On the Complexity of Constant Propagation

Markus Müller-Olm and Oliver Rüthing

Universität Dortmund, FB Informatik, LS V,
D-44221 Dortmund, Germany
{mmo,ruething}@ls5.cs.uni-dortmund.de

Abstract. *Constant propagation* (*CP*) is one of the most widely used optimizations in practice (cf. [9]). Intuitively, it addresses the problem of statically detecting whether an expression always evaluates to a unique constant at run-time. Unfortunately, as proved by different authors [4, 16], CP is in general undecidable even if the interpretation of branches is completely ignored. On the other hand, it is certainly decidable in more restricted settings, like on loop-free programs (cf. [7]). In this paper, we explore the complexity of CP for a three-dimensional taxonomy. We present an almost complete complexity classification, leaving only two upper bounds open.

1 Motivation

Constant propagation (*CP*) is one of the most widely used optimizations in practice (cf. [1, 4, 9]). Intuitively, it aims at detecting expressions that always yield a unique constant value at run-time. Unfortunately, the constant propagation problem is undecidable even if the interpretation of branches is completely ignored, like in the common model of nondeterministic flow graphs where every program path is considered executable. Independent proofs of this important observation have been given by Hecht [4] and by Reif and Lewis [16]. We briefly recall the construction of Hecht, which is based on the Post correspondence problem. A Post correspondence system consists of a set of pairs $(u_1, v_1), \ldots, (u_k, v_k)$ with $u_i, v_i \in \{0, 1\}^*$. The correspondence system has a solution, iff there is a sequence $i_1, \ldots, i_n$ such that $u_{i_1} \cdot \ldots \cdot u_{i_n} = v_{i_1} \cdot \ldots \cdot v_{i_n}$. Figure 1 illustrates Hecht's reduction. The variables x and y are used as decimal numbers representing strings in $\{0, 1\}^*$. For each pair of the correspondence system a distinct branch of the loop appends the strings u_i and v_i to x and y, respectively.[1] It is easy to see that $x - y$ always evaluates to a value different from 0, if the Post correspondence problem has no solution.[2] In this case the expression 1 `div` $((x-y)^2+1)$ always evaluates to 0. But if the Post correspondence system is solvable, this expression can evaluate to 1. Thus, r is constant (with value 0), if and only if the Post correspondence problem is not solvable. To exclude

[1] Technically, this is achieved by shifting the digits of x and y by $lg(u_i)$ and $lg(v_i)$ places first, where $lg(u_i)$ and $lg(v_i)$ are the length of the decimal representation of u_i and v_i, respectively.

[2] Note that the initialization of x and y with 1 avoids a problem with leading zeros.

D. Sands (Eds.): ESOP 2001, LNCS 2028, pp. 190–205, 2001.

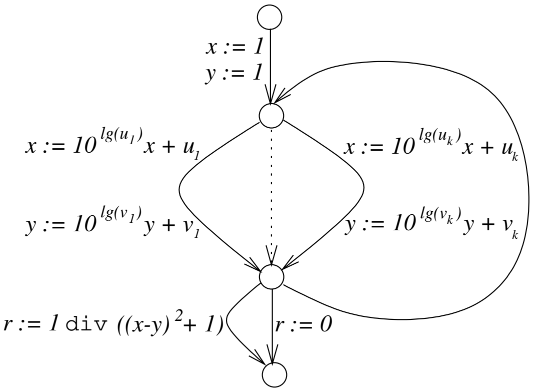

Fig. 1. Undecidability of CP: reduction of the Post correspondence problem.

r from being constantly 1 in the case that the Post correspondence system is universally solvable, r is set to 0 by a bypassing assignment statement.

On the other hand, constant propagation is certainly decidable for *acyclic*, i.e., loop-free, programs. But even in this setting the problem is intractable, as it has been shown to be co-NP-hard [7] recently . This result is based on a polynomial time reduction of the co-problem of 3-SAT, the *satisfiability* problem for clauses which are conjunctions consisting of three negated or unnegated Boolean variables (cf. [3]). An instance of 3-SAT is solvable if there is a variable assignment such that every clause is satisfied.

The reduction is illustrated in Figure 2 for a 3-SAT instance over the Boolean variables $\{b_1, \ldots, b_k\}$:

$$\underbrace{(b_3 \vee \overline{b_5} \vee b_6)}_{c_1} \wedge \ldots \wedge \underbrace{(b_2 \vee \overline{b_3} \vee b_5)}_{c_n}.$$

For each Boolean variable b_i two integer variables x_i and $\overline{x_i}$ are introduced that are initialized by 0. The idea underlying the reduction is the following: each path of the program chooses a witnessing literal in each clause by setting the corresponding variable to 1. If this can be done without setting both x_i and $\overline{x_i}$ for some i then we have found a satisfying truth assignment, and vice versa. On such a path $r1$ and consequently $r2$ evaluate to 0. On all other paths the value of $r1$ differs from 0 but stays in the range $\{1, \ldots, k\}$ enforcing that variable $r2$ is set to 1. Summarizing, $r2$ evaluates to 1 on every program path if and only if the underlying instance of 3-SAT has no solution. Similarly to the undecidability reduction of Figure 1 the assignment $r1 := 1$ avoids that $r1$ is constantly 0 in the case that all runs induce satisfying truth assignments.

