*. Since Euclid's algorithm performs at most logarithmic many iterations [Mol08, pp. 21–22] and in each iteration we introduce logarithmic many new unknowns, the assertion of the theorem follows. □

Theorem 2.11.9. *For finite sets E, E' of equalities of the form $As \doteq Bt$ where $s, t \in T \cup T'$ are succinctly represented, it is decidable in polynomial time whether $\bigwedge E$ approximately T-subsumes $\bigwedge E'$ or not, if A or B equals $\bullet$.*

Proof. Consider equalities of the form $As \doteq Bt$ where $s, t \in T$ are ground terms. According to Theorem 2.11.4 T-subsumption is decidable in polynomial time.

Consider the three equalities $As_i\mathbf{x} \doteq Bt_i\mathbf{y}$, $i = 1, 2, 3$ and let w.l.o.g. $|s_1| \geq |s_2|, |s_3|$. We then derive the two equalities $AuA^{-1} \doteq Bu'B^{-1}$ and $AvA^{-1} \doteq Bv'B^{-1}$ where $u \equiv s_1 s_2^{-1}$, $u' \equiv t_1 t_2^{-1}$, $v \equiv s_1 s_3^{-1}$, and $v' \equiv t_1 t_3^{-1}$ are maximally canceled in polynomial time. According to Lemma 2.11.8 it is decidable in polynomial time whether the conjunction is unsatisfiable, or equivalent to one equality, i.e., equality $As_1\mathbf{x} \doteq Bt_1\mathbf{y}$ is then subsumed, or is equivalent to one solved equality. In the latter case from a fourth equality either follows the same solved equality and is therefore subsumed or is contradictory. A similar argument holds for equalities of the format $As \doteq Bt\mathbf{x}$ (resp. $At\mathbf{x} \doteq Bs$).

We conclude that T-subsumption for equalities of the same format is decidable in polynomial time. Since we consider only polynomial many different formats of equalities, the assertion of the theorem follows. □

Theorem 2.11.10. *Assume that all right-hand sides of assignments of an initialization-restricted program contain at most one variable. Then for every program point u and program variables $\mathbf{x}$ and $\mathbf{y}$, a succinct representation of the form (2.32) of the set of all valid two-variable Herbrand equalities between $\mathbf{x}$ and $\mathbf{y}$, can be determined in time polynomial in the size of the program.* □

2.11.2 Polynomial Time Algorithms for Unrestricted Programs

For unrestricted programs there need not exist a unique factorization for every possible run-time value. Only for large terms, i.e., terms in $L = M_G\bar{R}$, unique factorizations are possible. Accordingly, a large term $t = t'r$ where $t' \in M_G$

and $r \in \bar{R}$ is *succinctly represented* by a pair (P, r) where P is an SLP such that $[\![P]\!] = t'$. We remark that the size of the term r is polynomially bound by the size of the program and therefore can be represented explicitly.

For small terms, i.e., terms in S, on the other hand, we cannot hope for unique factorizations. Since the size of each small term is bound by the size of the program, each small term $s \in S$ is *succinctly represented* by a pair (P, s) where P is an SLP such that $[\![P]\!] = \bullet$.

Similar as for initialization-restricted programs, during the weakest precondition calculation, we assume that each occurring term is succinctly represented. Let us again consider the operation substitution. In order to obtain polynomial algorithms, we must ensure that substitution of succinctly represented terms is polynomial. Consider the non-ground terms $s\mathbf{x}, t\mathbf{y}$ where $s, t \in M_G$. Then the succinct representation of the resulting term $(s\mathbf{x})[t\mathbf{y}/\mathbf{x}]$ is determined in a similar way as for initialization-restricted programs, and therefore can be constructed in polynomial time. Now consider the terms $s\mathbf{x}, t$ where $s \in M_G$ and $t \in T$ is ground. Then the resulting term of the substitution $(s\mathbf{x})[t/\mathbf{x}]$ is given as st. If the term is large, then in order to succinctly represent st, the unique factorization must be determined in polynomial time.

Lemma 2.11.11. *Given succinctly represented terms s, t where $s \in M_G$ and $t \in T$. Then a succinct representation of $st \in T$ can be determined in time polynomial in the size of a maximal element in $\bar{R}$.*

Proof. First assume that $t \in L$ is *large*. This means that t is represented by a pair (Q, r) where Q is an SLP for some term $t' \in M_G$ and r is a term in $\bar{R}$. Then the unique factorization of st is given by $s'r$ where $s' = st'$—for which an SLP can be constructed from an SLP for s and Q by introducing one fresh unknown together with a single definition.

Finally, assume that the term t is *small*. Given an SLP P for the term s, our goal is to determine the unique factorization $st = s'r$ with $s' \in M_G$ and $r \in \bar{R}$. If $s = \bullet$, nothing must be done. Otherwise, assume that s is given as the factorization $s_1 \cdots s_k$. Then we consider the factorization $s_1 \cdots s_{k-1} s'_k$ where $s'_k = s_k[t/\bullet]$. This factorization equals the term st. If $k = 1$, we are done. If $k > 1$ and the term $s'_k = s_k[t/\bullet]$ is contained in the set $\bar{R}$ of minimally large terms, i.e., s'_k is not a small term, then we have found the unique factorization of st. Otherwise, we proceed by constructing $s'_{k-1} = s_{k-1}[s'_k/\bullet]$ and so on, until either we exhausted the factors of s or obtained the factorization $st = s's'_{k-h}$ where $s' = s_1 \cdots s_{k-h-1}$ and $s'_{k-h} = s_{k-h} \cdots s_k t \in \bar{R}$. Since the size of every term in $\bar{R}$ is bounded by the size of the input program, so is the number h. For every length $h \leq h' \leq k$, SLPs for the intermediately occurring prefixes of s

can be determined in time $\mathcal{O}(d)$ by Theorem 2.11.1, if d is the depth of the SLP for s. □

The previous Lemma 2.11.11 enables us to state the following two lemmas:

Lemma 2.11.12. *Consider a single equality $As_1 \doteq Bs_2$ or $As \doteq C$ where $s_1, s_2, s \in T \cup T'$ are succinctly represented. Then a succinct representation of the weakest precondition of the equality w.r.t. an assignment $\mathbf{x} := t$ can be determined in time polynomial in the size of t and in the size of a maximal element in $\bar{R}$.* □

Lemma 2.11.13. *Consider a single equality $As\mathbf{x} \doteq Bt\mathbf{y}$ (resp. $As \doteq Bt\mathbf{x}$, $As\mathbf{x} \doteq Bt$, or $As\mathbf{x} \doteq C$) where the occurring terms $s, t \in M_G$ are succinctly represented. Moreover, assume that each term of type $T \cup T'$ occurring in the weakest precondition ϕ of a generic postcondition $A\mathbf{x} \doteq B\mathbf{y}$ (resp. $A\mathbf{x} \doteq C$) w.r.t. a procedure call $p()$ is also succinctly represented. Then a succinct representation of the weakest precondition of the equality w.r.t. a procedure call $p()$ can be computed in time polynomial in the number of equalities in ϕ and in the size of a maximal element in $\bar{R}$.* □

The proofs of the lemmas are analogous to the proofs of Lemma 2.11.2 and 2.11.3 except that for the substitution we also need Lemma 2.11.11.

In order to compute solutions in polynomial time for the constraint systems **S** and **R**, T-subsumption for one-variable and approximate T-subsumption for two-variable equalities must be decidable in polynomial time.

Theorem 2.11.14. *For finite sets E, E' of equalities of the form $As \doteq C$ where $s \in T \cup T'$ are succinctly represented it is decidable in polynomial time whether $\bigwedge E$ T-subsumes $\bigwedge E'$ or not.*

Proof. Consider two distinct equalities $As\mathbf{x} \doteq C$ and $At\mathbf{x} \doteq C$. If the conjunction of them is satisfiable, then $s = wu$ and $t = wv$ for some $u, v, w \in M_G$ such that w is a longest common prefix of s, t and $u \neq v$ but $u\mathbf{x} = v\mathbf{x}$ must hold. According to Theorem 2.11.1 the longest common prefix of two succinctly represented terms can be determined in polynomial time. Similar representations for u, v can be determined in polynomial time, too. Assume that the sizes of the terms u, v are not bound by the maximal size of an element in $\bar{R}$, then the terms $u\mathbf{x}, v\mathbf{x}$ are large terms no matter what ground term the variable $\mathbf{x}$ is actually bound to. But then the terms u, v must have a common prefix which is a contradiction to the assumption that w is the longest common prefix of s, t if the conjunction is satisfiable. Therefore, assume that

the sizes of the terms u, v are bound by the maximal size of an element in $\bar{R}$. Then the most general unifier of $u\mathbf{x} = v\mathbf{x}$ can be determined in polynomial time. Assume the most general unifier maps $\mathbf{x}$ to the ground term $t' \in S$. Then the initial conjunction is equivalent to the conjunction of the equalities $As\mathbf{x} \doteq C$ and $Att' \doteq C$ where the latter equality does not contain any program variable. According to Lemma 2.11.11 a succinct representation of the term tt' can be determined in polynomial time. Overall, the equivalent conjunction can be determined in polynomial time.

For equalities which contain no program variable we have the following result. Consider two equalities $As \doteq C$ and $At \doteq C$ where $s, t \in T$ are ground. If $s = t$, then one equality subsumes the other. Otherwise, if $s \neq t$, then the conjunction of them is unsatisfiable. For succinctly represented terms such equality checks can be performed in polynomial time from which the assertion of the theorem follows. □

Theorem 2.11.15. *For finite sets E, E' of equalities of the form $As \doteq Bt$ where $s, t \in T \cup T'$ are succinctly represented, it is decidable in polynomial time whether $\bigwedge E$ approximately T-subsumes $\bigwedge E'$ or not, if A or B equals $\bullet$.*

Proof. **Ground equalities:** Let us first consider only equalities of the form $As \doteq Bt$ where $s, t \in T$ are ground. Then in the following we assume that each conjunction of equalities is not trivially unsatisfiable, i.e., there exist no two equalities of the form $As \doteq Bt$ and $As \doteq Bt'$ where $t \neq t'$, or vice versa, where the roles of A and B are interchanged. If two succinctly represented terms in T are equal or not, is decidable in polynomial time.

First consider equalities of the form $As \doteq Bt$ where $s, t \in L$ are large terms. The proof is analogous to the corresponding proofs for Theorem 2.11.4 where the set T is replaced with the set $L = M_G\bar{R}$, i.e., instead of the set R we rely on the set $\bar{R}$ of unique end marker terms.

Now consider three equalities $As_i \doteq Bt_i$ where $s_i \in L$ are large terms and $t_i \in S$ are small terms for $i = 1, 2, 3$. For the proof of this case, we require to extend the notion of substitution to a replacement of occurrences of arbitrary subterms. Consider arbitrary ranked terms $s, t, t' \in \mathcal{T}_\Omega(X \cup \{\bullet\})$. Then by $s[t/t']$ we denote the term where all occurrences of t' in s are replaced by the term t. Formally, if s does not contain the subterm t', then $s[t/t'] = s$. Otherwise, if s contains the subterm t', then let $s = s't'$ such that $s' \in C_\Omega$ does not contain the subterm t'. Then $s[t/t'] = s't$.

We then proceed as follows. Assume that there exist $i, j \in [1, 3]$ such that t_i does not occur in s_j. If $i = j$, then the single equality $As_i \doteq Bt_i$ is not

satisfiable. Therefore assume now that $i \neq j$. If the conjunction of the three equalities is satisfiable, then the solution for B must not contain occurrences of t_i, i.e., $u = s_i[\bullet/t_i]$ is the only possible solution for B. If all three equalities are satisfied by this solution, then the first two would already have $B = u$ as their unique solution. Accordingly, the third equality is subsumed. Whether or not $B = u$ is a solution can be decided in polynomial time.

In the following we therefore assume that for each $i, j \in [1, 3]$ the term t_i occurs at least once in the term s_j. We define an equivalence relation of terms as follows. Let $\#$ denote a fresh symbol and let $s, s', t, t' \in T$. If t, t' are incomparable, i.e., there exists no $u \in M_G$ such that $t = ut'$ or $t' = ut$, then the terms s, s' are equivalent modulo the terms t, t' if $s[\#/t, \#/t'] = s'[\#/t, \#/t']$ holds. Otherwise, if there exists a $u \in M_G$ such that $t = ut'$, then the terms s, s' are equivalent modulo the terms t, t' if $(s[\#/t])[\#/t'] = (s'[\#/t])[\#/t']$ holds. The case where $t' = ut$ holds is similar. For all three cases we can decide which term to substitute first by comparing the size of both terms t, t'. That means, if $t = ut'$ (resp. $t' = ut$) holds, then $\mathsf{size}(t) > \mathsf{size}(t')$ (resp. $\mathsf{size}(t') > \mathsf{size}(t)$) must hold, too. In case the terms are incomparable it does not matter in which order we substitute the terms. Assume $\mathsf{size}(t) \geq \mathsf{size}(t')$, then the terms s, s' are equivalent modulo the terms t, t' if $(s[\#/t])[\#/t'] = (s'[\#/t])[\#/t']$ holds, which we denote by $(s = s') \bmod t, t'$. We extend the equivalence relation as follows. Let $t'' \in T$ and assume that $\mathsf{size}(t) \geq \mathsf{size}(t') \geq \mathsf{size}(t'')$, then s and s' are equivalent modulo the terms t, t', t'' if $((s[\#/t])[\#/t'])[\#/t''] = ((s'[\#/t])[\#/t'])[\#/t'']$ holds which we denote by $(s = s') \bmod t, t', t''$. We observe that if the conjunction $As_1 \doteq Bt_1 \wedge As_2 \doteq Bt_2$ is satisfiable, then s_1, s_2 differ only in some occurrences of t_1, t_2. That means that $(s_1 = s_2) \bmod t_1, t_2$ must hold. A similar argument holds for the conjunction $As_1 \doteq Bt_1 \wedge As_3 \doteq Bt_3$ and for the conjunction $As_2 \doteq Bt_2 \wedge As_3 \doteq Bt_3$. Observe that the other direction does not necessarily hold, i.e., if the conjunction is not satisfiable, then $(s_1 \neq s_2) \bmod t_1, t_2$ need not hold. For example, consider the conjunction $Af(a, b) \doteq Ba \wedge Af(b, a) \doteq Bb$ which is not satisfiable but $f(a, b)[\#/a, \#/b] = f(\#, \#) = f(b, a)[\#/a, \#/b]$ holds. However, we claim that if

$$(s_1 = s_2) \bmod t_1, t_2 \tag{2.33}$$
$$(s_1 = s_3) \bmod t_1, t_3 \tag{2.34}$$
$$(s_2 = s_3) \bmod t_2, t_3 \tag{2.35}$$
$$(s_1 = s_2 = s_3) \bmod t_1, t_2, t_3 \tag{2.36}$$

holds, then the conjunction $As_1 \doteq Bt_1 \wedge As_2 \doteq Bt_2 \wedge As_3 \doteq Bt_3$ is satisfiable.

Since for $A = \bullet$, the two first equalities uniquely determine the solution for B, we conclude that the third equality is subsumed. Our claim is proved as follows.

In the following we denote by $s|_p = t$ that a term $s \in T$ contains at position p the subterm $t \in T$. From (2.36) follows that all three terms s_1, s_2, s_3 share a common pattern u' which is obtained by successively replacing all subterms t_1, t_2, t_3 with $\#$, where we proceed from the larger to the smaller terms. From u', we then construct a solution u for B by replacing the occurrences of $\#$ in u' with terms t_1, t_2, t_3 or $\bullet$. Let p be any position of a leaf $\#$ in u'.

$$\text{If } s_1|_p = s_2|_p \text{ then let } u|_p = s_1|_p \tag{2.37}$$

$$\text{If } s_1|_p \neq s_2|_p \text{ then let } u|_p = \bullet \tag{2.38}$$

We claim that the resulting term u is indeed a solution for B, which satisfies all three equalities. If $u|_p = t_1$ then according to (2.37) $s_1|_p = s_2|_p = t_1$ and from (2.35) follows that $s_3|_p = t_1$. If $u|_p = t_2$ then according to (2.37) $s_1|_p = s_2|_p = t_2$ and from (2.34) follows that $s_3|_p = t_2$. Otherwise, assume that $u|_p = t_3$. Then according to (2.37) $s_1|_p = s_2|_p = t_3$. If $s_3|_p = t_1$, then (2.35) implies that $s_2|_p = t_1$ which is a contradiction. Similarly, if $s_3|_p = t_2$, then (2.34) implies that $s_1|_p = t_2$ which again is a contradiction. Therefore, $s_3|_p = t_3$ must hold. In total we have, if $u|_p = t_i$, then $s_1|_p = s_2|_p = s_3|_p = t_i$ for $i = 1, 2, 3$. Now consider the case where $u|_p = \bullet$. Assume that $s_1|_p = t_2$, then from (2.33) and (2.38) follows $s_2|_p = t_1$. However, then from (2.34) follows that $s_3|_p = t_2$ and from (2.35) follows that $s_3|_p = t_1$ which is a contradiction. Hence, $s_1|_p = t_1$ and $s_2|_p = t_2$ must hold. A similar argument holds for $s_3|_p = t_3$ from which we conclude that $s_i = ut_i$ for $i = 1, 2, 3$. Therefore, $B = u$ is indeed a solution satisfying all three equalities. This complete the proof of our claim.