Note that both reductions presented so far crucially depend on an operator like integer division (or modulo) which is capable of projecting many different values onto a single one.

Contributions This paper aims at examining the borderline of intractability and undecidability more closely. To this end, we investigate the constant propa-

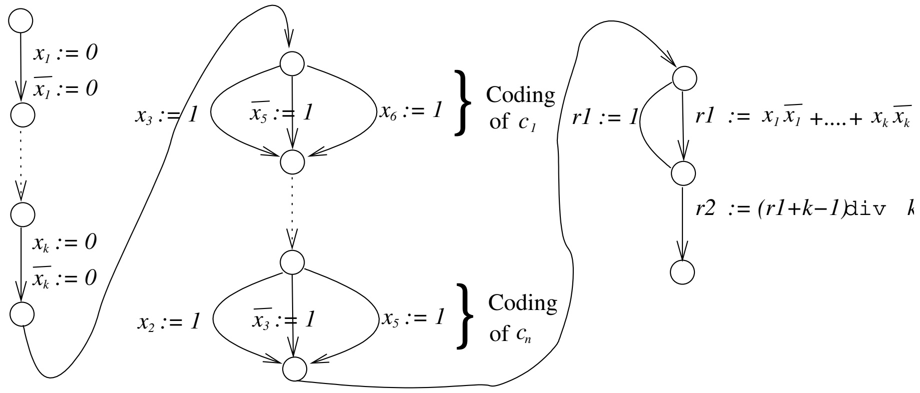

Fig. 2. Co-NP-hardness of CP for acyclic programs: reduction of co-3-SAT.

gation problem for integers with respect to a three-dimensional taxonomy. The first dimension is given by the distinction between arbitrary and loop-free programs. We are currently also examining further extensions of this dimension towards interprocedural and explicitly parallel programs (for first results see [11]). However, this is beyond the scope of this paper whose focus is more directed towards examining the influences of the other two dimensions

The second dimension is concerned with the underlying signature. We consider signatures without operators (*copy-constants*), with linear expressions $x := ay + b$ (*linear constants*), with operators restricted to the set $\{+, -\}$ (Presburger constants), operators restricted to $\{+, -, *\}$ ($+, -, *$-constants), and the standard signature, i.e., the one with operators $+, -, *, \mathtt{div}, \mathtt{mod}$. Finally, in the third dimension we investigate the general nature of the constant propagation problem. Besides the standard *must-constancy* problem we also consider the less frequently addressed problem of *may-constancy* here. Essentially, this problem asks if a variable may evaluate to a given constant c on some program path. Inspired by the work of Muth and Debray [12] we further distinguish between a *single value* and a *multiple value* variant, where in the latter case the values of multiple variables might be checked simultaneously.[3]

While the most prominent application of must-CP is the *compile-time simplification* of expressions, the must- and may-variants are equally well suited for eliminating unnecessary branches in programs. Furthermore, the may-variant reveals some interesting insight in the complexity of (may-)aliasing of array elements.

In this paper, we present an almost complete complexity classification, providing all hardness results, i.e., lower bounds, leaving only two upper bounds open. In particular, we observe that detecting may-constants is significantly harder than detecting their must-counterparts. Furthermore, we demonstrate that Presburger must-constants are polynomial time detectable which is some-

[3] Muth and Debray introduced the single and multiple value variants as models for independent-attribute and relational-attribute data flow analyses [5].

how surprising, as non-distributivity in the standard setting already shows up for this class. Finally, as a by-product we obtain some interesting results on the decidability of may-aliasing of arrays.

2 The Setting

Flow Graphs. As usual in data-flow analysis and program optimization, we represent programs by *directed flow graphs* $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N, edge set E, a unique start node $\mathbf{s}$, and a unique end node $\mathbf{e}$, which are assumed to have no predecessors and successors, respectively. Each edge is associated with an assignment statement or with the statement "skip". Edges represent the branching structure and the statements of a program, while nodes represent program points. For readability the annotation "skip" is omitted in the figures.

By $pred(n) =_{df} \{ m \mid (m, n) \in E \}$ and $succ(n) =_{df} \{ m \mid (n, m) \in E \}$ we denote the set of immediate predecessors and successors of a node n. Additionally, by $source(e)$ and $dest(e)$, $e \in E$, we denote the *source node* and the *destination node* of edge e. A *finite path* in G is a sequence $(e_1, \ldots, e_q)$ of edges such that $dest(e_j) = source(e_{j+1})$ for $j \in \{1, \ldots, q-1\}$. It is called a path from m to n, if $source(e_1) = m$ and $dest(e_q) = n$. By $\mathbf{P}[m, n]$ we denote the set of all (finite) paths from m to n. Without loss of generality we assume that every node of a flow graph G lies on a path from $\mathbf{s}$ to $\mathbf{e}$.

Semantics of Terms. In this article we concentrate on integer expressions Exp which are inductively built from variables $v \in \mathbf{V}$, constants $c \in \mathbf{C}$, and binary integer operators $\mathbf{Op} = \{+, -, *, \mathtt{div}, \mathtt{mod}\}$. The *semantics* of integer expressions is induced by the *standard interpretation* $S = (\mathbb{Z}_\perp, S_0)$, where $\mathbb{Z}_\perp =_{df} \mathbb{Z} \cup \{\perp\}$ is the flat integer domain with least element $\perp$ and S_0 is a function mapping every integer constant $c \in \mathbf{C}$ to the corresponding datum $S_0(c) \in \mathbb{Z}$, and every integer operator $op \in \mathbf{Op}$ to the corresponding total and strict function $S_0(op) : (\mathbb{Z}_\perp)^2 \to \mathbb{Z}_\perp$. $\Sigma =_{df} \{ \sigma \mid \sigma : \mathbf{V} \to \mathbb{Z}_\perp \}$ denotes the set of *states*, and $\sigma_\perp$ the distinct *start state* assigning $\perp$ to all variables $v \in \mathbf{V}$. This choice reflects that we do not assume anything about the context of the analyzed program. The *semantics* of an expression $e \in \mathsf{Exp}$ is then given by the evaluation function $\mathcal{E} : \mathsf{Exp} \to (\Sigma \to \mathbb{Z}_\perp)$ inductively defined by $\mathcal{E}(x)(\sigma) =_{df} \sigma(x)$ for $x \in \mathbf{V}, \mathcal{E}(c)(\sigma) =_{df} I_0(c)$ for $c \in \mathbf{C}$ and $\mathcal{E}(op(e_1, e_2))(\sigma) =_{df} I_0(op)(\mathcal{E}(e_1)(\sigma), \mathcal{E}(e_2)(\sigma))$ for composite expressions.

Each assignment statement $\iota \equiv x := e$ is associated with the *state transformation function* $\theta_\iota : \Sigma \to \Sigma$ which is defined by $\theta_\iota(\sigma)(y) =_{df} \mathcal{E}(e)(\sigma)$ if $y = x$ and $\theta_\iota(\sigma)(y) =_{df} \sigma(y)$ otherwise.

The statement $\iota \equiv skip$ is associated with the identity state transformer, $\theta_\iota(\sigma) = \sigma$. We obtain the set of states Σ_n, which are possible at a program point $n \in N$ as follows:[4] $\Sigma_n =_{df} \{ \theta_p(\sigma_\perp) \mid p \in \mathbf{P}[\mathbf{s}, n] \}$.

[4] In the definition of Σ_n, θ_p denotes the straightforward extension of the state transformation functions to paths.

Classes of Constants

In the following we briefly introduce some classes of constants that are of particular interest with respect to the taxonomy considered later on in the paper. We start by providing a distinction of constants into the more common class of must-constants and the less frequently considered class of may-constants. For both we provide their formal definitions as well as some application scenarios.

Must-Constants. Formally, an expression e is a must-constant at node n if and only if

$$\exists\, d \in \mathbb{Z} \;\; \forall\, \sigma \in \Sigma_n.\;\; \mathcal{E}(e)(\sigma) = d.$$

The problem of (must-)constancy propagation is to determine for a given expression e, whether e is a must-constant and if so what the value of the constant is. This information can be used in various ways. The most important application is the *compile-time simplification* of expressions. Furthermore, information on must-constancy can be exploited in order to eliminate conditional branches. For instance, if there is a condition $e \neq d$ situated at an edge leaving node n and e is determined a must-constant of value d at node n, then this branch can be classified unexecutable (cf. Figure 3(a)). Since (must-)constant propagation and the elimination of unexecutable branches mutually benefit from each other, approaches for *conditional constant propagation* where developed taking this effect into account [20, 2].

May-Constants. Complementary to the must-constancy problem an expression e is a may-constant of value $d \in \mathbb{Z}$ at node n if and only if

$$\exists\, \sigma \in \Sigma_n.\;\; \mathcal{E}(e)(\sigma) = d.$$

Note that opposed to the must-constancy definition here the value of the constant is given as an additional input parameter. This naturally induces a *multiple value* extension of the notion of may-constancy. Given expressions $e_1, \ldots, e_k$ and values $d_1, \ldots, d_k \in \mathbb{Z}$ the corresponding multiple value may-constancy problem is defined by:

$$\exists\, \sigma \in \Sigma_n.\;\; \mathcal{E}(e_1)(\sigma) = d_1 \;\wedge\; \ldots \;\wedge \mathcal{E}(e_k)(\sigma) = d_k.$$

While may-constancy information cannot be used for expression simplification, it has also some valuable applications. Most obvious is a complementary branch elimination transformation. If an expression e is not a may-constant of value d at node n then any branch being guarded by a condition $e = d$ is unexcecutable (cf. Figure 3(b)).

May-constancy information is also valuable for reasoning about the aliasing of array elements. This can be used, for instance, for parallelization of code or for improving the precision of other analyses by excluding a worst-case treatment of assignments to elements in an array. Figure 4 gives such an example in the context of constant propagation. Here the assignment to x can be simplified towards

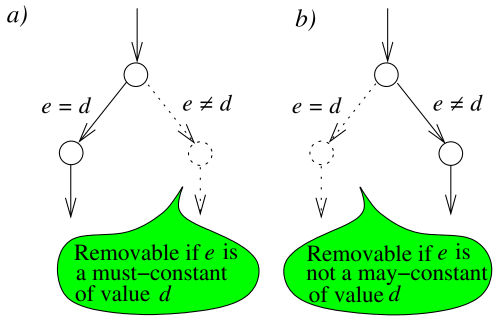

Fig. 3. Constancy information used for branch elimination.