What remains to prove is that the equality checks $(s_i = s_j) \bmod t_i, t_j$ and $(s_i = s_j) \bmod t_1, t_2, t_3$ can be done in polynomial time. For that we must show that from an arbitrary succinctly represented large term, a succinctly represented and uniquely factorized term can be derived where certain small terms are substituted by a fresh symbol. We explain the idea for the test $(s_1 = s_2) \bmod t_1, t_2$. W.l.o.g. let $\mathsf{size}(t_1) \geq \mathsf{size}(t_2)$ and $\sigma = [\#/t_1][\#/t_2]$. Let $G' = \{ g\sigma \mid g \in G \}$ and $R' = \{ r\sigma \mid r \in R \}$. We extend the factorization of terms in $T = M_G R$ to terms in $T' = M_{G'} R'$. In Section 2.6 we have partitioned the set of terms T into non-uniquely factorizable small terms S and uniquely factorizable large terms L, i.e., $T = M_G R = S \uplus L$. We proceed along the same line for T' which we partition into $\#$-small terms S' which are non-uniquely

factorizable, and into #-large terms L' which are uniquely factorizable. The set S' equals then the set $(G' \cup R')^*$ where $*$ is the subterm closure, and the set L' equals the set $M_{G'}R' \setminus S'$. We call a term minimally #-large if it is a minimal term in L'. The (finite) set of all minimally #-large terms is denoted by $\bar{R}'$. Then every #-large term $s' \in L'$ can be uniquely factored into $s' = u'r'$ where $u' \in M_{G'}$ and a term $r' \in \bar{R}'$ which is minimally #-large. If one of the terms $s_1\sigma$ or $s_2\sigma$ is not #-large, then the size of that term is polynomial. Therefore, the equality test can be realized in polynomial time as well. Accordingly assume that the terms $s_1\sigma$ and $s_2\sigma$ are both #-large. In this case, our goal is to determine from the succinct representations of the factorizations of s_1, s_2, succinct representations for the factorizations of $s_1\sigma, s_2\sigma$ which then can be compared in polynomial time. For that, consider a factorization $s = u_1 \cdots u_k r$ of a large term $s \in T$ into irreducible factors $u_i \in M_G$ and a minimally large term $r \in \bar{R}$, and assume that $s' = s\sigma$ is #-large. Then there is a maximal index j such that $r'_j = (u_j \cdots u_k r)\sigma$ is #-large. This index can be found in polynomial time. Moreover, r'_j can then be uniquely factored in polynomial time into $r'_j = u'r'$ for a minimally #-large term $r' \in \bar{R}'$ and $u' \in M_{G'}$. Then the unique factorization of s' is given by:

$$s' = v'_1 \cdots v'_{j-1} u' r'$$

where for each i, v'_i is a factorization of $u_i\sigma$ into irreducible factors in $M_{G'}$. Note that the *lengths* of the factorizations v'_i are bounded by the sizes of the corresponding factors and thus of the sizes of right-hand sides of the input program. Therefore these factorizations can be obtained in polynomial time as well. These factorizations then allow us to construct from an SLP for $u_1 \cdots u_{j-1}$, an SLP for $v'_1 \cdots v'_{j-1} u'$. Altogether, we obtain a succinct representation for s' from a succinct representation of s in polynomial time from which the assertion of this part follows.

A similar argument holds for equalities of the form $As \doteq Bt$ where $s \in S$ is small and $t \in L$ is large.

Non-ground Equalities: Now we consider equalities which contain at least one program variable. We first prove that L-subsumption for finite conjunctions of equalities of the same format is decidable in polynomial time.

For equalities of the formats $[F_{\mathbf{x},\mathbf{y}}], [F_{\mathbf{x},\cdot}], [F_{\cdot,\mathbf{x}}]$ the proofs are analogous to the corresponding proofs of Theorem 2.11.9 where the set T is replaced with the set $L = M_G\bar{R}$, i.e., instead of the set R we rely on the set $\bar{R}$ of unique end marker terms.

Now consider two equalities $As \doteq Bt\mathbf{x}$ and $As \doteq Bt'\mathbf{x}$ where $s \in S$ is a small term, i.e., equalities of the format $[F_{s,\mathbf{x}}]$. Then the first equality L-subsumes

the second equality, if $t = t'$ holds. This is decidable in polynomial time. Otherwise, the conjunction is L-unsatisfiable. A similar argument holds for two equalities of the format $[F_{\mathbf{x},s}]$.

We conclude that L-subsumption for equalities of the same format is decidable in polynomial time. In order to decide T-subsumption between conjunctions of sets E, E' of equalities of the same format, for each small substitution σ, L-subsumption between $E\sigma$ and $E'\sigma$ has to be decided. Since there exist at most polynomial many small substitutions and formats of equalities, we conclude that approximate T-subsumption is decidable in polynomial time. □

We showed that solutions to the constraint systems **S** and **R** can be determined in polynomial time. For *initialization-restricted* programs we also showed that a succinct representation of all solutions can be determined in polynomial time. Whereas for *unrestricted* programs we show that given a candidate solution for the template variables A and B where at least one equals $\bullet$, it is decidable in polynomial time whether or not the solution holds.

Theorem 2.11.16. *Given a term $u \in C_\Omega$ and an equality $As \doteq Bt$ where $s, t \in T$ are ground and succinctly represented. Then it is decidable in time polynomial in the size of the term u and in the size of a maximal term in S, whether or not $A = u$ and $B = \bullet$, or vice versa, $A = \bullet$ and $B = u$ is a solution for the equality.*

Proof. Let us first consider the case for $A = u$ and $B = \bullet$, i.e., decide if $us = t$ holds or not.

Assume that $s, t \in L$ are large terms. If $u \in M_G$, i.e., all subterms of u are small, then u must be a prefix of t, and s must be a suffix of t, i.e., $us = t$ must hold. This is decidable in time polynomial in the size of u and polynomial in the sizes and lengths of the SLPs representing s, t. Otherwise, if u contains large terms as subterms, i.e., $u \in C_\Omega \setminus M_G$. Then $u = vw$ for some $v \in M_G$ and some irreducible element $w \in C_\Omega \setminus M_G$ must hold. Furthermore, $t = vw's$ for some irreducible element $w' \in M_G$ such that w equals w' where some occurrences of $\bullet$ are substituted by s must hold. This is decidable in time polynomial in the size of u and polynomial in the lengths of the SLPs representing s, t.

Now consider the case where $s \in S$ is small and $t \in L$ is large. Then $us = t$ is decidable in time polynomial in the size of u and in the size of s which is bound by the size of the program.

Otherwise, if $s \in L$ is large and $t \in S$ is small, then the equality is not satisfiable.

Now assume that both $s, t \in S$ are small. Whether or not us is a small term and if us equals t is decidable in polynomial time.

Furthermore, verifying if $A = \bullet$ and $B = u$ is a solution for the equality is similar from which the assertion of this theorem follows. □

Finally this enables us to state our main result for unrestricted programs and one- or two-variable equalities:

Theorem 2.11.17. *Assume that p is a program where all right-hand sides of assignments contain at most one variable. Then for every program point u of p and every equality of the form $\mathbf{x} \doteq t$ where $t \in T \cup T'$, it can be verified in time polynomial in the size of the program as well as the size of t whether or not the equality is an invariant.* □

Recall that for *initialization-restricted* programs, each possible run-time value can be uniquely factorized. This property enabled us to derive in polynomial time from a ground equality $As \doteq Bt$ where $s, t \in T$ a succinct representation of all possible solutions for the template variables A and B where at least one equals $\bullet$. Consider the case where $A = \bullet$ and assume that $s = uvt$ for some $u, v \in M_G$ where v is an irreducible element. Then each solution for B has u as a prefix—which might be exponentially large. That means, that the solutions only differ in the very last factor which can be derived from the element v. Accordingly, we were able to provide a succinct characterization of *all* solutions. The situation is more complicated for *unrestricted* programs. For these, only weaker forms of factorization are available. Thus, substitutions of right-hand sides may still result in terms which are still *small* and therefore cannot be uniquely factorized.

Example 2.11.2. Assume that $a, b \in S$ are small terms and $r \in \bar{R}$ is a minimally large term. Then consider the uniquely factorized equalities

$$A\ f(\bullet,\bullet)\ g(a,h(b,\bullet,a),h(b,\bullet,b))\ r \doteq B\ a$$
$$A\ f(g(a,\bullet,\bullet),g(b,\bullet,\bullet))\ h(b,\bullet,b)\ r \doteq B\ b$$

Since a and b are small terms and the template variable A is applied to large terms, B cannot equal $\bullet$ in any possible solution. Therefore, now assume that $A = \bullet$. Then the unique solution for B, satisfying both equalities, equals

$$f(g(a,h(b,r,\bullet),h(b,r,b)),g(\bullet,h(b,r,\bullet),h(b,r,b)))$$

Thus, all three factors from the original equality are collapsed into a single irreducible term for B. This irreducible term contains the large term $h(b,r,b)$ as a subterm and is contained in $C_\Omega \setminus M_G$. ■

For initialization-restricted programs we observed that a solution is not necessarily in M_G but might very well also be in $C_\Omega \setminus M_G$. This is also the case for unrestricted programs. However, in contrast to *initialization-restricted* programs, we have for *unrestricted* programs that a solution need not reflect the factorizations of terms at all of initial equalities. The factorization of terms, however, was the basis of our compression scheme via SLPs. Accordingly, it remains unclear how to derive compressed representations of solutions in polynomial time.

2.11.3 Complexity Results for Verifying Multi-variable Equalities

Let us now consider a multi-variable invariant candidate such as $\mathbf{x} \doteq f(g\mathbf{y}, \mathbf{z})$. In this case, the right-hand side $f(g\mathbf{y}, \mathbf{z}) = t[\mathbf{y}, \mathbf{z}]$ where t is the (multi-variable) pattern $t = f(g\bullet_1, \bullet_2)$ for distinct variables $\bullet_1, \bullet_2$. Now consider a generic postcondition $\mathbf{x}' \doteq f(gA\mathbf{y}', B\mathbf{z}')$ which might occur during the proof that the given equality indeed is an invariant at some program point. In contrast to the pattern, the terms which may be substituted into one of the program variables or the template variables A, B of the right-hand side during the fixed point iteration may grow exponentially deep and therefore must be succinctly represented. Now consider a term which is substituted into the left-hand side. For this term, the root must be deconstructed according to the constructors occurring in t. This deconstruction can also be realized for succinctly represented terms in polynomial time.

For the multi-variable case we observe that during the weakest precondition computation we obtain for a postcondition a conjunction possibly containing one-, two-, and multi-variable equalities. A conjunction of two multi-variable equalities which coincide in the left-hand side and the pattern of the right-hand side is equivalent to a conjunction of one of them and polynomial many one- and two-variable equalities. Such an equivalent conjunction can be determined in time polynomial in the size of the invariant candidate. Since for conjunctions of one- and two-variable equalities approximate T-subsumption is decidable in polynomial time, approximate T-subsumption is also decidable in polynomial time for conjunctions containing multi-variable equalities, i.e., we have proven the following lemma:

Lemma 2.11.18. *For finite sets E, E' of one-, two-, and multi-variable equalities where each term in $T \cup T'$ is succinctly represented, it is decidable in polynomial time whether $\bigwedge E$ approximately T-subsumes $\bigwedge E'$ or not.* □

We note that from a single invariant candidate $\mathbf{x} \doteq t$ where $t \in \mathcal{T}_\Omega(\mathbf{X})$,

exponentially many generic multi-variable postconditions can be derived, i.e., we have exponentially many different formats of multi-variable equalities. Still, we have:

Theorem 2.11.19. *Assume that p is a program where all right-hand sides of assignments contain at most one variable. Then for every program point u of p and every multi-variable Herbrand equality $\mathbf{x} \doteq t$ where $t \in \mathcal{T}_\Omega(\mathbf{X})$ has at most k variables, it can be verified in time polynomial in the size of the program as well as the size of t, and exponential only in k whether or not the equality is an invariant.* □

2.12 Summary

In this chapter we provided an analysis which infers all interprocedurally valid Herbrand equalities for programs where all assignments are taken into account whose right-hand sides depend on at most one variable. The novel analysis is based on three main ideas. First, we restricted general satisfiability, subsumption, and equivalence to satisfiability, subsumption, and equivalence w.r.t. a set of values subsuming all possible run-time values of a given program. Together with our factorization theorem, this allowed us to apply the monoidal methods from [GT07] to effectively infer all interprocedurally valid two-variable Herbrand equalities, at least for programs, which we called *initialization-restricted*. In the second step, we abandoned this restriction by introducing the extra distinction between *large* values (which can be uniquely factored) and *small* values (of which there are only finitely many). Finally, we showed how all general Herbrand equalities can be inferred.

In Section 2.11 we analyzed the complexity of the presented analysis. For that we first observed that the factorization of terms already leads to compressed terms by sharing isomorphic subterms. However, the depth of terms may grow exponentially. Therefore, we followed the idea from [GT07] and introduced straight-line programs in order to also compress the depth of terms. By that we were able to provide a polynomial time algorithm which infers compact representations of all two-variable Herbrand equalities for *initialization-restricted* programs. For *unrestricted* programs, we were at least able to verify in polynomial time whether or not a given equality is an invariant at a given program point. The reason why we cannot infer all equalities in polynomial time for unrestricted programs as for initialization-restricted programs is that the succinct representation of terms relies on the factorization of terms. For unrestricted programs we have that a solution might not

reflect the factorization of terms at all. Hence it remains unclear how to derive compressed representations of solutions in polynomial time. The algorithm could also be extended to *general* Herbrand equalities (possibly containing more than one variable). Here we have the result that an invariant candidate $\mathbf{x} \doteq t$ where t contains at most k differing variables can be verified in time polynomial in the size of the program, and the term t, and exponentially only in k.

3 Terminating Local Solvers

It is well known that static analysis of run-time properties of programs by means of abstract interpretation can be compiled into systems of equations over complete lattices [CC77a]. Thereby, various interesting properties require complete lattices which may have infinite strictly ascending or descending chains [GH06; Che+10; Bag+05]. In order to determine a (post-) solution of a system of equations over such lattices, Cousot and Cousot propose to perform a first phase of iteration using a *widening* operator to obtain a post-solution which later may be improved by a second phase of iteration using a *narrowing* operator. This strict arrangement into separate phases, though, has the disadvantage that precision unnecessarily may be given up which later is difficult to recover. It has been observed that widening and narrowing need not be organized into separate phases [ASV13; ASV16; Ama+16]. Instead various algorithms are proposed which *intertwine* widening with narrowing in order to compute a (reasonably small) post-fixed point of the given system of equations. The idea there is to combine widening with narrowing into a single operator and then to iterate according to some fixed ordering over the variables of the system. Still, monotonicity of all right-hand sides is required for the resulting algorithms to be terminating [ASV13; Ama+16].

Non-monotone right-hand sides, however, are introduced by interprocedural analysis in the style of [ASV12] when partial tabulation of summary functions is used. In order to see this, consider an abstract lattice $\mathbb{D}$ of possible program invariants. Then the abstract effect of a procedure call can be formalized as a transformation $f^\sharp$ from $\mathbb{D} \to \mathbb{D}$. For rich lattices $\mathbb{D}$ such transformations may be difficult to represent and compute with. As a remedy, each single program procedure may be decomposed into a system of equations with equation variables—one for each possible argument—where each such variable now receives values from $\mathbb{D}$ only. As a result, the difficulty of dealing with elements of $\mathbb{D} \to \mathbb{D}$ is replaced with the difficulty of dealing with systems of equations which are infinite when $\mathbb{D}$ is infinite. Moreover, composition of abstract functions is translated into *indirect addressing* of variables (the outcome of the analysis for one function call determines for which argument another function is queried) which implies non-monotonicity [FS96]. Thus,

termination of interprocedural analysis by means of the solvers from [ASV13; Ama+16] cannot be guaranteed. Interestingly, the *local* solver SLR_3 [Ama+16] terminates in many practical cases. Nontermination, though, may arise in two flavors:

- infinitely many variables may be encountered, i.e., some procedure may be analyzed for an ever growing number of calling contexts;
- the algorithm may for some variable switch infinitely often from a narrowing iteration back to a widening iteration.

From a conceptual view, the situation still is unsatisfactory: any solver used as a fixed point engine within a static analysis tool should reliably terminate under reasonable assumptions. In this section of the thesis, we therefore re-examine interprocedural analysis by means of local solvers. First, we extend an ordinary local solver to a two-phase solver which performs widening and subsequently narrowing. The novel point is that both iterations are performed in a demand-driven way so that also during the narrowing phase fresh variables may be encountered for which no sound over-approximation has yet been computed.

In order to enhance precision of this demand-driven two-phase solver, we then design a new local solver which intertwines the two phases. In contrast to the solvers in [ASV13; Ama+16], however, we can no longer rely on a fixed combination of a widening and a narrowing operator, but must enhance the solver with extra logic to decide when to apply which operator. For both solvers, we prove that they terminate whenever only finitely many variables are encountered, irrespective whether the abstract system is monotone or not. Both solvers are guaranteed to return (partial) post-solutions of the abstract system of equations only if all right-hand sides are monotone. Therefore, we make clear in which sense the computed results are nonetheless sound even in the non-monotone case. For that, we provide a sufficient condition for an abstract variable assignment to be a sound description of a concrete system—given only a (possibly non-monotone) abstract system of equations. This sufficient condition is formulated by means of the *lower monotonization* of the abstract system. Also, we elaborate for partial solutions in which sense the domain of the returned variable assignment provides sound information. Here, the formalization of purity of functions based on computation trees and variable dependencies plays a crucial role. The concept of lower monotonization requires a Galois connection to relate concrete sets of states and abstract

values. Later, we lift this restriction enabling us to verify that the novel local solvers work correctly also in a more general setting.

Finally, we prove that interprocedural analysis in the style of [CC77c; ASV12] with partial tabulation using the novel local solvers terminates for all non-recursive programs and every complete lattice with or without infinite strictly ascending or descending chains.

3.1 Basics on Abstract Interpretation

In the following we recapitulate the basics of abstract interpretation as introduced by Cousot and Cousot [CC77a; CC92]. Assume that the concrete semantics of a system is described by a system of equations

$$x = f_x, \qquad x \in X \tag{3.1}$$

where X is a set of variables taking values in some power set lattice $(\mathbb{C}, \subseteq, \cup)$ where $\mathbb{C} = 2^Q$ for some set Q of concrete program states, and for each $x \in X$, $f_x : (X \to \mathbb{C}) \to \mathbb{C}$ is the defining right-hand side of x. For the concrete system of equations, we assume that all right-hand sides $f_x, x \in X$, are *monotone*. Accordingly, this system of equations has a unique least solution σ which can be obtained as the least upper bound of all assignments σ_τ, τ an ordinal. The assignments $\sigma_\tau : X \to \mathbb{C}$ are defined as follows. If $\tau = 0$, then $\sigma_\tau\, x = \bot$ for all $x \in X$. If $\tau = \tau' + 1$ is a successor ordinal, then $\sigma_\tau\, x = f_x\, \sigma_{\tau'}$, and if τ is a limit ordinal, then $\sigma_\tau\, x = \bigcup \{ \sigma_{\tau'}\, x \mid \tau' < \tau \}$. A corresponding *abstract* system of equations

$$y = f^\sharp_y, \qquad y \in Y \tag{3.2}$$

specifies an analysis of the concrete system of equations. Here, Y is a set of *abstract* variables which may not necessarily be in one-to-one correspondence to the concrete variables in the set X. The variables in Y take values in some complete lattice $(\mathbb{D}, \sqsubseteq, \sqcup)$ of abstract values and for every abstract variable $y \in Y$, $f^\sharp_y : (Y \to \mathbb{D}) \to \mathbb{D}$ is the abstract defining right-hand side of y. The elements $d \in \mathbb{D}$ are meant to represent invariants, i.e., properties of program states. It is for simplicity that we assume the set $\mathbb{D}$ of all possible invariants to form a complete lattice, as any partial order can be embedded into a complete lattice so that all existing least upper and greatest lower bounds are preserved [Mac37].

In order to relate concrete sets of program states with abstract values, let us assume for the moment that there is a Galois connection between $\mathbb{C}$ and

$\mathbb{D}$, i.e., there are monotone functions $\alpha : \mathbb{C} \to \mathbb{D}$, $\gamma : \mathbb{D} \to \mathbb{C}$ such that for all $c \in \mathbb{C}$ and $d \in \mathbb{D}$, $\alpha(c) \sqsubseteq d$ iff $c \subseteq \gamma(d)$. A similar assumption has already been made in [CC77a]. In [CC92] this assumption is weakened to frameworks where no best abstract descriptions of concrete sets of program states can be obtained via an abstraction function α. In Section 3.7, we therefore later will consider this more general situation.

In the presence or absence of a Galois connection, we assume that there is a *description relation* $\mathcal{R} \subseteq X \times Y$ between the sets of concrete and abstract variables. The description relation $\mathcal{R}$ between variables is lifted to a description relation $\mathcal{R}^*$ between assignments $\sigma : X \to \mathbb{C}$ and $\sigma^\sharp : Y \to \mathbb{D}$ such that

$$\sigma \; \mathcal{R}^* \; \sigma^\sharp \iff \sigma(x) \subseteq \gamma(\sigma^\sharp(y))$$

for all $x \in X$ and $y \in Y$ whenever $x \; \mathcal{R} \; y$ holds. Following [CC92], we do not assume that the right-hand sides of the abstract equation system are necessarily monotone. For a sound analysis, we only assume that all right-hand sides respect the description relation, i.e., that for all $x \in X$ and $y \in Y$ with $x \; \mathcal{R} \; y$,

$$f_x \, \sigma \subseteq \gamma(f^\sharp_y \, \sigma^\sharp) \tag{3.3}$$

whenever $\sigma \; \mathcal{R}^* \; \sigma^\sharp$ holds.

Our key concept for proving soundness of abstract variable assignments w.r.t. the concrete system of equations is the notion of the *lower monotonization* of the abstract system. For every function $f^\sharp : (Y \to \mathbb{D}) \to \mathbb{D}$ we consider the function

$$\underline{f^\sharp} \, \sigma = \bigsqcap \left\{ f^\sharp \, \sigma' \;\middle|\; \sigma \sqsubseteq \sigma' \right\} \tag{3.4}$$

which we call *lower monotonization* of $f^\sharp$. By definition, we have:

Lemma 3.1.1. *For every function $f^\sharp : (Y \to \mathbb{D}) \to \mathbb{D}$ the following holds:*

1. *$\underline{f^\sharp}$ is monotone;*
2. *$\underline{f^\sharp} \, \sigma^\sharp \sqsubseteq f^\sharp \, \sigma^\sharp$ for all $\sigma^\sharp$;*
3. *$\underline{f^\sharp} = f^\sharp$ whenever $f^\sharp$ is monotone.* □

The lower monotonization of the abstract system (3.2) then is defined as the system

$$y = \underline{f^\sharp}_y, \qquad y \in Y \tag{3.5}$$

Since all right-hand sides of (3.5) are monotone, this system has a least solution.

Example 3.1.1. Consider the single equation

$$y = \textbf{if } y = 0 \textbf{ then } 1 \textbf{ else } 0$$

over the complete lattice $\mathbb{N}^\infty$ of non-negative integers equipped with the natural ordering and extended by an infimum element. This system is not monotone. Its lower monotonization is given by $y = 0$. ■

Lemma 3.1.2. *Assume that σ is the least solution of the concrete system* (3.1)*. Then $\sigma \; \mathcal{R}^* \; \sigma^\sharp$ for every post-solution $\sigma^\sharp$ of the lower monotonization* (3.5)*.*

Proof. For every ordinal τ, let σ_τ denote the τth approximation of the least solution of the concrete system. Assume that $\sigma^\sharp$ is a post-solution of the lower monotonization of the abstract system, i.e.,

$$\underline{f}^\sharp_y \, \sigma^\sharp \sqsubseteq \sigma^\sharp \, y$$

holds for all $y \in Y$. By ordinal induction, we prove that $\sigma_\tau \; \mathcal{R}^* \; \sigma^\sharp$. The claim clearly holds for $\tau = 0$.

First assume that $\tau = \tau' + 1$ is a successor ordinal, and that the claim holds for τ', i.e., $\sigma_{\tau'} \; \mathcal{R}^* \; \sigma^\sharp$. Accordingly, $\sigma_{\tau'} \; \mathcal{R}^* \; \sigma'$ holds for all $\sigma' \sqsupseteq \sigma^\sharp$. Consider any pair of variables x, y with $x \; \mathcal{R} \; y$. Then $\sigma_\tau \, x = f_x \sigma_{\tau'} \sqsubseteq y(f^\sharp_y \, \sigma')$ for all $\sigma' \sqsupseteq \sigma^\sharp$. Accordingly, $\alpha(\sigma_\tau \, x) \sqsubseteq f^\sharp_y \, \sigma'$ for all $\sigma' \sqsupseteq \sigma^\sharp$, and therefore,

$$\alpha(\sigma_\tau \, x) \sqsubseteq \bigsqcap \left\{ f^\sharp_y \, \sigma' \;\middle|\; \sigma' \sqsupseteq \sigma^\sharp \right\} = \underline{f}^\sharp_y \, \sigma^\sharp \sqsubseteq \sigma^\sharp \, y$$

since $\sigma^\sharp$ is a post-solution. From that, the claim follows for the ordinal τ.

Now assume that τ is a limit ordinal, and that the claim holds for all ordinals $\tau' < \tau$. Again consider any pair of variables x, y with $x \; \mathcal{R} \; y$. Then

$$\sigma_\tau \, x = \bigcup \left\{ \sigma_{\tau'} \, x \;\middle|\; \tau' < \tau \right\} \sqsubseteq \bigcup \left\{ y(\sigma^\sharp \, y) \;\middle|\; \tau' < \tau \right\} = y(\sigma^\sharp \, y)$$

and the claim also follows for the limit ordinal τ. □

Note that for the proof of Lemma 3.1.2 it was crucial to assume that there is a Galois connection between the concrete and abstract domains. From Lemma 3.1.2 we conclude that for the abstract system from Example 3.1.1 the assignment $\sigma^\sharp = \{y \mapsto 0\}$ is a sound description of every corresponding concrete system, since $\sigma^\sharp$ is a post-solution of the corresponding lower monotonization $y = 0$.

In general, Lemma 3.1.2 provides us with a sufficient condition guaranteeing that an abstract assignment $\sigma^\sharp$ is sound w.r.t. the concrete system (3.1) and the

description relation $\mathcal{R}$, namely, that $\sigma^\sharp$ is a post-solution of the system (3.5). This sufficient condition is remarkable as it is an *intrinsic* property of the abstract system since it does not refer to the concrete system. As a corollary we obtain:

Corollary 3.1.3. *Every post-solution $\sigma^\sharp$ of the abstract system (3.2) is sound.*

Proof. For all $y \in Y$, $\sigma^\sharp\, y \sqsupseteq f^\sharp_y\, \sigma^\sharp \sqsupseteq \underline{f}^\sharp_y\, \sigma^\sharp$ holds. Accordingly, $\sigma^\sharp$ is a post-solution of the lower monotonization of the abstract system and therefore sound. □

Before we go on let us introduce some basic notation. We define an update operator $\oplus$ for total functions $g \colon Y \to \mathbb{D}$ and partial functions $h \subseteq Y \to \mathbb{D}$ as follows. For all $y \in Y$ we have

$$(g \oplus h)(y) \coloneqq \begin{cases} g(y) & \text{if } h \text{ is not defined for } y, \\ h(y) & \text{otherwise.} \end{cases}$$

Furthermore by $\underline{\bot} \colon Y \to \mathbb{D}$ respectively $\underline{\top} \colon Y \to \mathbb{D}$ we denote two constant functions which return $\bot$ respectively $\top$ for all inputs, i.e., for all $y \in Y$ we have

$$\underline{\bot}(y) \coloneqq \bot \quad \text{and} \quad \underline{\top}(y) \coloneqq \top.$$

3.2 Widening and Narrowing

It is instructive to recall the basic algorithmic approach to determine non-trivial post-solutions of abstract systems (3.2) when the set Y of variables is finite, all right-hand sides are monotone and the complete lattice $\mathbb{D}$ has finite strictly increasing chains only. In this case, *chaotic iteration* may be applied. This kind of iteration starts with the initial assignment $\underline{\bot}$ which assigns $\bot$ to every variable $y \in Y$ and then repeatedly evaluates right-hand sides to update the values of variables until the values for all variables have stabilized. This method may also be applied if right-hand sides are non-monotone. The only modification required is to update the value for each variable not just with the new value provided by the left-hand side, but with some upper bound of the old value for a variable with the new value. As a result, a *post-solution* of the system is computed which, according to Corollary 3.1.3, is sound.

The situation is more intricate, if the complete lattice in question has strictly ascending chains of infinite length. Here, we follow Cousot and Cousot [CC77a;

CC92; Cou15] who suggest to accelerate iteration by means of *widening* and *narrowing*. A widening operator $\nabla \colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ takes the old value $a \in \mathbb{D}$ and a new value $b \in \mathbb{D}$ and combines them to a value $a \sqcup b \sqsubseteq a \nabla b$ such that for any sequence b_i, $i \geq 0$, and any value a_0, the sequence $a_{i+1} = a_i \nabla b_i$, $i \geq 0$, is ultimately stable.

In contrast, a narrowing operator $\Delta \colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ takes the old value $a \in \mathbb{D}$ and a new value $b \in \mathbb{D}$ and combines them to a value $a \Delta b$ satisfying $a \sqcap b \sqsubseteq a \Delta b \sqsubseteq a$ such that for any sequence b_i, $i \geq 0$, and any value a_0, the sequence $a_{i+1} = a_i \Delta b_i$, $i \geq 0$, is ultimately stable.

While the widening operator is meant to reach a post-solution after a finite number of updates to each variable of the abstract system, the narrowing operator allows to improve upon a variable assignment once it is known to be sound. In particular, if all right-hand sides are monotone, the result of a narrowing iteration, if started with a post-solution of the abstract system, again results in a post-solution. Accordingly, the returned variable assignment can easily be verified to be sound. In analyzers which iterate according to the syntactical structure of programs such as Astrée [Cou+09], this strict separation into two phases, though, has been given up. There, when iterating over one loop, narrowing for the current loop is triggered as soon as locally a post-solution has been attained. This kind of intertwining widening and narrowing is systematically explored in [ASV13; Ama+16]. There, a widening operator is combined with a narrowing operator into a single derived operator $\boxtriangledown$ defined by

$$a \boxtriangledown b = \begin{cases} a \Delta b & \text{if } b \sqsubseteq a, \\ a \nabla b & \text{otherwise.} \end{cases}$$

The operator is also called *warrowing* operator.

Solvers which perform chaotic iteration and use warrowing to combine old values with new contributions, necessarily return post-solutions whenever they terminate. In [ASV13; ASV16], termination could only be guaranteed for systems of equations where all right-hand sides are monotone. For *non-monotone* systems as may occur at interprocedural analysis, only practical evidence could be provided for the proposed algorithms to terminate in interesting cases.

Here, our goal is to lift these limitations by providing solvers which terminate for all finite abstract systems of equations and all complete lattices—no matter whether right-hand sides are monotone or not. For that purpose, we dissolve the operator $\boxtriangledown$ again into its components. Instead, we equip the solving routines with extra logic to decide when to apply which operator such that termination is again guaranteed.

3.3 Terminating Structured Round-Robin Iteration

Let us consider a finite abstract system as given by:

$$y_i = f_i^\sharp, \qquad (i = 1, \ldots, n) \tag{3.6}$$

In [ASV13], a variation of round-robin iteration is presented which is guaranteed to terminate for monotone systems, while it may not terminate for non-monotone systems. In order to remedy this failure, we re-design this algorithm by additionally maintaining a flag which indicates whether the variable presently under consideration has or has not reached a sound value (Figure 3.1). Solving starts with a call $\mathsf{solve}(\mathsf{false}, n)$ where n is the highest priority of a variable. A variable y_i has a higher priority than a variable y_j whenever $i > j$ holds. A call $\mathsf{solve}(b, i)$ considers variables up to priority i only. The Boolean argument b indicates whether a sound abstraction (relative to the current values of the higher priority variables) has already been reached. The algorithm first iterates on the lower priority variables (if there are any). Once solving of these is completed, the right-hand side $f_i^\sharp$ of the current variable y_i is evaluated and stored in the variable *tmp*. Additionally, b' is initialized with the Boolean argument b.

First assume that b has already the value true. Then the old value $\sigma[y_i]$ is combined with the new value in *tmp* by means of the narrowing operator giving the new value of *tmp*. If that is equal to the old value, we are done and solve returns. Otherwise, $\sigma[y_i]$ is updated to *tmp*, and $\mathsf{solve}(\mathsf{true}, i)$ is called tail-recursively.

Next assume that b has the value false. Then the algorithm distinguishes two cases. If the old value $\sigma[y_i]$ exceeds the new value, the variable *tmp* receives the combination of both values by means of the narrowing operator. Additionally, b' is set to true. Otherwise, the new value for *tmp* is obtained by means of widening. Again, if the resulting value of *tmp* is equal to the current value $\sigma[y_i]$ of y_i, the algorithm returns, whereas if they differ, then $\sigma[y_i]$ is updated to *tmp* and the algorithm recursively calls itself for the actual parameters (b', n).

In light of Theorem 3.3.1, the resulting algorithm is called *Terminating Structured Round-Robin iteration* or TSRR for short.

Theorem 3.3.1. *The algorithm in Figure 3.1 terminates for all finite abstract systems of the form (3.6). Upon termination, it returns a variable assignment σ which is sound. If all right-hand sides are monotone, σ is a post-solution.*

```
void solve(b, i) {
    if (i ≤ 0) return;
    solve(b, i − 1);
    tmp ≔ f_i^♯ σ;
    b′ ≔ b;
    if (b) tmp ≔ σ[y_i] Δ tmp;
    else if (tmp ⊑ σ[y_i]) {
        tmp ≔ σ[y_i] Δ tmp;
        b′ ≔ true;
    } else tmp ≔ σ[y_i] ∇ tmp;
    if (σ[y_i] = tmp) then return;
    σ[y_i] ≔ tmp;
    solve(b′, i);
}
```

Figure 3.1: Terminating Structured Round-Robin iteration (TSRR)

Proof. By induction on i, we prove that $\mathsf{solve}(b, i)$ terminates. For $i = 0$, the statement is obviously true. Now assume that $i > 0$ and that by induction hypothesis, $\mathsf{solve}(b', i-1)$ terminates for $b' \in \{\mathsf{true}, \mathsf{false}\}$. First consider the case where $b = \mathsf{true}$. In this case, the flag b' for the tail-recursive call will be equal to true as well, and only narrowing will be applied to y_i. Therefore, the sequence of tail-recursive calls $\mathsf{solve}(b, i)$ eventually will terminate.

Now consider the case where $b = \mathsf{false}$. By induction hypothesis, all recursive calls $\mathsf{solve}(b', i-1)$ terminate. Consider the sequence of tail recursive calls where the flag b' is not set to true. Within this sequence, the new values for y_i form an ascending chain $d_0 \sqsubset d_1 \sqsubset d_2 \cdots$ where $d_{j+1} = d_j \nabla a_j$ for suitable values a_j. Due to the properties of a widening operator, this sequence is finite, i.e., there is some j such that $d_{j+1} \sqsubseteq d_j$. In this case the call either terminates directly or recursively calls $\mathsf{solve}(b', i)$ for $b' = \mathsf{true}$. Therefore, solving terminates also in this case.

It remains to prove that upon termination, a sound variable assignment σ is found. For $j = 1, \ldots, n$, and a variable assignment $\rho : \{y_1, \ldots, y_n\} \to \mathbb{D}$, we consider the system $\mathcal{E}_{\rho,j}$ defined by:

$$y_i = f^\sharp_{\rho,i}, \qquad (i = 1, \ldots, j)$$

with $f^\sharp_{\rho,i}\, \sigma^\sharp = f_i(\rho \oplus \sigma^\sharp)$ for $\sigma^\sharp : \{y_1, \ldots, y_j\} \to \mathbb{D}$. Let $\underline{\mathcal{E}}_{\rho,j}$ denote the lower monotonization of $\mathcal{E}_{\rho,j}$. We claim:

1. Assume σ_1 is a post-solution of the system $\underline{\mathcal{E}}_{\rho,j}$. Then $\mathsf{solve}(\mathsf{true}, j)$ when started with $\rho_1 = \rho \oplus \sigma_1$, returns with a variable assignment $\rho_2 = \rho \oplus \sigma_2$ where σ_2 is still a post-solution of $\underline{\mathcal{E}}_{\rho,j}$.

2. $\mathsf{solve}(\mathsf{false}, j)$ returns a post-solution of $\underline{\mathcal{E}}_{\rho,j}$.

We proceed by induction on j. For $j = 0$, nothing must be proven. Therefore assume $j > 0$. Consider the first claim. As post-solutions of $\underline{\mathcal{E}}_{\rho,j}$ are preserved by each update which combines an old value $\sigma[y_i]$ with the value of the corresponding right-hand side $f^\sharp_{\rho,i}$ for σ by means of $\sqcap$ and thus also by Δ, the claim follows.

For a proof of the second claim, let us consider the sub-sequence of tail-recursive calls $\mathsf{solve}(b, j)$ where b' remains false. Eventually this sequence ends with a last call where b' is set to true. Let ρ' denote the variable assignment before this update occurs. Then $\rho'(y_j) \sqsupseteq f_j\,\rho' \sqsupseteq \underline{f}_j\,\rho'$. Likewise, by induction hypothesis, $\rho'|_{\{y_1,\ldots,y_{j-1}\}}$ is a post-solution of $\underline{\mathcal{E}}_{\rho',j-1}$. Altogether therefore, $\rho' = \rho \oplus \sigma'$ for some variable assignment $\sigma' \colon \{y_1, \ldots, y_j\} \to \mathbb{D}$ which is a post-solution of $\underline{\mathcal{E}}_{\rho',j}$. Accordingly, $\mathsf{solve}(\mathsf{false}, j)$ either directly terminates with ρ', and the second claim follows, or $\mathsf{solve}(\mathsf{true}, j)$ is called, and the second claim follows from the first one. This completes the proof of the two claims. Since the second claim, instantiated with $j = n$, implies that the variable assignment returned by the algorithm is a post-solution of the lower monotonization of the system, it is sound. By Lemma 3.1.1, it is then also a post-solution of the original abstract system whenever all right-hand sides are monotone. □

In fact, for monotone systems, the new variation of round-robin iteration behaves identical to the algorithm SRR from [ASV13].

3.4 Local Solvers

Local solving may gain speed-ups by querying the value of only a small subset of variables which is still sufficient to answer the initial query. Such solvers are at the heart of program analysis frameworks such as the CIAO system [Her+05; Her+12] or GOBLINT [Voj+16]. In order to reason about *partial* variable assignments as computed by local solvers, we can no longer consider right-hand sides in equations as black boxes, but require a notion of *variable dependence*.