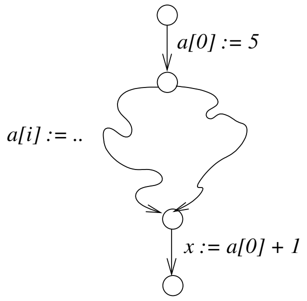

Fig. 4. Using array alias information from may-constant propagation in the context of (must)-constant propagation.

$x := 6$, only if the assignment to $a[i]$ does not influence $a[0]$. This, however, can be guaranteed if i is not a may-constant of value 0 at the corresponding program node.

Next we formally introduce some classes of constants according to the form of expressions that are allowed on the right hand side of assignments. Except of linear constants these classes are induced by considering only a fragment of the standard signature. While the first two classes are well-known in the field of constant propagation and the class of Presburger constants is closely related to the class of affine constants investigated in [6][5], we are not aware of any work devoted to the fragment of $+, -, *$-constants.

Copy-Constants. If all expressions in the program are non-composite, then the resulting constants are denoted *copy-constants*. This is due to the fact that constants can only be produced by assignments $x := c$ and be propagated by assignments of the form $x := y$.

Linear Constants. If the expressions in the program are restricted to linear ones, which means that all assignments take the form $x := a\,z + b$ where a and b are integer constants and z is an integer variable,[6] then the resulting constants are denoted *linear constants*.

Presburger Constants. If the expressions of the program are restricted to ones built from the operator $+$ and $-$, then the resulting constants are denoted *Presburger constants*. We decided for this term according to Presburger arithmetics, where integer operations are also restricted to addition and subtraction. However, the complexity issues in deciding Presburger formulas and Presburger constants are of a completely different nature, since in the context of constant propagation the problem is mainly induced by path conditions and not by a given logical formula. As far as expressiveness is concerned Presburger expressions and

[5] Affine constants are linear ones generalized towards multiple variables, i.e., constants in programs where expressions take only the form $a_1\,x_1 + \ldots + a_k\,x_k$.

[6] If $a = 0$ or $b = 0$ the corresponding summand may be omitted.

affine expressions coincide because multiplication with constants can be simulated by iterated addition. Affine expressions can, however, be more succinct. Nevertheless all results of this paper equally apply to both characterizations.

$+, -, *$-Constants. If the expressions of the program are restricted to ones built from the operator $+, -$ and $*$, then the resulting constants are called $+, -, *$-*Constants.*

It is for technical convenience and conceptual clarity that the classes of constants are introduced by means of a restriction to the form of all assignments in the considered programs. In practice, one uses a complete algorithm for either of the classes and extends it to programs with a more general kind of expressions in assignments by worst-case or approximating assumptions. The resulting algorithm can then still detect constants in the program completely that only depend on assignments of the given form. When judging the practical relevance of the results this should be kept in mind.

3 The Taxonomy and Complexity Results

3.1 Known Results

Table 1 summarizes the already known complexity results. Problems with a polynomial time algorithm are emphasized in a light shade of grey, those being decidable though intractable in a dark shade of grey, and the undecidable fields are filled black. White fields represent problems where the complexity and decidability is unknown or at least, to the best of our knowledge, undocumented.

In the following we briefly comment on these results. For an unrestricted signature we already presented Hecht's undecidability reduction for must-constants and the co-NP-hardness result for the acyclic counterpart.

It is also well-known that the must-constant propagation problem is distributive [4], if all right-hand side expressions are either constant or represent a one-to-one function in $\mathbb{Z} \to \mathbb{Z}$ depending on a single variable (see the remark on page 206 in [18]). Hence the class of linear constants defines a distributive data flow problem, which guarantees that the standard maximum fixed point iteration strategy over $\mathbb{Z} \cup \{\perp, \top\}$ computes the exact solution in polynomial time.[7]

On side of the may-constancy problem the class of copy-constant has recently been examined by Muth and Debray [12]. It is obvious, that the single value case can be dealt with efficiently. This is due to the fact that the number of constant values that a variable may posses at a program point (via copy-assignments) is bound to the number of assignments to constants in the program. Hence one can essentially keep track of any possible constant value at a program point by

[7] Sagiv, Reps and Horwitz [17] gave an alternative procedure for detecting linear constants by solving a graph reachability problem on the *exploded supergraph* of a program. They additionally showed that with this method linear constant propagation can be solved precisely even for interprocedural control flow.

		Must-Constants	May-Constants	
			single value	multiple value
acyclic control flow	Copy Constants	**P**	**P**	**NP-complete** Muth & Debray [12]
	Linear Constants	**P** Sharir & Pnueli [18]		
	Pressburger Constants			
	+,-,* Constants			
	Full Constants	**Co-NP hard** Knoop & Rüthing [7]		
unrestricted control flow	Copy Constants	**P**	**P**	**PSPACE-compl.** Muth & Debray [12]
	Linear Constants	**P** Sharir & Pnueli [18]		
	Presburger Constants			
	+,-,* Constants			
	Full Constants	**undecidable** Hecht [4]		

Table 1. Complexity classification of a taxonomy of CP: the known results

collecting the set of possible values of variables. Formally, this can be achieved by computing the union-over-all-path solution in a union-distributive data flow framework over the lattice $\{\sigma | \sigma : \mathbf{V} \to \mathfrak{P}(\mathbb{Z}_G)\}$, where $\mathbb{Z}_G$ denotes the set of constant right-hand sides in the flow graph G under consideration.