For the concrete system we assume that right-hand sides are mathematical functions of type $(X \to \mathbb{C}) \to \mathbb{C}$ where for any such function f and variable

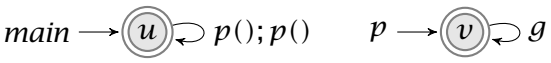
assignment $\sigma : X \to \mathbb{C}$, we are given a superset $\mathsf{dep}(f, \sigma)$ of variables onto which f possibly depends, i.e.,

$$(\forall x \in \mathsf{dep}(f,\sigma).\, \sigma[x] = \sigma'[x]) \quad \Longrightarrow \quad f\,\sigma = f\,\sigma' \tag{3.7}$$

for all $\sigma' : X \to \mathbb{C}$. Let $\sigma : X \to \mathbb{C}$ denote a solution of the concrete system. Then we call a subset $X' \subseteq X$ of variables *σ-closed*, if for all $x \in X'$, $\mathsf{dep}(f_x, \sigma) \subseteq X'$. Then for every x contained in the σ-closed subset X', the value $f_x\,\sigma$ can be determined already if the values of σ are known for the variables in X' only.

In [MH90; SF00] it is observed that for suitable formulations of interprocedural analysis, the set of all run-time calling contexts of procedures can be extracted from σ-closed sets of variables.

Example 3.4.1. Consider the following example program



consisting of a procedure *main* which at some program point u inside a loop calls the procedure p twice in a row. The procedure p iterates on some program point v by repeatedly applying the function $g : Q \to 2^Q$ which describes the operational behaviour of the body of the loop at v. The following system of equations arises from the program

$$\begin{array}{lcl} \langle u,q\rangle & = & \bigcup \{\, \langle v,q_1\rangle \mid q_1 \in \bigcup \{\, \langle v,q_2\rangle \mid q_2 \in \langle u,q\rangle \,\} \,\} \cup \{q\} \\ \langle v,q\rangle & = & \bigcup \{\, g\,q_1 \mid q_1 \in \langle v,q\rangle \,\} \cup \{q\} \end{array}$$

for $q \in Q$. Here, Q is a superset of all possible system states, and the unary function $g : Q \to 2^Q$ describes the operational behavior of the body of the loop at v. The set of variables of this system is given by $X = \{\langle u,q\rangle, \langle v,q\rangle \mid q \in Q\}$ where $\langle u,q\rangle, \langle v,q\rangle$ represent the sets of program states possibly occurring at program points u and v, respectively, when the corresponding procedures have been called in context q. For any variable assignment σ, the dependence sets of the right-hand sides are naturally defined by:

$$\begin{array}{lcl} \mathsf{dep}(f_{\langle u,q\rangle},\sigma) & = & \{\langle u,q\rangle\} \cup \{\langle v,q_2\rangle \mid q_2 \in \sigma\,\langle u,q\rangle\} \\ & & \qquad \cup \{\langle v,q_1\rangle \mid q_2 \in \sigma\langle u,q\rangle, q_1 \in \sigma\,\langle v,q_2\rangle\} \\ \mathsf{dep}(f_{\langle v,q\rangle},\sigma) & = & \{\langle v,q\rangle\} \end{array}$$

where f_x again denotes the right-hand-side function for a variable x. Assuming that $g(q_0) = \{q_1\}$ and $g(q_1) = \emptyset$, then the least solution σ maps

$\langle u, q_0\rangle, \langle v, q_0\rangle$ to the set $\{q_0, q_1\}$ and maps $\langle u, q_1\rangle, \langle v, q_1\rangle$ to $\{q_1\}$. Accordingly, the set $\{\langle u, q_i\rangle, \langle v, q_i\rangle \mid i = 0, 1\}$ is σ-closed. We conclude, given the program is called with initial context q_0, that the procedure p is called with contexts q_0 and q_1 only. ■

In concrete systems of equations, right-hand sides may depend on *infinitely* many variables. Since abstract systems are meant to give rise to effective algorithms, we impose more restrictive assumptions onto their right-hand-side functions. For these, we insist that only finitely many variables may be queried. Following the considerations in [HKS10b; HKS10a; Kar13], we demand that every right-hand side $f^\sharp$ of the abstract system is *pure* in the sense of [HKS10b]. This means that, operationally, the evaluation of $f^\sharp$ for any abstract variable assignment $\sigma^\sharp$ consists of a finite sequence of variable look-ups before eventually, a value is returned. Technically, $f^\sharp$ can be represented by a *computation tree*, i.e., is an element of

$$\mathsf{tree} \coloneqq \mathsf{Answer}\ \mathbb{D} \quad | \quad \mathsf{Query}\ Y \times (\mathbb{D} \to \mathsf{tree})$$

Thus, a computation tree either is a leaf immediately containing a value or a query, which consists of a variable together with a continuation which, for every possible value of the variable returns a tree representing the remaining computation. Each computation tree defines a function $[\![t]\!] \colon (Y \to \mathbb{D}) \to \mathbb{D}$ by:

$$\begin{array}{lcl} [\![\mathsf{Answer}\, d]\!]\, \sigma & = & d \\ [\![\mathsf{Query}\, (y, c)]\!]\, \sigma & = & [\![c\, (\sigma[y])]\!]\, \sigma \end{array}$$

Following [HKS10b], the tree representation is uniquely determined by (the operational semantics of) $f^\sharp$.

Example 3.4.2. Computation trees can be considered as generalizations of binary decision diagrams to arbitrary sets $\mathbb{D}$. For example, let $\mathbb{D} = \mathbb{N}^\infty$ and the function $f^\sharp \colon (Y \to \mathbb{D}) \to \mathbb{D}$ with $\{y_1, y_2\} \subseteq Y$, defined by

$$f^\sharp\, \sigma = \textbf{if}\ \sigma[y_1] > 5\ \textbf{then}\ 1 + \sigma[y_2]\ \textbf{else}\ \sigma[y_1]$$

is then represented by the tree

$$\begin{array}{l} \mathsf{Query}\, (y_1, \textbf{fun}\, d_1 \to \textbf{if}\ d_1 > 5\ \textbf{then} \\ \qquad\qquad \mathsf{Query}\, (y_2, \textbf{fun}\, d_2 \to \mathsf{Answer}\, (1 + d_2)) \\ \qquad\qquad \textbf{else} \\ \qquad\qquad \mathsf{Query}\, (y_1, \textbf{fun}\, d_1 \to \mathsf{Answer}\, d_1)) \end{array}$$ ■

A set $\mathsf{dep}(f^\sharp, \sigma^\sharp) \subseteq Y$ with a property analogous to (3.7) can be explicitly obtained from the tree representation t of $f^\sharp$ by defining $\mathsf{dep}(f^\sharp, \sigma^\sharp) = \mathsf{treedep}(t, \sigma^\sharp)$ where:

$$\begin{array}{lcl} \mathsf{treedep}(\mathsf{Answer}\, d, \sigma^\sharp) & = & \emptyset \\ \mathsf{treedep}(\mathsf{Query}\,(y, c), \sigma^\sharp) & = & \{y\} \cup \mathsf{treedep}(c\,(\sigma^\sharp[y]), \sigma^\sharp) \end{array}$$

Technically, this means that the value $f^\sharp\,\sigma^\sharp = [\![t]\!]\,\sigma^\sharp$ can be computed already for *partial* variable assignments $\sigma' : Y' \to \mathbb{D}$, whenever $\mathsf{dep}(f^\sharp, \underline{\top} \oplus \sigma') = \mathsf{treedep}(t, \underline{\top} \oplus \sigma') \subseteq Y'$.

Example 3.4.3. Consider the function $f^\sharp$ from Example 3.4.2 together with the partial assignment $\sigma' = \{y_1 \mapsto 3\}$. Then $\mathsf{dep}(f^\sharp, \underline{\top} \oplus \sigma') = \{y_1\}$. ■

We call a partial variable assignment $\sigma' : Y' \to \mathbb{D}$ *closed* (w.r.t. an abstract system (3.2)), if for all $y \in Y'$, $\mathsf{dep}(f^\sharp_y, \underline{\top} \oplus \sigma') \subseteq Y'$.

In the following, we strengthen the description relation $\mathcal{R}$ additionally to take variable dependencies into account. We say that the abstract system (3.2) *simulates* the concrete system (3.1) (relative to the description relation $\mathcal{R}$) iff for all pairs x, y of variables with $x\ \mathcal{R}\ y$, such that for the concrete and abstract right-hand sides f_x and $f^\sharp_y$, respectively, property (3.3) holds and additionally $\mathsf{dep}(f_x, \sigma)\ \mathcal{R}\ \mathsf{dep}(f^\sharp_y, \sigma^\sharp)$ whenever $\sigma\ \mathcal{R}^*\ \sigma^\sharp$. Here, a pair of sets X', Y' of concrete and abstract variables is in relation $\mathcal{R}$ if for all $x \in X'$, $x\ \mathcal{R}\ y$ for some $y \in Y'$. Theorem 3.4.1 demonstrates the significance of closed abstract assignments which are sound.

Theorem 3.4.1. *Assume that the abstract system (3.2) simulates the concrete system (3.1) (relative to $\mathcal{R}$) where σ is the least solution of the concrete system. Assume that $\sigma^\sharp : Y' \to \mathbb{D}$ is a partial assignment with the following properties:*

1. *$\sigma^\sharp$ is closed;*
2. *$\underline{\top} \oplus \sigma^\sharp$ is a post-solution of the lower monotonization of the abstract system.*

Then the set $X' = \{\, x \in X \mid \exists\, y \in Y'.\, x\ \mathcal{R}\ y \,\}$ is σ-closed.

Proof. By Lemma 3.1.2, $\sigma\ \mathcal{R}^*\ (\underline{\top} \oplus \sigma^\sharp)$ holds. Now assume that $x\ \mathcal{R}\ y$ for some $x \in X$ and $y \in Y'$. By definition therefore, $\mathsf{dep}(f_x, \sigma)\ \mathcal{R}\ \mathsf{dep}(f^\sharp_y, \underline{\top} \oplus \sigma^\sharp)$. Since the latter is a subset of Y', the former must be a subset of X', and the assertion follows. □

3.5 Terminating Structured Two-Phase Solving

We first present a local version of a two-phase algorithm to determine a sound variable assignment for an abstract system of equations. As the algorithm is local, no pre-processing of the equation system is possible. Accordingly, variables where widening or narrowing is to be applied must be determined dynamically (in contrast to solvers based on static variable dependencies where widening points can be statically determined [Bou93]). We solve this problem by assigning *priorities* to variables in decreasing order in which they are encountered, and consider a variable as a candidate for widening/narrowing whenever it is queried during the evaluation of a lower priority variable. The second issue is that during the narrowing iteration of the second phase, variables may be encountered which have not yet been seen and for which therefore no sound approximation is available. In order to deal with this situation, the algorithm does not maintain a single variable assignment, but two distinct ones. While assignment σ_0 is used for the widening phase, σ_1 is used for narrowing with the understanding that, once the widening phase is completed, the value of a variable y from σ_0 is copied as the initial value of y into σ_1. This clear distinction allows to continue the widening iteration for every newly encountered variable y' in order to determine an acceptable initial value before continuing with the narrowing iteration. The resulting algorithm can be found in Figure 3.2.

Initially, the priority queue Q and the sets dom_i and dom are empty. Accordingly, the mappings $\sigma_i : \mathsf{dom}_i \to \mathbb{D}$ and $\mathsf{infl} : \mathsf{dom} \to 2^Y$ are also empty. Likewise, the set point is initially empty. Solving for the variable y_0 starts with the call $\mathsf{solve}_1(y_0, 0)$.

Let us first consider the functions $\mathsf{solve}_0, \mathsf{iterate}_0, \mathsf{do\_var}_0$. These are meant to realize a local widening iteration. A call $\mathsf{solve}_0(y)$ first checks whether $y \in \mathsf{dom}_0$. If this is the case, solving immediately terminates. Otherwise, $\sigma_0[y]$ is initialized with $\bot$, y is added to dom_0, $\mathsf{infl}[y]$ is set to the empty set, and y receives the next available priority by means of the call $\mathsf{next\_prio}()$. Subsequently, $\mathsf{do\_var}_0(y)$ is called, followed by a call to $\mathsf{iterate}_0(\mathsf{prio}[y])$ to complete the widening phase for y. Upon termination, a call $\mathsf{iterate}_0(n)$ for an integer n has removed all variables of priority at most n from the queue Q. It proceeds as follows. If Q is empty or contains only variables of priority exceeding n, it immediately returns. Otherwise, the variable y with least priority is extracted from Q. Having processed $\mathsf{do\_var}_0(y)$, the iteration continues with the tail-recursive call $\mathsf{iterate}_0(n)$.

It remains to describe the function $\mathsf{do\_var}_0$. When called with a variable

```
void iterate_0(n) {
    if (Q ≠ ∅ ∧ min_prio(Q) ≤ n) {
        y := extract_min(Q);

        do_var_0(y);
        iterate_0(n);
    }
}

void solve_0(y) {
    if (y ∈ dom_0) return;
    dom_0 := dom_0 ∪ {y};
    prio[y] := next_prio();
    σ_0[y] := ⊥;
    infl[y] := ∅;
    do_var_0(y);
    iterate_0(prio[y]);
}

void do_var_0(y) {
    isp := y ∈ point;
    point := point \ {y};
    𝔻 eval_0(z) {
        solve_0(z);
        if (prio[z] ≥ prio[y])
            point := point ∪ {z};
        infl[z] := infl[z] ∪ {y};
        return σ_0[z]);
    }
    tmp := f♯_y eval_0;
    if (isp) tmp := σ_0[y] ∇ tmp;
    if (σ_0[y] = tmp) return;
    σ_0[y] := tmp;
    forall (z ∈ infl[y]) insert z Q;
    infl[y] := ∅;
    return;
}
```

```
void iterate_1(n) {
    if (Q ≠ ∅ ∧ min_prio(Q) ≤ n) {
        y := extract_min(Q);
        solve_1(y, prio[y] − 1);
        do_var_1(y);
        iterate_1(n);
    }
}

void solve_1(y, n) {
    if (y ∈ dom_1) return;
    solve_0(y);
    dom_1 := dom_1 ∪ {y};
    σ_1[y] := σ_0[y];
    forall (z ∈ {y} ∪ infl[y]) insert z Q;
    infl[y] := ∅;
    iterate_1(n);
}

void do_var_1(y) {
    isp := y ∈ point;
    point := point \ {y};
    𝔻 eval_1(z) {
        solve_1(z, prio[y] − 1);
        if (prio[z] ≥ prio[y])
            point := point ∪ {z};
        infl[z] := infl[z] ∪ {y};
        return σ_1[z];
    }
    tmp := f♯_y eval_1;
    if (isp) tmp := σ[y] Δ tmp;
    if (σ_1[y] = tmp) return;
    σ_1[y] := tmp;
    forall (z ∈ infl[y]) insert z Q;
    infl[y] := ∅;
    return;
}
```

Figure 3.2: Terminating Structured Two-Phase solver (TSTP)

y, the algorithm first determines whether or not y is a widening/narrowing point, i.e., contained in the set point. If so, y is removed from point, and the flag *isp* is set to true. Otherwise, *isp* is set to false. Then the right-hand side $f_y^\sharp$ is evaluated and the result stored in the variable *tmp*. For its evaluation, the function $f_y^\sharp$, however, does not receive the current variable assignment σ_0 but an auxiliary function eval_0 which serves as a wrapper to the assignment σ_0. The wrapper function eval_0, when queried for a variable z, first calls $\mathsf{solve}_0\,(z)$ to compute a first non-trivial value for z. If the priority of z is greater or equal to the priority of y, a potential widening point is detected. Therefore, z is added to the set point. Subsequently, the fact that z was queried during the evaluation of the right-hand side of y, is recorded by adding y to the set $\mathsf{infl}[z]$. Finally, $\sigma_0[z]$ is returned.

Having evaluated $f_y^\sharp\,\mathsf{eval}_0$ and stored the result in *tmp*, the function $\mathsf{do\_var}_0$ then applies widening only if *isp* equals true. In this case, *tmp* receives the value of $\sigma[y] \triangledown tmp$. In the next step, *tmp* is compared with the current value $\sigma_0[y]$. If both values are equal, the procedure returns. Otherwise, $\sigma_0[y]$ is updated to *tmp*. The variables in $\mathsf{infl}[y]$ are inserted into the queue Q, and the set $\mathsf{infl}[y]$ is reset to the empty set. Only then the procedure returns.

The functions solve_1, $\mathsf{iterate}_1$ and $\mathsf{do\_var}_1$, on the other hand, are meant to realize the narrowing phase. They essentially work analogously to the corresponding functions solve_0, $\mathsf{iterate}_0$ and $\mathsf{do\_var}_0$. In particular, they re-use the mapping infl which records the currently encountered variable dependencies as well as the variable priorities and the priority queue Q. Instead of σ_0, dom_0, however, they now refer to σ_1, dom_1, respectively. Moreover, there are the following differences.

First, the function solve_1 now receives not only a variable, but a pair of an integer n and a variable y. When called, the function first checks whether $y \in \mathsf{dom}_1$. If this is the case, solving immediately terminates. Otherwise, $\mathsf{solve}_0(y)$ is called first. After that call, the widening phase for y is assumed to have terminated where the resulting value is $\sigma_0[y]$. Accordingly, $\sigma_1[y]$ is initialized with $\sigma_0[y]$, and y is added to dom_1. As the value of σ_1 for y has been updated, y together with all variables in $\mathsf{infl}[y]$ are added to the queue, whereupon $\mathsf{infl}[y]$ is set to the empty set, and $\mathsf{iterate}_1(n)$ is called to complete the narrowing phase up to the priority n. Upon termination, a call $\mathsf{iterate}_1(n)$ for an integer n has removed all variables of priority at most n from the queue Q. In contrast to $\mathsf{iterate}_0$, however, it may extract variables y from Q which have not yet been encountered in the present phase of iteration, i.e., are not yet included in dom_1 and thus have not yet received a value in σ_1. To ensure initialization, $\mathsf{solve}_1(y, n)$ is called for $n = \mathsf{prio}[y] - 1$. This

choice of the extra parameter n ensures that all lower priority variables have been removed from Q before $\mathsf{do\_var}_1(y)$ is called.

It remains to explain the function $\mathsf{do\_var}_1(y)$. Again, it essentially behaves like $\mathsf{do\_var}_0(y)$ with the distinction that the narrowing operator is applied instead of the widening operator. Furthermore, the auxiliary local function eval_0 is replaced with eval_1 which now uses a call to solve_1 for the initialization of its argument variable z (instead of solve_0) where the extra integer argument is given by $\mathsf{prio}[y] - 1$, i.e., an iteration is performed to remove all variables from Q with priorities lower than the priority of y (not of z).

In light of Theorem 3.5.1, we call the algorithm from Figure 3.2 *terminating structured two-phase solver*.

Theorem 3.5.1. *The local solver* TSTP *from Figure 3.2 when started with a call* $\mathsf{solve}_1(y_0, 0)$ *for a variable* y_0*, terminates for every system of equations whenever only finitely many variables are encountered.*

Upon termination, assignments $\sigma_i^\sharp : Y_i \to \mathbb{D}$, $i = 0, 1$ *are obtained for finite sets* $Y_0 \supseteq Y_1$ *of variables so that the following holds:*

1. $y_0 \in Y_1$*;*
2. $\sigma_0^\sharp$ *is a closed partial post-solution of the abstract system* (3.2)*;*
3. $\sigma_1^\sharp$ *is a closed partial assignment such that* $\bot \oplus \sigma_1^\sharp$ *is a post-solution of the lower monotonization of the abstract system* (3.2)*.*

Proof. Assume that only finitely many variables are encountered during the run of the algorithm, i.e., from some point neither dom_0 nor dom_1 receive new elements. Since $\mathsf{solve}_0(y)$ is called before the variable y is added to dom_1, and $\mathsf{solve}_0(y)$ enforces that y is included in dom_0, we have that $\mathsf{dom}_1 \subseteq \mathsf{dom}_0$ throughout the algorithm. Therefore, we define $Y_i = \mathsf{dom}_i$ when the iteration has terminated for $i = 0, 1$. Due to the initial call $\mathsf{solve}_1(y_0, 0)$, y_0 is contained in Y_1 implying the first item in the list.

Variables y are added into sets $\mathsf{infl}[z]$ only during the evaluation of a call to eval_i and after an appropriate call to solve_i implying that y is contained in dom_i whenever eval_i was called inside a call $\mathsf{do\_var}_i(y)$. Accordingly, all variables added to the priority queue necessarily are contained in dom_0. Thus, all variables for which $\mathsf{do\_var}_0$ is called at a call of $\mathsf{iterate}_0(n)$ are all contained in dom_0, while all variables for which $\mathsf{do\_var}_1$ is called at a call of $\mathsf{iterate}_1(n)$ are already contained in dom_1. We claim that for every priority n, the following holds:

1. Every call $\mathsf{iterate}_0(n)$ during the evaluation of $\mathsf{solve}_0(0, y_0)$ terminates.
2. Every call $\mathsf{iterate}_1(n)$ during the evaluation of $\mathsf{solve}_1(0, y_0)$ terminates as well.

In order to prove the first claim, assume for a contradiction that there is some n such that the call $\mathsf{iterate}_0(n)$ does not terminate. Since Y_0 is finite, there must be a variable y of maximal priority $\mathsf{prio}(y) \leq n$ so that $\mathsf{do\_var}_0(y)$ is evaluated infinitely often. This means that from some point on, y is the variable of maximal priority for which $\mathsf{do\_var}_0$ is called. Let d_i, $i \geq 0$ denote the sequence of the new values for y. We claim that for every $i \geq 0$, $d_{i+1} = d_i \nabla a_i$ holds for some suitable value a_i. This holds if $y \in \mathsf{point}$ from the first evaluation onward. Clearly, if this were the case, we arrive at a contradiction, as any such widening sequence is ultimately stable. Accordingly, it remains to prove that from the first evaluation onward, y is contained in point whenever $\mathsf{do\_var}_0(y)$ is called. Assume for a contradiction that there is a first such call where y is not contained in point. Assume that this call provided the ith value d_i for y. This means that, since the last evaluation of $f_y^\sharp$, no query to the value of y during the evaluation of lower priority variables has occurred. Accordingly, the set $\mathsf{infl}[y]$ does not contain any lower priority variables, which means that no further variable is evaluated before the next call $\mathsf{do\_var}_0(y)$. But then this next evaluation of $f_y^\sharp$ will return the value a_i. Subsequently, the queue Q does no longer contain variables of priority less then or equal to n, and therefore the iteration would terminate, in contradiction to our assumption.

Now consider the second claim. For a contradiction now assume that there is some n so that the call $\mathsf{iterate}_1(n)$ does not terminate. Since every call $\mathsf{iterate}_0(m)$ encountered during its evaluation is already known to terminate, we conclude that there must be a variable y of maximal priority less then or equal to n so that $\mathsf{do\_var}_1(y)$ is evaluated infinitely often. As before this means that from some point on, y is the variable of maximal priority for which $\mathsf{do\_var}_1(y)$ is called. Let d_i, $i \geq 0$ denote the sequence of the new values for y. We claim that for every $i \geq 0$, $d_{i+1} = d_i \Delta a_i$ holds for some suitable value a_i. This holds if $d_i \sqsubseteq d_{i+1}$ and $y \in \mathsf{point}$ from $i = 1$ onward. Again, if this were the case, we arrive at a contradiction, as any such narrowing sequence is ultimately stable. Accordingly, it remains to prove that from the first evaluation onward, y is contained in point whenever $\mathsf{do\_var}_1(y)$ is called. This, however, follows by the same argument as for $\mathsf{iterate}_0(y)$. This completes the proof of the claim.

By the claim which we have just proven, each occurring call $\mathsf{iterate}_i(n)$ will terminate. From that, the termination of the call $\mathsf{solve}_1(y_0, 0)$ follows as stated by the theorem.

It remains to prove the remaining two assertions of the enumeration. Again, we assume that only finitely many variables are encountered in a run of the local two-phase solver when started for a variable y_0, and assume that after some call to $\mathsf{do\_var}_i$, no further variable is added to dom_0, and likewise no further variable is added to dom_1. In order to prove the second assertion, we prove that the following invariants hold before every call $\mathsf{do\_var}_i(y_1)$:

1. For every variable y in the current domain dom_0, $\mathsf{infl}[y]$ contains (at least) all variables $z \notin Q \cup \{y_1\}$ whose last evaluation of $f_z^\sharp$ has called $\mathsf{eval}_i(y)$;
2. If $y \in \mathsf{dom}_0 \setminus (Q \cup \{y_1\})$, then $\sigma_0[y] \sqsupseteq f_y^\sharp(\underline{\top} \oplus \sigma_0)$;
3. If $y \in \mathsf{dom}_1 \setminus (Q \cup \{y_1\})$, then $\sigma_1[y] \sqsupseteq \underline{f}_y^\sharp(\underline{\top} \oplus \sigma_1)$;

Here, $\underline{\top}$ is the variable assignment which maps each variable in Y_0 to $\top$.

We prove the first invariant. The set $\mathsf{infl}[y]$ is initialized to the empty set in function solve_i, i.e., when the variable is solved for the very first time. In that case no prior call to $\mathsf{eval}_i(y)$ from any prior evaluation of any $f_z^\sharp$ happened. Therefore, no other variable depends on y and $\mathsf{infl}[y] = \emptyset$ holds. Now assume that the variable y is initialized. Then $\mathsf{infl}[y]$ is set to the empty set only in a call to function $\mathsf{do\_var}_i$. Before $\mathsf{infl}[y]$ is set to the empty set, all variables which are influenced by y are added to the queue Q. Therefore, all variables which are influenced by y are in Q or are in the set $\mathsf{infl}[y]$ which proves the first invariant.

We prove the second invariant. Consider variables y whose current iteration already terminated, i.e., which are neither in Q nor equal y_1. For those variables y we have that the current iteration only terminates if $\sigma_0[y] \sqsubseteq f_y^\sharp(\underline{\top} \oplus \sigma_0)$ holds, proving the second invariant.

We prove the third invariant. For that, consider a call to $\mathsf{do\_var}_1(y_1)$. If this is the very first call of $\mathsf{do\_var}_1$ for y_1, then this occurs inside a call $\mathsf{solve}_1(y_1)$. Accordingly, the value $\sigma_1[y]$ has been initialized to $\sigma_0[y]$. Then we have, by the second invariant:

$$\sigma_1[y_1] = \sigma_0[y_1] \sqsupseteq f_{y_1}^\sharp(\underline{\top} \oplus \sigma_0) \sqsupseteq \underline{f}_{y_1}^\sharp(\underline{\top} \oplus \sigma_0)$$

At that moment, the priority queue does not contain any variable y with priority less or equal the priority of y_1, implying that for all these y, $\sigma_1[y] \sqsupseteq \underline{f}_y^\sharp(\underline{\top} \oplus \sigma_1)$ holds. Accordingly, all variables z from $\mathsf{infl}[y_1]$ with priority less or equal to y_1 will subsequently be iterated upon with $\mathsf{iterate}_1$. But since these

variables z satisfy $\sigma_1[z] \sqsupseteq \underline{f}^\sharp_z(\bot \oplus \sigma_1)$, the invariant holds for the calls of $\mathsf{do\_var}_1$ therein.

By construction, 0 is the maximal priority of any variable. y_0 receives the greatest priority. Therefore, $\mathsf{solve}_1(0, y_0)$ returns with an empty queue Q. By the third invariant the second assertion of the theorem follows. □

3.6 Terminating Structured Mixed-Phase Solving

The draw-back of the two-phase solver TSTP from the last section is that it may lose precision already in very simple situations.

Example 3.6.1. Consider the system:

$$\begin{aligned} y_1 &= \mathsf{max}(y_1, y_2) \\ y_2 &= \mathsf{min}(y_3, 2) \\ y_3 &= y_2 + 1 \end{aligned}$$

over the complete lattice $\mathbb{N} \cup \{\infty\}$ of natural numbers (equipped with the natural ordering and extended with ∞ as top element) and the following widening and narrowing operators:

$$a \nabla b = \begin{cases} \infty & \text{if } a < b \\ a & \text{otherwise} \end{cases}$$

$$a \Delta b = \begin{cases} b & \text{if } a = \infty \\ a & \text{otherwise} \end{cases}$$

Then $\mathsf{solve}_0(y_1)$ detects y_2 as the only widening point resulting in

$$\sigma_0 = \{y_1 \mapsto \infty, y_2 \mapsto \infty, y_3 \mapsto \infty\}$$

A call to $\mathsf{solve}_1(y_1, 0)$ therefore initializes y_1 with ∞ implying that $\sigma_1[y_1] = \infty$ irrespective of the fact that $\sigma_1[y_2] = 2$. ■

We may therefore aim at intertwining the two phases into one without sacrificing the termination guarantee. The idea is to operate on a single variable assignment only and iterate on each variable first in widening and then in narrowing mode. In order to keep soundness, after every update of a variable y in the widening phase, all possibly influenced lower priority variables are iterated upon until all stabilize with widening and narrowing. Only then the

widening iteration on y continues. If on the other hand an update for y occurs during narrowing, the iteration on possibly influenced lower priority variables is with narrowing only. The distinction between the two modes of the iteration is maintained by a flag where false and true correspond to the widening and narrowing phases, respectively. The algorithm is provided in Figure 3.3.

Initially, the priority queue Q and the set dom are empty. Accordingly, the mappings $\sigma : \mathsf{dom} \to \mathbb{D}$ and $\mathsf{infl} : \mathsf{dom} \to Y$ are also empty. Likewise, the set point is initially empty. Solving for the variable y_0 starts with the call $\mathsf{solve}(y_0)$. Solving for some variable y first checks whether $y \in \mathsf{dom}$. If this is the case, solving immediately terminates. Otherwise, y is added to dom and receives the next available priority by means of a call to next_prio. That call should provide a value which is less than any priority of a variable in dom. Subsequently, the entries $\sigma[y]$ and $\mathsf{infl}[y]$ are initialized to $\bot$ and the empty set, respectively, and do_var is called for the pair (false, y). The return value of this call is stored in the Boolean variable b'. During its execution, this call may have inserted further variables into the queue Q. These are dealt with by the call $\mathsf{iterate}(b', \mathsf{prio}[y])$.

Upon termination, a call $\mathsf{iterate}(b, n)$ has removed all variables of priority at most n from the queue Q. It proceeds as follows. If Q is empty or contains only variables of priority exceeding n, it immediately returns. Otherwise, the variable y with least priority n' is extracted from Q. For (b, y), do_var is called and the return value of this call is stored in b'.

Now we distinguish several cases. If $b = \mathsf{true}$, then the value b' returned by do_var will necessarily be true as well. In that case, iteration proceeds by tail-recursively calling again $\mathsf{iterate}(\mathsf{true}, n)$. If on the other hand $b = \mathsf{false}$, then the value b' returned by do_var can be either true or false. If $b' = \mathsf{false}$ or $b' = \mathsf{true}$ and $n' = n$, then $\mathsf{iterate}(b', n)$ is tail-recursively called. If, however, $b' = \mathsf{true}$ and $n > n'$, then first a sub-iteration is triggered for (true, n') before the main loop proceeds with the call $\mathsf{iterate}(\mathsf{false}, n)$.

It remains to describe the function do_var. When called for a pair (b, y) consisting of a Boolean value b and variable y, the algorithm first determines whether or not y is a widening/narrowing point, i.e., contained in the set point. If so, y is removed from point, and the flag *isp* is set to true. Otherwise, *isp* is just set to false. Then the right-hand side $f_y^\sharp$ is evaluated and the result stored in the variable *tmp*. For its evaluation, the function $f_y^\sharp$, however, does not receive the current variable assignment σ, but an auxiliary function eval which serves as a wrapper to σ. The wrapper function eval, when queried for a variable z, first calls $\mathsf{solve}(z)$ to compute a first non-trivial value for z. If the priority of z exceeds or is equal to the priority of y, a potential

```
void iterate(b, n) {
    if (Q ≠ ∅ ∧ min_prio(Q) ≤ n) {
        y := extract_min(Q);
        b' := do_var(b, y);
        n' := prio[y];
        if (b ≠ b' ∧ n > n') {
            iterate(b', n');
            iterate(b, n);
        } else iterate(b', n);
    }
}
```

```
void solve(y) {
    if (y ∈ dom) return;
    dom := dom ∪ {y};
    prio[y] := next_prio();
    σ[y] := ⊥;
    infl[y] := ∅;
    b' := do_var(false, y);
    iterate(b', prio[y]);
}
```

```
bool do_var(b, y) {
    isp := y ∈ point;
    point := point \ {y};
    𝔻 eval(z) {
        solve(z);
        if (prio[z] ≥ prio[y])
            point := point ∪ {z};
        infl[z] := infl[z] ∪ {y};
        return σ[z];
    }
    tmp := f_y^♯ eval;
    b' := b;
    ...
```

```
    ...
    if (isp)
        if (b) tmp := σ[y] Δ tmp;
        else if (tmp ⊑ σ[y]) {
            tmp := σ[y] Δ tmp;
            b' := true;
        } else tmp := σ[y] ∇ tmp;
    if (σ[y] = tmp) return true;
    σ[y] := tmp;
    forall (z ∈ infl[y]) insert z Q;
    infl[y] := ∅;
    return b';
}
```

Figure 3.3: Terminating Structured Mixed-Phase solver (TSMP)

widening/narrowing point is detected. Therefore, z is added to the set point. Subsequently, the fact that the value of z was queried during the evaluation of the right-hand side of y, is recorded by adding y to the set infl$[z]$. Finally, the value $\sigma[z]$ is returned.

Having evaluated $f_y^\sharp$ eval and stored the result in *tmp*, the function do_var then decides whether to apply widening or narrowing or none of them according to the following scheme. If *isp* has not been set to true, no widening or narrowing is applied. In this case, the flag b' receives the value b. Therefore now consider the case *isp* = true. Again, the algorithm distinguishes three cases. If b = true, then necessarily narrowing is applied, i.e., *tmp* is updated to the value of $\sigma[y] \,\Delta\, tmp$, and b' still equals b, i.e., true. If b = false then narrowing is applied whenever $tmp \sqsubseteq \sigma[y]$ holds. In that case, *tmp* is set to $\sigma[y] \,\Delta\, tmp$, and b' to true. Otherwise, i.e., if b = false and $tmp \not\sqsubseteq \sigma[y]$, then widening is applied by setting *tmp* to $\sigma[y] \,\nabla\, tmp$, and b' obtains the value false.

In the next step, *tmp* is compared with the current value $\sigma[y]$. If their values are equal, the value true is returned. Otherwise, $\sigma[y]$ is updated to *tmp*. The variables in infl$[y]$ are inserted into the queue Q, and the set infl$[y]$ is reset to the empty set. Only then the value of b' is returned.

Example 3.6.2. Consider the system of equations from Example 3.6.1. Calling solve for variable y_1 will assign the priorities $0, -1, -2$ to the variables y_1, y_2 and y_3, respectively. Evaluation of the right-hand side of y_1 proceeds only after solve(y_2) has terminated. During the first update of y_2, y_2 is inserted into the set point, implying that at the subsequent evaluation the widening operator is applied resulting in the value ∞ for y_2 and y_3. The subsequent narrowing iteration on y_2 and y_3 improves these values to 2 and 3, respectively. Only then the value for y_1 is determined which is 2. During that evaluation, y_1 has also been added to the set point. The repeated evaluation of its right-hand side, will however, again produce the value 2 implying that the iteration terminates with the assignment

$$\sigma = \{y_1 \mapsto 2, y_2 \mapsto 2, y_3 \mapsto 3\}$$ ■

In light of Theorem 3.6.1, we call the algorithm from Figure 3.3, *terminating structured mixed-phase solver* or TSMP for short.

Theorem 3.6.1. *The local solver* TSMP *from Figure 3.3 when started for a variable y_0, terminates for every system of equations whenever only finitely many variables are encountered.*

Upon termination, an assignment $\sigma^\sharp : Y_0 \to \mathbb{D}$ is returned where Y_0 is the set of variables encountered during $\mathsf{solve}(y_0)$ *such that the following holds:*

- $y_0 \in Y_0$,
- $\sigma^\sharp$ *is a closed partial assignment such that* $\underline{\top} \oplus \sigma^\sharp$ *is a post-solution of the lower monotonization of the abstract system* (3.2).

Proof. Assume that only variables from the finite set Y_0 are encountered during the run of the algorithm. We claim that for every priority i, the following holds:

1. Every call $\mathsf{iterate}(\mathsf{true}, i)$ during the evaluation of $\mathsf{solve}(y_0)$ terminates.
2. Every call $\mathsf{iterate}(\mathsf{false}, i)$ during the evaluation of $\mathsf{solve}(y_0)$ terminates as well.

In order to prove the first claim, assume for a contradiction that there is some i such that the call $\mathsf{iterate}(\mathsf{true}, i)$ does not terminate. Note that then any subsequent call to do_var as well as iterate will always be evaluated for the Boolean value true. Since Y_0 is finite, there is a variable y of maximal priority $\mathsf{prio}(y) \leq i$ so that $\mathsf{do\_var}(\mathsf{true}, y)$ is evaluated infinitely often. This means that from some point on, y is the variable of maximal priority for which do_var is called. Let d_i, $i \geq 0$ denote the sequence of the new values for y. We claim that for every $i \geq 0$, $d_{i+1} = d_i \,\Delta\, a_i$ holds for some suitable value a_i. This holds if $y \in \mathsf{point}$ from the first evaluation onward. Clearly, if this were the case, we arrive at a contradiction, as any such narrowing sequence is ultimately stable.