The multiple value problem has been shown NP-complete in the acyclic case and PSPACE-complete in the presence of unrestricted control flow [12].

In the remainder of this section we aim at successively filling the white parts in Table 1. To this end, we start with providing new undecidability results, then give some new intractability results and finally indicate that constant propagation can be achieved efficiently for the class of Presburger constants.

3.2 New Undecidability Results

Fortunately, Hecht's construction that was sketched in the introduction can easily be adapted for proving undecidability of Presburger may-constants. The only modification necessary for this is to replace the two assignment to r in Figure 1 by a single assignment $x := x - y$. As argued before, x may equal y immediately after leaving the loop, if and only if the instance of the Post correspondence problem has a solution. Hence in this case $x - y$ may evaluate to 0. As the multiplications with the constants $10^{lg(u_i)}$ and $10^{lg(v_i)}$ can be expressed by iterated additions, we get:

Theorem 1. *Deciding single valued may-constancy at a program point is undecidable for the class of Presburger constants.*

This construction can be further modified to obtain an even stronger undecidability result for the class of multiple value may-constants. Here we have:

Theorem 2. *Deciding multiple valued may-constancy at a program point is undecidable for the class of linear constants. This even holds if only two values are questioned.*

The idea is to substitute the difference $x - y$ in the assignment to r by a loop which simultaneously decrements x and y. It is easy to see that $x = 0 \wedge y = 0$ may hold at the end of such a program fragment, if and only if x may equal y at the end of the main loop.

Complexity of Array Aliasing. The previous two undecidability results have an immediate impact on the problem of array aliasing, which complements similar results known in the field of pointer induced aliasing [8]. In fact as a consequence of Theorem 1 we have:

Corollary 1. *Deciding whether $a[i]$ may alias $a[c]$ for a one-dimensional array a, integer variable i and integer constant c is undecidable, even if i is computed only using the operators $+$ and $-$.*

In fact, Theorem 2 even provides some negative results for array accesses when using only linear index calculations.[8] We have:

Corollary 2. *Let c_1, c_2 be integer constants and i, j integer variables being computed only with linear assignments of the form $x := a\,y + b$. Then the following problems are undecidable:*

1. *Determining whether $a[i]$ may alias $a[j]$ for a one-dimensional array a.*
2. *Determining whether $a[i, j]$ may alias $a[c_1, c_2]$ for a two-dimensional array a.*

It should be noted that traditional work on array dependences like the omega test [14, 15] is restricted to scenarios where array elements are addressed by affine functions depending on some index variables of possibly nested for-loops. In this setting the aliasing problem can be stated as an integer linear programming problem which can be solved effectively. In contrast, our results address the more fundamental issue of aliasing in the presence of arbitrary loops.

3.3 New Intractability Results

After having marked off the range of undecidability we prove in this section intractability of some of the uncovered fields.

We start by strengthening the result on the co-NP-hardness of must-constant propagation for acyclic control flow. Here the construction of Figure 2 can be

[8] The first part is not an immediate corollary, but relies on the same construction as Theorem 2.

modified such that the usage of integer division is no longer necessary. Basically, the trick is to use multiplication by 0 as the projective operation, i.e., as the operation with the power to map many different values onto a single one. In the construction of Figure 2 this requires the following modifications. All variables are now initialized by 1. The part reflecting the clauses then sets the corresponding variables to 0. Finally the assignments to r_1 and r_2 are substituted by a single assignment $r := (x_1 + \overline{x}_1) \cdot \ldots \cdot (x_k + \overline{x}_k)$ being bypassed by another assignment $r := 0$. It is easy to see that the instance of 3-SAT has no solution if and only if on every path both x_i and $\overline{x_i}$ are set to 0 for some $i \in \{1, \ldots, k\}$. This, however, guarantees that at least one factor of the right-hand side expression defining r is 0 which then ensures that r is a must-constant of value 0. Finally, the branch performing the assignment $r := 0$ assures that r cannot be a must-constant of any other value. Thus we have:

Theorem 3. *Must-constant propagation is co-NP hard even when restricted to acyclic control flow and to* $+, -, *$*-constants.*

On the other hand, we can show that the problem of must-constant propagation is in co-NP for acyclic control flow. To this end, one has to prove that the co-problem, i.e., checking non-constancy at a program point, is in NP, which is easy to see: a non-deterministic Turing machine can guess two paths through the program witnessing two different values. Since each path is of linear length in the program size and the integer operations can be performed in linear time with respect to the sum of the lengths of the decimal representation of their inputs, this can be done in polynomial time. Hence we have:

Theorem 4. *Must-constant propagation is in co-NP when restricted to acyclic control flow.*

Next we are going to show that the problem addressed by Theorem 3 gets presumably harder without the restriction to acyclic control flow.