Accordingly, it remains to prove that from the first evaluation onward, y is contained in point whenever $\mathsf{do\_var}(\mathsf{true}, y)$ is called. Assume for a contradiction that there is a first such call where y is not contained in point. Assume that this call provided the ith value d_i for y. This means in particular that, since the last evaluation of $f^\sharp_y$, no query to the value of y during the evaluation of lower priority variables has occurred. Accordingly, the set $\mathsf{infl}[y]$ does not contain any lower priority variables, which means that no further variable is evaluated before the next call $\mathsf{do\_var}(\mathsf{true}, y)$. But then this next evaluation of $f^\sharp_y$ will return the value d_i. Subsequently, the queue Q no longer contains variables of priority $\leq i$, and therefore the iteration would terminate, in contradiction to our assumption.

Let us therefore now consider the second claim. For a contradiction now assume that there is some i so that the call $\mathsf{iterate}(\mathsf{false}, i)$ does not terminate. Since every call $\mathsf{iterate}(\mathsf{true}, j)$ encountered during its evaluation is already

known to terminate, we conclude that there must be a variable y of maximal priority $\leq i$ so that do_var(false, y) is evaluated infinitely often. As before this means that from some point on, y is the variable of maximal priority for which do_var(false, y) is called. Let d_i, $i \geq 0$ denote the sequence of the new values for y. We claim that for every $i \geq 0$, $d_{i+1} = d_i \nabla a_i$ holds for some suitable value a_i where $y \in \mathsf{point}$ from $i = 1$ onward. Again, if this were the case, we arrive at a contradiction, as any such widening sequence is ultimately stable.

Accordingly, it remains to prove that from the first evaluation onward, y is contained in point whenever do_var(false, y) is called. This, however, follows by the same argument as for iterate(true, y). This completes the proof of the claim.

Now assume that only finitely many variables are encountered in a run of TSMP when started for a variable x, and assume that after some call to do_var, no further variable is encountered. Let Y_0 denote this set of variables. By the claim which we have just proven, each subsequent call to the function iterate will terminate. From that, the termination of the call solve(y_0) follows as stated by the theorem.

It remains to prove the second assertion. We remark that, whenever a new variable is encountered, it is added into the set dom and never removed. Let Y_0 again denote the finite set of variables encountered during solve(y_0), i.e., the final value of dom. In particular, y_0 is contained in Y_0. In order to prove the second assertion, we note that the following invariants hold before every call do_var(b, y_1):

1. For every variable y in the current domain dom, infl[y] contains (at least) all variables $z \notin Q \cup \{y_1\}$ whose last evaluation of $f_z^\sharp$ has called eval(y);
2. If $y \in \mathsf{dom} \setminus (Q \cup \{y_1\})$, then $\sigma[y] \sqsupseteq \underline{f}_y^\sharp(\underline{\top} \oplus \sigma)$;
3. If $b = \mathsf{true}$, then $\sigma[y_1] \sqsupseteq \underline{f}_{y_1}^\sharp(\underline{\top} \oplus \sigma)$.

The proof of the first and second invariant is similar as for the proof in Theorem 3.5.1. Here, $\underline{\top}$ is the variable assignment which maps each variable in Y_0 to $\top$. In order to see the third statement, we observe that iteration on a variable always starts with the flag false. Now consider a call to do_var(false, y_1) where b' is set to true. This is the case when $\sigma[y_1] \sqsupseteq tmp$ where tmp is the value of the last evaluation of the right-hand side of y_1. Accordingly,

$$\sigma[y_1] \sqsupseteq f_{y_1}^\sharp(\underline{\top} \oplus \sigma) \sqsupseteq \underline{f}_{y_1}^\sharp(\underline{\top} \oplus \sigma)$$

At that moment, the priority queue does not contain any variable y with priority less or equal the priority of y_1, implying that for y, $\sigma[y] \sqsupseteq \underline{f}^\sharp_y(\underline{\top} \oplus \sigma)$ holds. Accordingly, all variables z from $\mathsf{infl}[y_1]$ with priority less or equal to y_1 will subsequently be iterated upon with $b = \mathsf{true}$. But since these satisfy $\sigma[z] \sqsupseteq \underline{f}^\sharp_z(\underline{\top} \oplus \sigma)$, the invariant holds for the calls of do_var therein.

By construction, y_0 receives the greatest priority. Therefore, $\mathsf{solve}(y_0)$ returns with an empty queue Q. By the second invariant the second assertion of the theorem follows. □

By Theorem 3.6.1 the only condition for TSMP to terminate is that only finitely many variables are encountered. No further assumptions, e.g., w.r.t. monotonicity of right-hand sides must be made as in [ASV13; Ama+16]. Upon termination, the algorithm is guaranteed to return sound results. The returned variable assignment is a (partial) post-solution of the lower monotonization of the system, which means it may not necessarily be a post-solution of the original system, given that some right-hand sides are not monotone.

3.7 Concretization only

So far, we have assumed that we are given a *Galois connection* to relate concrete sets of states and abstract values. In some applications, though, such a Galois connection may not exist (see [CC92] for a detailed discussion). In this case, a best abstract description of a set of states (as otherwise given by means of the abstraction function α) may not always exist. Accordingly, the meet of two sound descriptions need no longer be a sound description. This implies that the concept of *lower monotonization* as presented in Section 3.1 can no longer be used for proving the correctness of fixed point algorithms. In the following we indicate how we can deal with this more general setting.

Assume thus that we are given a monotone concretization function $\gamma : \mathbb{D} \to \mathbb{C}$ with $\gamma(\top) = Q$ only which provides for each element $d \in \mathbb{D}$ the set $\gamma(d)$ described by d. For a non-empty subset $D \subseteq \mathbb{D}$, we define the γ-closure $\uparrow_\gamma D$ by

$$\uparrow_\gamma D \coloneqq \{\, d \in \mathbb{D} \mid Q_D \subseteq \gamma(d) \,\}$$

where

$$Q_D \coloneqq \bigcap \{\, \gamma(d') \mid d' \in D \,\}$$

Note that, according to our assumption on the value $\gamma(\top)$, $\top$ is always contained in $\uparrow_\gamma D$. We call a non-empty subset $D \subseteq \mathbb{D}$ γ-closed, if $D = \uparrow_\gamma D$. For

$\mathbb{D}$ consider the set $\overline{\mathbb{D}}_\gamma$ consisting of all γ-closed subsets $D \subseteq \mathbb{D}$, i.e., $\overline{\mathbb{D}}_\gamma = \left\{ D \subseteq \mathbb{D} \;\middle|\; D = \uparrow_\gamma D \right\}$. We establish functions $\bar{\alpha} : \mathbb{C} \to \overline{\mathbb{D}}_\gamma$ and $\bar{\gamma} : \overline{\mathbb{D}}_\gamma \to \mathbb{C}$ by

$$\bar{\alpha}(Q) \coloneqq \{\, d \in \mathbb{D} \mid Q \subseteq \gamma(d) \,\} \quad \text{and} \quad \bar{\gamma}(D) \coloneqq Q_D$$

With the ordering $\sqsubseteq_\gamma$ given by $D_1 \sqsubseteq_\gamma D_2$ iff $D_1 \supseteq D_2$, the set $\overline{\mathbb{D}}_\gamma$ is a complete lattice where the least upper bound is given by $\bigsqcup_\gamma \mathcal{D} = \bigcap \mathcal{D}$. For the latter, we need to convince ourselves that $\bigcap \mathcal{D}$ is indeed γ-closed:

$$\begin{aligned}
\bigcap \mathcal{D} &= \{\, d \in \mathbb{D} \mid \forall D \in \mathcal{D}. d \in D \,\} \\
&= \{\, d \in \mathbb{D} \mid \bigcup \{\, Q_D \mid D \in \mathcal{D} \,\} \subseteq \gamma(d) \,\} \\
&= \bar{\alpha}\left(\bigcup \{\, Q_D \mid D \in \mathcal{D} \,\}\right) \\
&\in \overline{\mathbb{D}}_\gamma
\end{aligned}$$

Accordingly, the pair $(\bar{\alpha}, \bar{\gamma})$ forms a Galois connection between $\mathbb{C}$ and $\overline{\mathbb{D}}_\gamma$.

The greatest lower bound operation $\bigsqcap_\gamma$ on $\overline{\mathbb{D}}_\gamma$ exists (as $\overline{\mathbb{D}}_\gamma$ is a complete lattice) but is more complicated than plain union of sets:

$$\bigsqcap_\gamma \mathcal{D} = \uparrow_\gamma (\bigcup \mathcal{D})$$

as γ-closedness is not preserved under union.

Furthermore, we consider the mapping $\iota_\gamma : \mathbb{D} \to \overline{\mathbb{D}}_\gamma$ defined by

$$\begin{aligned}
\iota_\gamma(d) &= \{\, d' \in \mathbb{D} \mid \gamma(d) \subseteq \gamma(d') \,\} \\
&= \uparrow_\gamma \{\, d \,\}
\end{aligned}$$

This mapping is monotone, but not necessarily injective. Still, the following two properties hold:

1. $\gamma(d) = \bar{\gamma}(\iota_\gamma(d))$, and
2. $\bar{\gamma}(D) \subseteq \gamma(d)$ for all $d \in D$.

For a function $f : (Y \to \mathbb{D}) \to \mathbb{D}$, we introduce the *$\gamma$-monotonization* $f_\gamma : (Y \to \overline{\mathbb{D}}_\gamma) \to \overline{\mathbb{D}}_\gamma$ as the substitute of the lower monotonization in case that we are given a concretization function γ only. The function f_γ is defined by

$$\begin{aligned}
f_\gamma\, \sigma &= \bigsqcap_\gamma \left\{\, \iota_\gamma(f\, \sigma') \;\middle|\; \forall y \in Y. \sigma'(y) \in \sigma(y) \,\right\} \\
&= \uparrow_\gamma \left(\bigcup \left\{\, \iota_\gamma(f\, \sigma') \;\middle|\; \forall y \in Y. \sigma'(y) \in \sigma(y) \,\right\}\right) \\
&= \bar{\alpha}\left(\bigcap \{\, \gamma(f\, \sigma') \mid \forall y \in Y. \sigma'(y) \in \sigma(y) \,\}\right)
\end{aligned}$$

We notice that the function f_γ is monotone where for every $\sigma' : Y \to \mathbb{D}$,

$$f_\gamma(\iota_\gamma \circ \sigma') \sqsubseteq_\gamma \iota_\gamma(f\,\sigma')$$

Thus, if f is sound (w.r.t. $\mathcal{R}$ and γ) then so is f_γ (w.r.t. $\mathcal{R}$ and $\bar{\gamma}$). As the γ-monotonization of the system (3.2), we therefore define the system

$$y = f_{\gamma,y}, \qquad y \in Y \tag{3.8}$$

As all right-hand sides of equations in the system (3.8) are monotone, this system has a least solution $\bar{\sigma} : Y \to \overline{\mathbb{D}}_\gamma$ which is sound. Accordingly, any assignment $\sigma : Y \to \mathbb{D}$ so that $\iota_\gamma \circ \sigma$ is a post-solution of the γ-monotonization, i.e., if $\bar{\sigma} \sqsubseteq_\gamma \iota_\gamma \circ \sigma$ holds, is sound as well.

Assume that the binary operators ∇ and Δ on $\mathbb{D}$ satisfy the properties

$$\gamma(a) \cup \gamma(b) \subseteq \gamma(a \nabla b)$$
$$\gamma(a) \cap \gamma(b) \subseteq \gamma(a \Delta b) \subseteq \gamma(a)$$

In particular, for any γ-closed set D, $a, b \in D$ implies that

$$D \sqsubseteq_\gamma \iota_\gamma(a) \sqcap_\gamma \iota_\gamma(b) \sqsubseteq_\gamma \iota_\gamma(a \Delta b) \sqsubseteq_\gamma \iota_\gamma(a)$$

holds. We additionally assume that for all sequences $b_i, i \geq 0$ the sequences a_i, $i \geq 0$ and a'_i, $i \geq 0$, defined by:

$$a_{i+1} = a_i \nabla b_i \quad \text{and} \quad a'_{i+1} = a'_i \Delta b_i$$

both are ultimately stable for all $a_0, a'_0 \in \mathbb{D}$. Note that the stability notion is defined relative to $\mathbb{D}$ (and not relative to $\overline{\mathbb{D}}_\gamma$). Operators with these properties have already been proposed in [Cou15] where they are called *sound* (w.r.t. the concretization γ).

Theorem 3.7.1. *Assume that the local solvers* TSTP *and* TSMP *from Sections 3.5 and 3.6 are equipped with sound widening and narrowing operators. Upon termination, each of them returns a closed partial assignment $\sigma \subseteq Y \to \mathbb{D}$ so that $\iota_\gamma \circ (\underline{\top} \oplus \sigma)$ is a post-solution of system (3.8), i.e., the γ-lower monotonization of system (3.2). Consequently, the mapping $\underline{\top} \oplus \sigma$ is sound (w.r.t. the description relation $\mathcal{R}$ and the concretization γ). Upon termination, each of them returns a closed partial assignment $\sigma : Y \to \mathbb{D}$ so that $\iota_\gamma \circ (\underline{\top} \oplus \sigma)$ is a post-solution of system (3.8), i.e., the γ-monotonization of system (3.2).* □

We remark that at no point during the whole construction we had to refer to the original ordering of $\mathbb{D}$, but instead based all our considerations onto the function γ, the element $\top$ and the operators ∇ and Δ. Therefore, we may drop the assumption for the algorithms that $\mathbb{D}$ is a complete lattice and work with ordinary partial orders, possibly extended with a maximal element $\top$ where the start value $\bot$ may be any element in $\mathbb{D}$.

3.8 Interprocedural Analysis

As seen in example 3.4.1, the concrete semantics of programs with procedures can be formalized by a system of equations over a set of variables $X = \{\langle u, q\rangle \mid u \in U, q \in Q\}$ where U is a finite set of program points and Q is the set of possible system states. A corresponding abstract system of equations for interprocedural analysis can be formalized using abstract variables from the set $Y = \{\langle u, a\rangle \mid u \in U, a \in \mathbb{D}\}$ where the complete lattice $\mathbb{D}$ of abstract values may also serve as the set of abstract calling contexts for which each program point u may be analyzed. The description relation $\mathcal{R}$ between concrete and abstract variables is then given by $\langle u, q\rangle \ \mathcal{R} \ \langle u, a\rangle \iff q \in \gamma(a)$ for all $\langle u, q\rangle \in X$ and $\langle u, a\rangle \in Y$ and program points $u \in U$. Moreover, we require that for all right-hand sides f_x of the concrete system and $f_y^\sharp$ of the abstract system $f_x\, q \subseteq \gamma(f_y^\sharp\, a)$ holds, whenever $x \ \mathcal{R} \ y$ and $q \in \gamma(a)$. Right-hand sides for abstract variables are given by expressions e according to the following grammar:

$$e \quad ::= \quad d \mid \alpha \mid g^\sharp\, e_1 \cdots e_k \mid \langle u, e\rangle \tag{3.9}$$

where $d \in \mathbb{D}$ denotes arbitrary constants, α is a dedicated variable representing the current calling context, $g^\sharp : \mathbb{D} \to \cdots \to \mathbb{D}$ is a k-ary function, and $\langle u, e\rangle$ with $u \in U$ refers to a variable of the equation system. Each expression e describes a function $[\![e]\!]^\sharp : \mathbb{D} \to (Y \to \mathbb{D}) \to \mathbb{D}$ which is defined by:

$$\begin{array}{lcl}
[\![d]\!]^\sharp\, a\, \sigma & = & d \\
[\![\alpha]\!]^\sharp\, a\, \sigma & = & a \\
[\![g^\sharp\, e_1 \cdots e_k]\!]^\sharp\, a\, \sigma & = & g^\sharp\, ([\![e_1]\!]^\sharp\, a\, \sigma) \cdots ([\![e_k]\!]^\sharp\, a\, \sigma) \\
[\![\langle u, e\rangle]\!]^\sharp\, a\, \sigma & = & \sigma\, \langle u, [\![e]\!]^\sharp\, a\, \sigma\rangle
\end{array}$$

A finite representation of the abstract system of equations is then given by the finite set of *schematic* equations

$$\langle u, \alpha\rangle = e_u, \qquad u \in U \tag{3.10}$$

for expressions e_u. Each schematic equation $\langle u, \alpha \rangle = e_u$ denotes the (possibly infinite) family of equations for the variables $\langle u, a \rangle, a \in \mathbb{D}$. For each $a \in \mathbb{D}$, the right-hand-side function of $\langle u, a \rangle$ is given by the function $[\![e_u]\!]^\sharp\, a$. This function is indeed pure for every expression e_u and every $a \in \mathbb{D}$. Such systems of equations have been used, e.g., in [CC77b; ASV12] to specify interprocedural analyses.

Example 3.8.1. Consider the schematic system:

$$\begin{array}{lcl} \langle u, \alpha \rangle & = & \langle v, \langle v, \langle u, \alpha \rangle \rangle \rangle \sqcup \alpha \\ \langle v, \alpha \rangle & = & g^\sharp\, \langle v, \alpha \rangle \sqcup \alpha \end{array}$$

for some unary function $g^\sharp : \mathbb{D} \to \mathbb{D}$. The resulting abstract system simulates the concrete system from Example 3.4.1, if $g(q) \subseteq \gamma(g^\sharp(a))$ holds whenever $q \in \gamma(a)$. ■

As we have seen in the example, function calls result in indirect addressing via nesting of variables. In case that the program does not have recursive procedures, there is a mapping $\lambda : U \to \mathbb{N}$ so that for every u with current calling context α, right-hand side e_u and every subexpression $\langle u', e' \rangle$ of e_u the following holds:

- If $\lambda(u') = \lambda(u)$, then $e' = \alpha$;
- If $\lambda(u') \neq \lambda(u)$, then $\lambda(u') < \lambda(u)$.

If this property is satisfied, we call the equation scheme *stratified* where $\lambda(u)$ is the *level* of u. Intuitively, stratification means that a new context is created only for some point u' of a strictly lower level. For the interprocedural analysis as formalized, e.g., in [ASV12], all program points of a given procedure may receive the same level while the level decreases whenever another procedure is called. The system from Example 3.8.1 is stratified: we may, e.g., define $\lambda(u) = 2$ and $\lambda(v) = 1$.