Theorem 5. *Must-constant propagation is PSPACE-hard even when restricted to* $+, -, *$*-constants.*

Theorem 5 is proved by means of a polynomial time reduction from the language universality problem of nondeterministic finite automata (NDFA) (cf. remark to Problem AL1 in [3]). This is the question whether an NDFA $\mathcal{A}$ over an alphabet X accepts the universal language, i.e., $L(\mathcal{A}) = X^*$. W.l.o.g. let us thus consider an NDFA $\mathcal{A} = (X, S, \delta, s_1, F)$, where $X = \{0, 1\}$ is the underlying alphabet, $S = \{1, \ldots, k\}$ the set of states, $\delta \subseteq S \times X \times S$ the transition relation, s_1 the start state, and $F \subseteq S$ the set of accepting states. The polynomial time reduction to a constant propagation problem is depicted in Figure 5.

For every state $i \in \{1, \ldots, k\}$ a variable s_i is introduced. The idea of the construction is to guess an arbitrary input word letter by letter. While this is done, it is ensured by appropriate assignments that each variable s_i holds 0 if and only state i is reachable in the automaton under the word. $\prod_{i \in F} s_i$ is then 0 for all words if and only if $\mathcal{A}$ accepts the universal language.

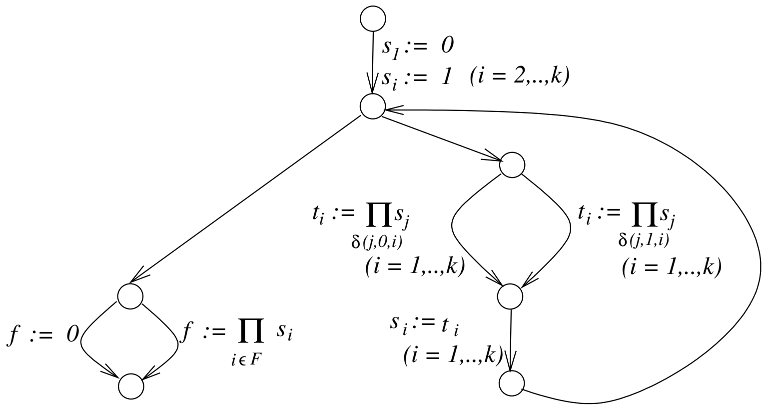

Fig. 5. PSPACE-hardness of must-constant propagation for $+, -, *$-constants.

Initially, only the start state variable s_1 is set to 0 as it is the only state which is reachable under the empty word. The central part of the program is a loop which guesses an alphabet symbol for the next transition. If we decide, for instance, for 0 then, for each i, an auxiliary state variable t_i is set to 0 by the assignment $t_i := \prod_{\delta(j,0,i)} s_j$, if and only if one of its 0-predecessors is recognized reachable.[9] After all variables t_i have been set in this way their values are copied to the variables s_i, respectively. The loop can be left at any time; then it is checked whether the guessed word is accepted. Like before, the direct assignment $f := 0$ has the purpose to ensure that constant values different from 0 are impossible. Therefore, f is a must-constant (of value 0) at the end of the program, if and only if the underlying automaton accepts the universal language $\{0, 1\}^*$.

The final reduction in this section addresses the complexity of linear may-constants. Here we have:

Theorem 6. *May-constant propagation is NP-hard even when restricted to the class of linear constants.*

Again we employ a polynomial time reduction from 3-SAT which however differs from the ones seen before. The major idea here is to code a set of satisfied clauses by a number interpreted as a bit-string. For example, in an instance with four clauses the number 1100 would indicate that clause two and three are satisfied, while clause zero and one are not. To avoid problems with carry-over effects, we employ a $(k+1)$-adic number representation where k is the number of variables in the 3-SAT instance. With this coding we can use linear assignments to set the single "bits" corresponding to satisfied clauses.

[9] Auxiliary state variables are introduced in order to avoid overwriting state variables which are still used in consecutive assignments.

To illustrate our reduction let us assume an instance of 3-SAT with Boolean variables $\{b_1, \dots, b_k\}$ and clauses $c_0, \dots, c_{n-1}$, where the literal b_1 is contained in c_3 and c_5, and the negated literal $\neg b_1$ is contained in c_2 only. Then this is coded in a program as depicted in Figure 6. We have a non-deterministic choice part for each Boolean variable b_i. The left branch sets the bits for the clauses that contain b_i and the right branch those for the clauses that contain $\overline{b_i}$. Every assignment can be bypassed by an empty edge in case that the clause is also made true by another literal. It is now easy to see that r is a may-constant of value $\underbrace{1 \dots 1}_{n\ times}$ (in $(k+1)$-adic number representation) if and only if the underlying instance of 3-SAT is satisfiable.

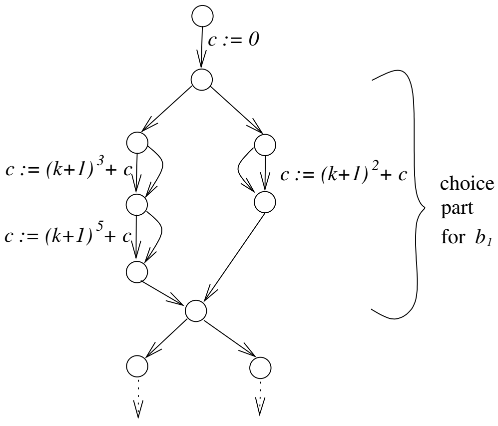

Fig. 6: NP-hardness of linear may-CP.

On the other hand, it is easy to see that detecting may-constancy is in NP for acyclic control flow, since a nondeterministic Turing machine can guess a witnessing path for a given constant in polynomial time. We have:

Theorem 7. *May-constant propagation is in NP when restricted to acyclic control flow.*

3.4 New Polynomial-Time Algorithm

In this section we fill the last field in our taxonomy by showing that all Presburger constants can be detected in polynomial time.

One way of showing this claim is by carefully investigating a polynomial-time algorithm proposed by Karr [6]. He employs a *forward* data flow analysis that establishes for each program point n an *affine vector space* (over $\mathbb{Q}$) that over-approximates Σ_n. This information can in turn be used to detect certain constants. It can be shown that the resulting algorithm is complete with respect to the detection of Presburger constants, a question that has not been explored by Karr, but this proof is beyond the scope of the current paper. In the following we sketch a new algorithm that leads to a more transparent proof of polynomial-time detectability of Presburger constants.

Figure 7 gives an impression on the problem dimension behind this class where the emphasized annotation will be explained later.

Part (a) of this Figure extends the classical non-distributivity pattern of constant propagation (cf. [4]). The point here is that z is a must-constant of value 14 at the end of the program. However, none of its operands is constant, although both are defined outside of any conditional branch. Part (b) shows a small loop example where z is a must-constant of value 0. However, also this

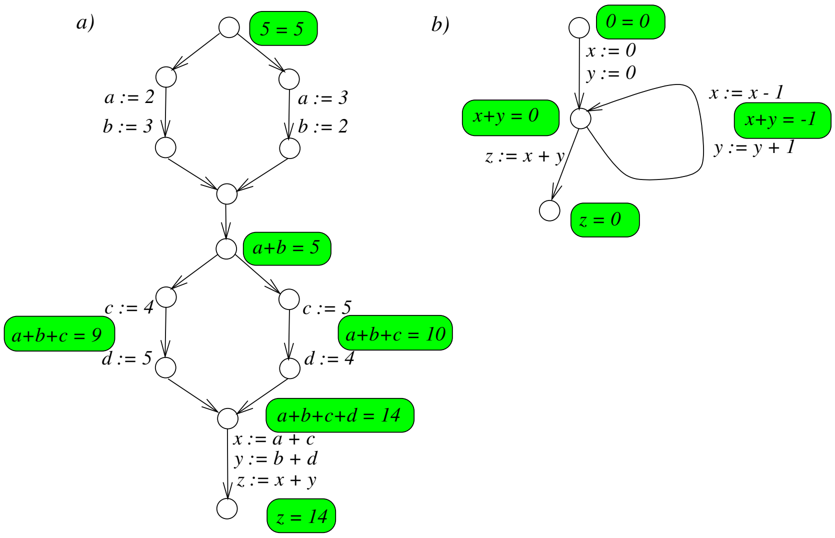

Fig. 7. Deciding Presburger constants by backward propagation of linear constraints.

example is outside of the scope of any standard algorithm except of Karr's, and even outside of the scope of Knoop's and Steffen's EXPTIME algorithm for detecting finite constants [19].

The algorithm at a glance. Our algorithm employs techniques known from linear algebra. In fact, we use a backward analysis propagating sets of *linear equational constraints* describing *affine vector spaces* (over $\mathbb{Q}$).

The Data Flow Framework. Given a set of program variables $\{x_1, \ldots, x_k\}$ a linear constraint is an equation of the form: $\sum_i a_i\, x_i = b$ where $a_i, b \in \mathbb{Q}$ $(i = 1, \ldots, k)$. Since at most k of these linear constraints are linearly independent, an affine vector space can always be described by means of a linear equation system $\mathbf{A}\, x = b$ where $\mathbf{A}$ is a $k \times k$-matrix. The affine vector sub-spaces of $\mathbb{Q}^k$ can be partially ordered by set inclusion. This results in a (complete) lattice where the length of chains is bounded by k as any affine space strictly contained in another affine space has a smaller dimension.

The Meet Operation. The meet of two affine vector spaces represented by the equations $\mathbf{A}_1\, x = b_1$ and $\mathbf{A}_2\, x = b_2$ can be computed by normalizing the equation

$$\begin{pmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

which can be done efficiently using Gauss-elimination [13].

Local Transfer Functions. The local transfer functions are realized by performing a backward substitution on the linear constraints. For instance, a constraint $3x + y = 10$ is backward-substituted along an assignment $x := 2u - 3v + 5$ towards $3\,(2\,u\, -\, 3\,v\, +\, 5) + y = 10$ which then can be "normalized" towards $y + 6\,u\, -$

$9\,v = 5$. Clearly, this can be done in polynomial time. After this normalization, the resulting equation system is also simplified using Gauss-elimination.