Theorem 3.8.1. *The solver* TSTP *as well as the solver* TSMP *terminate for stratified equation schemes.*

Proof. We only consider the statement of the theorem for solver TSMP, for the solver TSTP the proof is similar. Assume we run the solver TSMP on an abstract system specified by a stratified equation scheme. In light of Theorem 3.6.1, it suffices to prove that for every $u \in U$, only finitely many contexts $a \in \mathbb{D}$ are encountered during fixed point computation. First, we note

that variables $\langle u, a\rangle$ may not be influenced by variables $\langle v, a'\rangle$ with a higher level, i.e., whenever $\lambda(u) < \lambda(v)$ holds. Second, we have that new contexts for a point u at some level k are created only by evaluation of right-hand sides of variables v of higher levels. Third, since U is finite, the level can decrease only finitely often.

We proceed by induction on k. Assume that k is the least level, i.e., there exists no $v \in U$ such that $\lambda(v) < k$. Then evaluating a call $\mathsf{solve}\,\langle u, a\rangle$ with $\lambda(u) = k$ will query at most other variables $\langle u', a'\rangle$ where $\lambda(u') = k$ and therefore $a = a'$ holds. Hence, no new context is created and therefore only finitely many variables have to be queried.

Now assume that k is not the least level and that we have proven termination for all calls $\mathsf{solve}\,\langle v, b\rangle$, $\lambda(v) < k$. Then evaluating a call $\mathsf{solve}\,\langle u, a\rangle$ with $\lambda(u) = k$ will either query the values of other variables $\langle u', a'\rangle$ where $\lambda(u') = k$ and therefore $a = a'$ holds. Hence, no new context is created and therefore only finitely many variables have to be queried. Or variables $\langle u', a'\rangle$ are queried with $\lambda(u') < k$. For those which have not yet been encountered $\mathsf{solve}\,\langle u', a'\rangle$ is called. By induction hypothesis, all these calls terminate and therefore only finitely many variables are queried. As the evaluation of $\mathsf{solve}\,\langle u, a\rangle$ encounters only finitely many variables, it terminates. □

A similar argument explains why interprocedural analyzers based on the functional approach of Sharir and Pnueli [SP81; AM95] terminate not only for finite domains but also for full constant propagation, if only the programs are non-recursive. However, for interprocedural analysis we often encounter the situation where we are faced with recursive programs and therefore with recursive equation systems, i.e., in particular with non-stratified equation systems. We already observed that local solving of a particular variable does not terminate, if during the computation infintely many contexts and therefore infinetely many variables, have to be considered. In the following we introduce a method in order to decide which variable is abstracted by another one. By a suitable choice of an abstract variable function, this enables use to finitely bound the number of contexts and, therefore, also of variables, which have to be considered during a fixed point iteration. A similar approach has been proposed for the CIAO compiler where widening is performed on the calling contexts [PAH06].

An abstract variable function $\mathsf{absVar} : (Y \rightsquigarrow \mathbb{D}) \to Y \to Y$, where the first parameter is a partial variabel assignment, must fulfill the following properties:

(absVar.1) $\forall \sigma \subseteq Y \to \mathbb{D} : x \mathcal{R} y \implies x \mathcal{R} (\mathsf{absVar}\,\sigma\, y)$

(absVar.2) Let $y_1, y_2, y_3, \ldots$ with $y_k \in Y$ be an arbitrary sequence of variables, and $\sigma_1, \sigma_2, \sigma_3, \ldots$ with $\sigma_k : Y_k \to \mathbb{D}$ where $Y_k \subseteq Y$ be a sequence of partial variable assignments such that $Y_{k+1} = Y_k \cup \{ \mathsf{absVar}\, \sigma_k\, y_k \}$. Then there exists an $i \in \mathbb{N}$ such that $Y_i = Y_{i+1}$.

The first property (absVar.1) ensures that an abstract variable function absVar respects the description relation $\mathcal{R}$. Hence whenever a concrete variable $x \in X$ and an abstract variable $y \in Y$ are in the description relation $\mathcal{R}$, i.e., $x\ \mathcal{R}\ y$ holds, then the abstract variable $y' = \mathsf{absVar}\, \sigma\, y$ is also in relation with x, i.e., $x\ \mathcal{R}\ y'$ holds, for all σ. The second property (absVar.2) ensures that any increasing sequence $Y_1, Y_2, Y_3, \ldots$ with $Y_{k+1} = Y_k \cup \{ \mathsf{absVar}\, \sigma_k\, y_k \}$ is ultimately stable. In particular, if initially Y_1 is a finite subset of Y, then by the property it is ensured that there exists a partial assignment $\sigma : Y' \to \mathbb{D}$ such that $\forall y \in Y.(\mathsf{absVar}\, \sigma\, y) \in Y'$ where Y' is a finite subset of Y. For each such abstract variable function we have:

Lemma 3.8.2. *For any concrete variable assignment $\sigma : X \to \mathbb{C}$ and partial abstract variable assignment $\sigma^\sharp \subseteq Y \to \mathbb{D}$, if $\sigma\ \mathcal{R}^*\ (\underline{\top} \oplus \sigma^\sharp)$ holds, then $\sigma\ \mathcal{R}^*\ ((\underline{\top} \oplus \sigma^\sharp) \circ (\mathsf{absVar}\, \sigma^\sharp))$.* □

We already observed that if indirect addressing is involved, i.e., right-hand sides of the equation system contain expressions of the form $\langle u, g^\sharp\, e_1 \cdots e_k \rangle$ or $\langle u, \langle v, e \rangle \rangle$, then local solving may not terminate since infinitely many variables might be encountered. However, the problem of non-termination might even arise if only direct addressing occurs.

Example 3.8.2. Consider the infinite abstract equation system

$$\begin{array}{lcll} \langle u, i \rangle & = & i + \langle u, i+1 \rangle, & i \in \mathbb{N} \\ \langle u, \infty \rangle & = & 0 & \end{array}$$

over the complete lattice $\mathbb{N} \cup \{\infty\}$ of natural numbers equipped with the natural ordering and extended with ∞ as top element. Although, every context of every variable is statically known, in order to solve any variable $\langle u, k \rangle$, $k \in \mathbb{N}$ infinitely many variables have to be solved. Let $y_i = \langle u, i \rangle$ and $f_i^\sharp = i + y_{i+1}$ be the corresponding right-hand side of the variable y_i. We then have $\mathsf{dep}(f_n^\sharp, \underline{\top} \oplus \sigma^\sharp) = Y \setminus \{ y_1, \ldots, y_n \}$ for all $\sigma^\sharp \subseteq Y \to \mathbb{D}$ and $n \in \mathbb{N}$. ■

In the following we modify the abstract equation system in order to ensure that only finitely many variables have to be solved for any initial query. So far right-hand-side functions $[\![e]\!]^\sharp : \mathbb{D} \to (Y \to \mathbb{D}) \to \mathbb{D}$ of an abstract

equation system where derived from expressions e which are generated by the grammar (3.9). Let us now consider modified right-hand-side functions $[\![\cdot]\!]^\diamond : \mathbb{D} \to (Y \to \mathbb{D}) \to \mathbb{D}$ which incorporate an abstract variable function absVar. An expression of the form $\langle u, e\rangle$ then describes a function $[\![\langle u, e\rangle]\!]^\diamond$ which is defined by:

$$[\![\langle u, e\rangle]\!]^\diamond\, a\, \sigma = \sigma(\mathsf{absVar}\, \sigma\, \langle u, [\![e]\!]^\diamond\, a\, \sigma\rangle)$$

For all other expressions e we have $[\![e]\!]^\diamond = [\![e]\!]^\sharp$. The functions $[\![\cdot]\!]^\diamond$ then give rise to an equation system of the form

$$\langle u, a\rangle = [\![e_u]\!]^\diamond\, a, \qquad (u \in U, a \in \mathbb{D}) \tag{3.11}$$

for which we obtain the following lemma:

Lemma 3.8.3. *Let σ be the least solution of the concrete equation system (3.1) and $\sigma^\sharp$ be a post-solution of the lower monotonization of the abstract equation system (3.11), then $\sigma\ \mathcal{R}^*\ \sigma^\sharp$.* □

Theorem 3.8.4. *For every local solver, assume that $\sigma_k : Y_k \to \mathbb{D}$ is the kth computed partial assignment such that $Y_k \subseteq Y_{k+1} \subseteq Y$ holds for all k. If started for some variable in Y and abstract equation system (3.11), then there exists an $i \in \mathbb{N}$ such that $Y_i = Y_{i+n}$ for all $n \in \mathbb{N}$.*

Proof. By construction of the equation system (3.11) it is ensured that each variable $y \in Y$ occurring in a right-hand side is not queried directly. Instead in iteration i a variable $y' = \mathsf{absVar}\, \sigma_i\, y$ is queried by a local solver such that the resulting assignment $\sigma_{i+1} : Y_{i+1} \to \mathbb{D}$ satisfies $Y_{i+1} = Y_i \cup \{y'\}$. According to property (absVar.2) there exists an $i \in \mathbb{N}$ such that $Y_i = Y_{i+1}$ holds for any variable y'. □

What remains is to give an effective function absVar, which we provide in the following. Let us denote by $\mathsf{dom}(\sigma)$ the domain of a given partial assignment $\sigma \subseteq Y \to \mathbb{D}$. Furthermore, assume we are given a function $t : U \to \mathbb{N}$ which assigns to each program point $u \in U$ a number $n \in \mathbb{N}$ with the additional understanding that the program point u should be analyzed for at most n many contexts precisely.

$$\begin{aligned}
&\textbf{let}\ \mathsf{absVar}\ \sigma\ \langle u, a\rangle = \\
&\quad \textbf{if}\ a \in (\mathsf{ctx}_u\, \sigma)\ \textbf{or}\ \#(\mathsf{ctx}_u\, \sigma) \leq t(u)\ \textbf{then}\ \langle u, a\rangle \\
&\quad \textbf{else if}\ \exists b \in (\mathsf{ctx}_u\, \sigma).a \sqsubseteq b\ \textbf{then}\ \langle u, b\rangle \\
&\quad \textbf{else let}\ d = \bigsqcup(\mathsf{ctx}_u\, \sigma)\ \textbf{in}\ \langle u, d \triangledown a\rangle \\
&\\
&\textbf{let}\ \mathsf{ctx}_u\ \sigma = \{\, a \mid \langle u, a\rangle \in \mathsf{dom}(\sigma)\,\}
\end{aligned} \tag{3.12}$$

Lemma 3.8.5. *The function* absVar *from* (3.12) *fulfills the properties* (absVar.1) *and* (absVar.2)*.*

Proof. We first prove that absVar fulfills property (absVar.1). Let $\sigma \subseteq Y \to \mathbb{D}$ and $y' = \langle u, b\rangle = \mathsf{absVar}\,\sigma\,y$ for some $y = \langle u, a\rangle \in Y$. Furthermore, assume that $x \mathcal{R} y$ holds for some $x = \langle u, q\rangle \in X$. We then have that $a \sqsubseteq b$ holds. Since $a \in \gamma(a)$ and $\gamma(a) \subseteq \gamma(b)$, we conclude that $x \mathcal{R} y'$ holds proving the first assertion of the lemma.

We prove that absVar fulfills property (absVar.2). Let $\sigma : Y_i \to \mathbb{D}$ with $Y_i \subseteq Y$ and $y' = \mathsf{absVar}\,\sigma\,y$ for some $y = \langle u, a\rangle \in Y$. We have that $y' \notin Y_i$ if and only if there exists no $\langle u, b\rangle \in Y_i$ with $a \sqsubseteq b$. In that case $y' = \langle u, (\bigsqcup \mathsf{ctx}_u(\sigma)) \triangledown a\rangle$. By widening we conclude that there exists an i such that $y' \in Y_i$. Therefore, we consider only finitely many contexts for each program point and since we only consider finitely many program points the assertion of the lemma follows. □

In total we have the following result for our local solvers TSTP and TSMP.

Theorem 3.8.6. *The solvers* TSTP *and* TSMP *terminate for every abstract equation system of the form* (3.11) *equipped with the function* absVar *from* (3.12)*. The computed partial assignments are sound.* □

3.9 Summary

We have presented local solvers which are guaranteed to terminate for all abstract systems of equations whenever only finitely many variables are encountered—irrespective of whether right-hand sides of the equations are monotone or not or whether the complete lattice has infinite strictly ascending/descending chains or not. Furthermore, we showed that interprocedural analysis with partial tabulation of procedure summaries based on these solvers is guaranteed to terminate firstly for non-recursive programs and generalized it afterwards even for recursive programs. Clearly, theoretical termination proofs may only give an indication that the proposed algorithms are well-suited as fixed point engines within a practical analysis tool. Termination within reasonable time and space bounds is another issue. Some practical experiments within the analysis framework GOBLINT have been provided in [SSV17] which seem encouraging. Interestingly, a direct comparison of the two-phase versus mixed-phase solver for full context-sensitive interprocedural analysis, indicated that TSMP was virtually always faster, while the picture w.r.t. precision

is not so clear. Also, the new solvers always returned post-solutions of the abstract systems—although they are not bound to do so.

In case that widening and narrowing were defined relative to the ordering on $\mathbb{D}$, our correctness proofs required a Galois connection to relate the concrete with the abstract domain. In Section 3.7, however, we showed that our algorithms remain correct when widening and narrowing operators are used which are sound w.r.t. the concretization. In this case, the requirement of a Galois connection can be dropped.

4 Conclusion

In this thesis we covered two aspects of interprocedural static program analysis. First we explored Herbrand equalities and introduced a novel analysis which is based on procedure summaries. In the second part of this thesis we then introduced two novel local solvers which can be used in the general setting of abstract interpretation. We also gave explicit examples of interprocedural analyses in Section 3.8 where we made use of partial tabulation of procedure summaries in order to demonstrate the strengths of the presented local solvers.

4.1 Contributions

In the field of *Herbrand equalities* we extended the state of the art analyses in order to infer all interprocedurally valid Herbrand equalities for programs where all assignments are taken into account whose right-hand sides depend on at most one variable. The novel analysis is based on the following core contributions:

- An extension of the analysis presented in [GT07] where programs must contain only unary operators to programs with operators of any arity—as long as at most one variable occurs (possibly multiple times) in a term. This extension can be seen as the non-trivial complication derived from dealing with terms instead of words.
- A method to uniquely factorize all possibly occurring run-time values into tree patterns—except finitely many.
- A notion of approximate subsumption which is decidable and which still guarantees that every occurring conjunction of equalities is effectively equivalent to a finite conjunction.
- An in-depth analysis of the complexity showing that for initialization-restricted programs a succinct representation of all interprocedurally valid two-variable Herbrand equalities can be computed in polynomial time. For unrestricted programs we had that a two-variable invariant

candidate $\mathbf{x} \doteq t\mathbf{y}$ can be verified in time polynomial in the size of the program and in the size of t. Additionally for unrestricted programs a multi-variable invariant candidate $\mathbf{x} \doteq t$ where $t \in \mathcal{T}_\Omega(\mathbf{X})$ can be verified in time polynomial in the size of the program and in the size of t and only exponentially in the number of variables occurring in t.

In the second part of this thesis two novel *local solvers* TSTP and TSMP were provided in order to compute (partial) solutions of equation systems. At first the solver TSTP was introduced which separates widening and narrowing into different phases. Subsequently the solver TSMP was presented which breaks up with this approach and intertwines widening and narrowing into one phase, similarly as it is done for the solver SLR_3. However, in contrast to SLR_3 the solver TSMP introduces extra logic in order to decide when widening or narrowing has to be used. The following core contributions were provided:

- Termination guarantees for every system of equations—irrespective if occurring right-hand sides are monotone or not—were given for both local solvers, if only finitely many variables are encountered during a run.
- For stratified equation systems, which may arise for interprocedural analysis of non-recursive programs, we showed that at most finitely many variables are encountered during a run and, therefore, both solvers always terminate.
- For the general case an abstract variable function was introduced which ensures that at most finitely many variables are encountered during any run. That means, by over-approximating calling contexts for each procedure, the number of possible contexts is kept finite. Hence, even for recursive programs, termination is guaranteed.
- Soundness arguments were given for both local solvers, i.e., upon termination the computed partial solutions of both solvers are sound w.r.t. description relation $\mathcal{R}$ and the concretization γ. At first we required a Galois connection and subsequently dropped this requirement, i.e., we considered the case when no abstraction function α is available.

4.2 Future work

For the case of computing all valid two-variable *Herbrand equalities* we made use of conjunctions of equalities of the form $As \doteq Bt$ where s and t are terms

containing at most one program variable. The presented algorithm to decide subsumption heavily depends on the unique factorizations of terms. However, as observed in Section 2.11.2 for unrestricted programs, a solution for A or B might not reflect the unique factorization of s or t at all, i.e., these terms can be in the set $C_\Omega \setminus M_G$. Hence, in order to compute all solutions or to verify an invariant candidate a succinctly represented term might very well be made explicit. Therefore, for unrestricted programs we were only able to verify an invariant candidate in time polynomial in the size of the program and in the size of the invariant candidate. It might be worthwhile to reconsider the second-order unification problem for these particular equalities. In case that satisfiability is decidable and finally also subsumption, then the terms need not be represented by their unique factorizations. By a suitable compact representation of such terms it might be possible to compute a succinct representation of all valid two-variable Herbrand equalities in time polynomial in the size of the program. The succinct representation might be more natural than the representation of the invariants by a conjunction of equalities—as it is the case now—or even more natural than the program itself.

Ultimately it is still an open question how to infer *all* valid Herbrand equalities of programs where right-hand sides contain arbitrarily many program variables.

For the two novel *local solvers* TSTP and TSMP it remains open to extend them in a way which allows restarting as proposed in [Cou15]. That means, once a solution is computed, the solvers start all over again but this time in each widening step the computed result is upper bounded by the previous known solution. This could result in more precise solutions by not overshooting the least/greatest fixed point too much in the widening step which is hard or even impossible to recover in the narrowing step.

Finally in Section 3.8 we presented the idea of a function absVar in order to keep the number of variables of an equation system for which a solution must be computed finite. Instead of using the function for every variable look-up it might be of interest to only use this function for variables which correspond to recursive procedures. By that loss of precision would be introduced only for recursive procedures, while it should still be guaranteed that only finitely many variables are encountered during any run. This would mean that termination would be guaranteed even for equation systems corresponding to recursive programs.

List of Figures

List of Tables

Bibliography

[All74] Frances Elizabeth Allen
"Interprocedural Data Flow Analysis"
In: *Proceedings of the IFIP Congress 74*. Ed. by Jack L. Rosenfeld. North-Holland, 1974, pp. 398–402. ISBN: 9780720428032 (cit. on p. 3).

[AM95] Martin Alt and Florian Martin
"Generation of Efficient Interprocedural Analyzers with PAG"
In: *Proceedings of the 2nd International Symposium on Static Analysis, SAS*. Ed. by Alan Mycroft. Vol. 983. Lecture Notes in Computer Science. Springer, 1995, pp. 33–50. DOI: 10.1007/3-540-60360-3_31 (cit. on p. 101).

[Ama+16] Gianluca Amato et al.
"Efficiently intertwining widening and narrowing"
In: *Science of Computer Programming* 120 (2016), pp. 1–24. DOI: 10.1016/j.scico.2015.12.005 (cit. on pp. 7, 71, 72, 77, 96).

[ASV12] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani
"Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis"
In: *Proceedings of the 10th Asian Symposium on Programming Languages and Systems, APLAS*. Ed. by Ranjit Jhala and Atsushi Igarashi. Vol. 7705. Lecture Notes in Computer Science. Springer, 2012, pp. 157–172. DOI: 10.1007/978-3-642-35182-2_12 (cit. on pp. 7, 8, 71, 73, 100).

[ASV13] Kalmer Apinis, Helmut Seidl, and Vesal 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*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013,

pp. 377–386. DOI: 10.1145/2491956.2462190
(cit. on pp. 7, 71, 72, 77, 78, 80, 96).

[ASV16] Kalmer Apinis, Helmut Seidl, and Vesal 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 Christian W. Probst, Chris Hankin, and René Rydhof Hansen. Vol. 9560.
Lecture Notes in Computer Science. Springer, 2016, pp. 272–288.
DOI: 10.1007/978-3-319-27810-0_14 (cit. on pp. 71, 77).

[Bag+05] Roberto Bagnara et al.
"Precise widening operators for convex polyhedra"
In: *Science of Computer Programming* 58.1-2 (2005), pp. 28–56.
DOI: 10.1016/j.scico.2005.02.003 (cit. on p. 71).

[Bar77] Jeffrey M. Barth
"An Interprocedural Data Flow Analysis Algorithm"
In: *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*.
Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi.
ACM, 1977, pp. 119–131. DOI: 10.1145/512950.512962
(cit. on p. 3).

[Bou93] François Bourdoncle
"Efficient chaotic iteration strategies with widenings"
In: *Proceedings of the International Conference on Formal Methods in Programming and Their Applications*.
Ed. by Dines Bjørner, Manfred Broy, and Igor V. Pottosin. Vol. 735.
Lecture Notes in Computer Science. Springer, 1993, pp. 128–141.
DOI: 10.1007/BFb0039704 (cit. on p. 84).

[CC77a] Patrick Cousot and Radhia Cousot
"Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints"
In: *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*.
Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi.
ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973
(cit. on pp. 6, 71, 73, 74, 76).

[CC77b] Patrick Cousot and Radhia Cousot
"Static Determination of Dynamic Properties of Generalized Type Unions"
In: *Proceedings of an ACM Conference on Language Design for Reliable Software, LDRS*. Ed. by David B. Wortman. ACM, 1977, pp. 77–94. DOI: 10.1145/800022.808314 (cit. on p. 100).

[CC77c] Patrick Cousot and Radhia Cousot
"Static Determination of Dynamic Properties of Recursive Procedures"
In: *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*. Ed. by Erich J. Neuhold. North-Holland, 1977, pp. 237–277. ISBN: 9780444851079 (cit. on pp. 3, 6, 8, 73).

[CC92] Patrick Cousot and Radhia Cousot
"Abstract Interpretation Frameworks"
In: *Journal of Logic and Computation* 2.4 (1992), pp. 511–547. DOI: 10.1093/logcom/2.4.511 (cit. on pp. 73, 74, 77, 96).

[Che+10] Liqian Chen et al.
"An Abstract Domain to Discover Interval Linear Equalities"
In: *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI*. Ed. by Gilles Barthe and Manuel V. Hermenegildo. Vol. 5944. Lecture Notes in Computer Science. Springer, 2010, pp. 112–128. DOI: 10.1007/978-3-642-11319-2_11 (cit. on p. 71).

[Cou+09] Patrick Cousot et al.
"Why does Astrée scale up?"
In: *Formal Methods in System Design* 35.3 (2009), pp. 229–264. DOI: 10.1007/s10703-009-0089-6 (cit. on p. 77).

[Cou15] Patrick Cousot
"Abstracting Induction by Extrapolation and Interpolation"
In: *Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI*. Ed. by Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen. Vol. 8931. Lecture Notes in Computer Science. Springer, 2015, pp. 19–42. DOI: 10.1007/978-3-662-46081-8_2 (cit. on pp. 77, 98, 109).

[Cou90] Patrick Cousot
"Methods and Logics for Proving Programs"
In: *Formal Models and Semantics*. Ed. by Jan van Leeuwen. Vol. B. Handbook of Theoretical Computer Science. Elsevier, 1990. Chap. 15, pp. 843–993. ISBN: 9780444880741 (cit. on p. 17).

[CS70] John Cocke and Jacob T. Schwartz
Programming Languages and Their Compilers: Preliminary Notes
Courant Institute of Mathematical Sciences, New York University, 1970 (cit. on p. 9).

[Dij75] Edsger W. Dijkstra
"Guarded Commands, Nondeterminacy and Formal Derivation of Programs"
In: *Communications of the ACM* 18.8 (1975), pp. 453–457.
DOI: 10.1145/360933.360975 (cit. on p. 3).

[Eng80] Joost Engelfriet
"Some open questions and recent results on tree transducers and tree languages"
In: *Formal Language Theory: Perspectives and Open Problems*. Ed. by Ronald V. Book. Academic Press, 1980, pp. 241–286.
ISBN: 9780121153502.
DOI: 10.1016/B978-0-12-115350-2.50014-2 (cit. on p. 20).

[Fle+11] Andrea Flexeder et al.
"Fast Interprocedural Linear Two-Variable Equalities"
In: *ACM Transactions on Programming Languages and Systems, TOPLAS* 33.6 (2011), 21:1–21:33.
DOI: 10.1145/2049706.2049710 (cit. on pp. 10, 15).

[Flo67] Robert W. Floyd
"Assigning Meanings to Programs"
In: *Proceedings of a Symposium on Applied Mathematics*. Ed. by Jacob T. Schwartz. Vol. 19. Mathematical Aspects of Computer Science. American Mathematical Society. 1967, pp. 19–31.
ISBN: 9780821867280 (cit. on p. 2).

[FS96] Christian Fecht and Helmut Seidl
"An Even Faster Solver for General Systems of Equations"
In: *Proceedings of the 3rd International Symposium on Static Analysis, SAS*. Ed. by Radhia Cousot and David A. Schmidt.

Vol. 1145. Lecture Notes in Computer Science. Springer, 1996, pp. 189–204. DOI: 10.1007/3-540-61739-6_42 (cit. on p. 71).

[Gal78] Jean H. Gallier
"Semantics and Correctness of Nondeterministic Flowchart Programs with Recursive Procedures"
In: *Proceedings of the 5th International Colloquium on Automata, Languages and Programming, ICALP*.
Ed. by Giorgio Ausiello and Corrado Böhm. Vol. 62.
Lecture Notes in Computer Science. Springer, 1978, pp. 251–267.
DOI: 10.1007/3-540-08860-1_19 (cit. on p. 3).

[GH06] Laure Gonnord and Nicolas Halbwachs
"Combining Widening and Acceleration in Linear Relation Analysis"
In: *Proceedings of the 13th International Symposium on Static Analysis, SAS*. Ed. by Kwangkeun Yi. Vol. 4134.
Lecture Notes in Computer Science. Springer, 2006, pp. 144–160.
DOI: 10.1007/11823230_10 (cit. on p. 71).

[GN04] Sumit Gulwani and George C. Necula
"A Polynomial-Time Algorithm for Global Value Numbering"
In: *Proceedings of the 11th International Symposium on Static Analysis, SAS*. Ed. by Roberto Giacobazzi. Vol. 3148.
Lecture Notes in Computer Science. Springer, 2004, pp. 212–227.
DOI: 10.1007/978-3-540-27864-1_17 (cit. on p. 9).

[Gol81] Warren David Goldfarb
"The undecidability of the second-order unification problem"
In: *Theoretical Computer Science* 13.2 (1981), pp. 225–230.
DOI: 10.1016/0304-3975(81)90040-2 (cit. on p. 11).

[GT07] Sumit Gulwani and Ashish Tiwari
"Computing Procedure Summaries for Interprocedural Analysis"
In: *Proceedings of the 16th European Symposium on Programming, ESOP*. Ed. by Rocco De Nicola. Vol. 4421.
Lecture Notes in Computer Science. Springer, 2007, pp. 253–267.
DOI: 10.1007/978-3-540-71316-6_18
(cit. on pp. 9, 12, 13, 15, 25, 28, 29, 49, 69, 107).

[GT09] Guillem Godoy and Ashish Tiwari
"Invariant Checking for Programs with Procedure Calls"
In: *Proceedings of the 16th International Symposium on Static*

Analysis, SAS. Ed. by Jens Palsberg and Zhendong Su. Vol. 5673. Lecture Notes in Computer Science. Springer, 2009, pp. 326–342. DOI: 10.1007/978-3-642-03237-0_22 (cit. on p. 10).

[Her+05] Manuel V. Hermenegildo et al.
"Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)"
In: *Science of Computer Programming* 58.1-2 (2005), pp. 115–140. DOI: 10.1016/j.scico.2005.02.006 (cit. on pp. 7, 80).

[Her+12] Manuel V. Hermenegildo et al.
"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 (cit. on p. 80).

[HJM96] Yoram Hirshfeld, Mark Jerrum, and Faron Moller
"A polynomial algorithm for deciding bisimilarity of normed context-free processes"
In: *Theoretical Computer Science* 158.1&2 (1996), pp. 143–159. DOI: 10.1016/0304-3975(95)00064-X (cit. on p. 51).

[HKS10a] Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl
"Verifying a Local Generic Solver in Coq"
In: *Proceedings of the 17th International Symposium on Static Analysis, SAS*. Ed. by Radhia Cousot and Matthieu Martel. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, pp. 340–355. DOI: 10.1007/978-3-642-15769-1_21 (cit. on p. 82).

[HKS10b] Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl
"What is a Pure Functional?"
In: *Proceedings of the 37th International Colloquium on Automata, Languages and Programming, ICALP*.
Ed. by Samson Abramsky et al. Vol. 6199.
Lecture Notes in Computer Science. Springer, 2010, pp. 199–210. DOI: 10.1007/978-3-642-14162-1_17 (cit. on p. 82).

[Hoa69] Charles Antony Richard Hoare
"An Axiomatic Basis for Computer Programming"
In: *Communications of the ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259 (cit. on pp. 2, 17).

[Jeż14] Artur Jeż
"Context Unification is in PSPACE"
In: *Proceedings of the 41st International Colloquium on Automata, Languages and Programming, ICALP*. Ed. by Javier Esparza et al. Vol. 8573. Lecture Notes in Computer Science. Springer, 2014, pp. 244–255. DOI: 10.1007/978-3-662-43951-7_21 (cit. on p. 11).

[Kar13] Aleksandr Karbyshev
"Monadic Parametricity of Second-Order Functionals"
PhD thesis. Technische Universität München, 2013.
URL: http://mediatum.ub.tum.de/?id=1144371 (cit. on p. 82).

[Kil73] Gary Arlen Kildall
"A Unified Approach to Global Program Optimization"
In: *Proceedings of the 1st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*.
Ed. by Patrick C. Fischer and Jeffrey D. Ullman. ACM, 1973, pp. 194–206. DOI: 10.1145/512927.512945 (cit. on pp. 9, 48).

[Loh12] Markus Lohrey
"Algorithmics on SLP-compressed strings: A survey"
In: *Groups Complexity Cryptology* 4.2 (2012), pp. 241–299.
DOI: 10.1515/gcc-2012-0016 (cit. on p. 50).

[Loh15] Markus Lohrey
"Grammar-Based Tree Compression"
In: *Proceedings of the 19th International Conference on Developments in Language Theory, DLT*. Ed. by Igor Potapov. Vol. 9168. Lecture Notes in Computer Science. Springer, 2015, pp. 46–57. DOI: 10.1007/978-3-319-21500-6_3 (cit. on p. 49).

[Lom77] David B. Lomet
"Data Flow Analysis in the Presence of Procedure Calls"
In: *IBM Journal of Research and Development* 21.6 (1977), pp. 559–571. DOI: 10.1147/rd.216.0559 (cit. on p. 3).

[LV00] Jordi Levy and Margus Veanes
"On the Undecidability of Second-Order Unification"
In: *Information and Computation* 159.1-2 (2000), pp. 125–150.
DOI: 10.1006/inco.2000.2877 (cit. on p. 11).

[Mac37] Holbrook Mann MacNeille
"Partially ordered sets"
In: *Transactions of the American Mathematical Society* 42.3 (1937), pp. 416–460.
DOI: 10.1090/S0002-9947-1937-1501929-X (cit. on p. 73).

[MH90] Kalyan Muthukumar and Manuel V. Hermenegildo
Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs
Technical Report ACT-DC-153-90. Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, Apr. 1990 (cit. on p. 81).

[Mol08] Richard A. Mollin
Fundamental Number Theory with Applications
Second. Chapman & Hall/CRC, 2008. ISBN: 9781420066593 (cit. on p. 59).

[MPS06] Markus Müller-Olm, Michael Petter, and Helmut Seidl
"Interprocedurally Analyzing Polynomial Identities"
In: *Proceedings of the 23rd Symposium on Theoretical Aspects of Computer Science, STACS*.
Ed. by Bruno Durand and Wolfgang Thomas. Vol. 3884.
Lecture Notes in Computer Science. Springer, 2006, pp. 50–67.
DOI: 10.1007/11672142_3 (cit. on p. 15).

[MS04] Markus Müller-Olm and Helmut Seidl
"Precise Interprocedural Analysis through Linear Algebra"
In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*.
Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 330–341.
DOI: 10.1145/982962.964029 (cit. on pp. 10, 15).

[MS07] Markus Müller-Olm and Helmut Seidl
"Analysis of Modular Arithmetic"
In: *ACM Transactions on Programming Languages and Systems, TOPLAS* 29.5 (2007), 29:1–29:27.
DOI: 10.1145/1275497.1275504 (cit. on p. 10).

[MS08] Markus Müller-Olm and Helmut Seidl
"Upper Adjoints for Fast Inter-procedural Variable Equalities"
In: *Proceedings of the 17th European Symposium on Programming, ESOP*. Ed. by Sophia Drossopoulou. Vol. 4960.

Lecture Notes in Computer Science. Springer, 2008, pp. 178–192. DOI: 10.1007/978-3-540-78739-6_15 (cit. on pp. 10, 15).

[MSS05] Markus Müller-Olm, Helmut Seidl, and Bernhard Steffen
"Interprocedural Herbrand Equalities"
In: *Proceedings of the 14th European Symposium on Programming, ESOP*. Ed. by Shmuel Sagiv. Vol. 3444.
Lecture Notes in Computer Science. Springer, 2005, pp. 31–45.
DOI: 10.1007/978-3-540-31987-0_4 (cit. on p. 9).

[MSU97] Kurt Mehlhorn, R. Sundar, and Christian Uhrig
"Maintaining Dynamic Sequences Under Equality-Tests in Polylogarithmic Time"
In: *Algorithmica* 17.2 (1997), pp. 183–198.
DOI: 10.1007/BF02522825 (cit. on p. 51).

[PAH06] Germán Puebla, Elvira Albert, and Manuel Hermenegildo
"Abstract Interpretation with Specialized Definitions"
In: *Proceedings of the 13th International Symposium on Static Analysis, SAS*. Ed. by Kwangkeun Yi. Vol. 4134.
Lecture Notes in Computer Science. Springer, 2006, pp. 107–126.
DOI: 10.1007/11823230_8 (cit. on pp. 8, 101).

[Pet10] Michael Petter
"Interprocedural Polynomial Invariants"
PhD thesis. Technische Universität München, 2010.
URL: http://mediatum.ub.tum.de/?id=956851
(cit. on pp. 9, 14, 15).

[Pla94] Wojciech Plandowski
"Testing Equivalence of Morphisms on Context-Free Languages"
In: *Proceedings of the 2nd European Symposium on Algorithms, ESA*. Ed. by Jan van Leeuwen. Vol. 855.
Lecture Notes in Computer Science. Springer, 1994, pp. 460–470.
DOI: 10.1007/BFb0049431 (cit. on p. 51).

[Ric53] Henry Gordon Rice
"Classes of Recursively Enumerable Sets and Their Decision Problems"
In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. DOI: 10.2307/1990888 (cit. on p. 1).

[Ros79] Barry K. Rosen
"Data Flow Analysis for Procedural Languages"
In: *Journal of the ACM* 26.2 (1979), pp. 322–344.
DOI: 10.1145/322123.322135 (cit. on p. 3).

[Sch13] Manfred Schmidt-Schauß
"Linear Compressed Pattern Matching for Polynomial Rewriting (Extended Abstract)"
In: *Proceedings of the 7th International Workshop on Computing with Terms and Graphs, TERMGRAPH*.
Ed. by Rachid Echahed and Detlef Plump. Vol. 110. EPTCS. 2013, pp. 29–40. DOI: 10.4204/EPTCS.110.5 (cit. on p. 49).

[SF00] Helmut Seidl and Christian Fecht
"Interprocedural Analyses: A Comparison"
In: *Journal of Logic Programming* 43.2 (2000), pp. 123–156.
DOI: 10.1016/S0743-1066(99)00058-8 (cit. on p. 81).

[SKR90] Bernhard Steffen, Jens Knoop, and Oliver Rüthing
"The Value Flow Graph: A Program Representation for Optimal Program Transformations"
In: *Proceedings of the 3rd European Symposium on Programming, ESOP*. Ed. by Neil D. Jones. Vol. 432.
Lecture Notes in Computer Science. Springer, 1990, pp. 389–405.
DOI: 10.1007/3-540-52592-0_76 (cit. on p. 9).

[SP81] Micha Sharir and Amir Pnueli
"Two Approaches to Interprocedural Data Flow Analysis"
In: *Program Flow Analysis: Theory and Application*.
Ed. by Steven S. Muchnick and Neil D. Jones.
Prentice Hall Professional Technical Reference, 1981,
pp. 189–233. ISBN: 9780137296811 (cit. on pp. 3, 14, 101).

[SSV17] Stefan Schulze Frielinghaus, Helmut Seidl, and Ralf Vogler
"Enforcing Termination of Interprocedural Analysis"
In: *Formal Methods in System Design* (2017).
DOI: 10.1007/s10703-017-0288-5 (cit. on p. 104).

[SWH12] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack
Compiler Design: Analysis and Transformation
Springer, 2012. ISBN: 9783642175473.
DOI: 10.1007/978-3-642-17548-0 (cit. on pp. 46, 48).

[Tur37] Alan Mathison Turing
"On Computable Numbers, with an Application to the Entscheidungsproblem"
In: *Proceedings of the London Mathematical Society*. 2nd ser. 42.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230 (cit. on p. 1).

[Voj+16] Vesal Vojdani et al.
"Static Race Detection for Device Drivers: The Goblint Approach"
In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE*. ACM, 2016, pp. 391–402. DOI: 10.1145/2970276.2970337 (cit. on p. 80).