The Overall Procedure. Our backward data flow analysis can be regarded as a demand-driven analysis which works separately for each variable x and program point n. Conceptually, it is organized in three stages:

Stage 1: Guess an arbitrary cycle-free path leading to n, for instance using depth-first search, and compute the value d of x on this path.
Stage 2: Solve the backward data flow analysis where initially the program point n is annotated by the affine vector space described by the linear constraint: $x = d$ and all other program points by the universal affine space, i.e., the one given by $\mathbf{0}\,x = 0$.
Stage 3: The guess generated in stage 1 is proved, if and only if the start node is still associated with the universal affine vector space.[10]

The completeness of the algorithm is a simple consequence of the distributivity of the analysis. Obviously, the guessed constraint is true iff the backward substitution along every path originating at the start node yields a universally valid constraint at the start node. Since this defines the meet-over-all-paths solution of our data flow framework the algorithmic solution is guaranteed to coincide if the transfer functions are distributive, which is immediate from the definition.

The algorithm can also be understood from a program verification point of view. By Stage 1, d is the only candidate value for x being constant at n. Stage 2 effectively computes the weakest (liberal) precondition of the assertion $x = d$ at program point n. Clearly, x is a constant at n if and only if the weakest liberal precondition of $x = d$ is universally valid.

As mentioned, the length of chains in the analysis is bound by the number of variables k. Any change at a node can trigger a reevaluation at its predecessor nodes. Therefore, we have at most $O(e \cdot k)$ Gauss-elimination steps, where e denotes the number of edges in the flow graph. Each Gauss-elimination step is of order $\mathcal{O}(k^3)$ [13]. Thus the complexity for the complete data flow analysis w.r.t. a single occurrence of a program variable is $\mathcal{O}(e\,k^4)$. For an exhaustive analysis that computes must-constancy information for any left-hand side occurrence of a variable the estimation becomes $\mathcal{O}(n\,e\,k^4)$, where n denotes the number of nodes in the flow graph. Summarizing, we have:

Theorem 8. *The class of Presburger must-constants can be detected in polynomial time.*

Finally, we are going to illustrate our algorithm by means of the example of Figure 7. The emphazised annotation of Figure 7 contains the constraints resulting from the initial guess $z = 14$ (in Figure 7(a)) and $z = 0$ (in Figure 7(b)), respectively. It should be noted that for the sake of presentation we did not display the constraints for every program point. The particular power of this

[10] In practice, one may already terminate with the result of non-constancy of x whenever a linear equation system encountered during the analysis renders unsolvable.

technique lies in the normalization performed on the linear constraints which provides a handle to cope with arithmetic properties like commutativity and associativity to a certain extent. For instance, the constraint $a + b = 5$ in Figure 7(a) has been the uniform result of two different intermediate constraints.

4 Summary and Conclusions

The decidability and complexity considerations of this paper are summarized in Table 2. In fact, we almost completely succeeded in filling the white fields of Table 1. As apparent, only two upper bounds are left open. At the moment we neither have an upper bound for the class of $+, -, *$-must-constants nor for the class of linear may-constants. Although we do not expect one of the problems to be undecidable, a solution might require some deeper number theoretical insights.

An interesting observation which is immediately obvious from inspecting the table is that the detection of may-constants is significantly harder than detecting their must-counterparts.

Future work will be concerned with answering the open upper bounds, with tuning the constraint based technique for Presburger constants into an algorithm that is usable in practice, and with extending the taxonomy by considering advanced settings, like interprocedural or parallel ones, too. In as yet unpublished work we show that in a setting with fork-join type parallelism the intraprocedural problem is PSPACE-complete already for may- and must-copy constants and becomes even undecidable for the corresponding interprocedural problems (see [11] for somewhat weaker results).

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181 – 196, 1995.
3. M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co, San Francisco, 1979.
4. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
5. N. D. Jones and S. S Muchnick. Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra. In [10], chapter 12, pages 380–393.
6. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
7. J. Knoop and O. Rüthing. Constant propagation on the value graph: simple constants and beyond. In *CC'2000*, LNCS 1781, pages 94–109, Berlin, Germany, 2000. Springer-Verlag.
8. W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.
9. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

		Must-Constants	May-Constants	
			single value	multiple value
acyclic control flow	Copy Constants			
	Linear Constants			
	Pressburger Constants		NP complete	
	+,-,* Constants	Co-NP compl.		
	Full Constants			
unrestricted control flow	Copy Constants			PSPACE-compl.
	Linear Constants		NP-hard	
	Presburger Constants		undecidable	
	+,-,* Constants	PSPACE-hard		
	Full Constants			

Table 2. Complexity classification of a taxonomy of CP: summarizing the results

10. S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1981.
11. M. Müller-Olm. The complexity of copy constant detection in parallel programs. In *STACS'2001*, LNCS, 2001. Springer-Verlag. To appear.
12. R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analysis. In *POPL'2000*, pages 67–81, ACM, Boston, MA, 2000.
13. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
14. W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In IEEE, editor, *Supercomputing '91*, Albuquerque, NM, pages 4–13. IEEE Computer Society Press, 1991.
15. W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, 1998.
16. J. H. Reif and R. Lewis. Symbolic evaluation and the gobal value graph. *POPL'77*, pages 104–118, ACM, Los Angeles, CA, 1977.
17. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.
18. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In [10], chapter 7, pages 189–233.
19. B. Steffen and J. Knoop. Finite constants: Characterizations of a new decidable set of constants. *Theoretical Computer Science*, 80(2):303–318, 1991.
20. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2), 1991.