On the Complexity of Constant Propagation

Markus Müller-Olm and Oliver Rüthing

University of Dortmund
{mmo,ruething}@ls5.cs.uni-dortmund.de

Abstract. *Constant propagation* (*CP*) is one of the most widely used optimisations in practice (cf. [6]). Intuitively, it addresses the problem of statically detecting whether a variable evaluates to a constant at runtime. Unfortunately, as proved by different authors [3, 10], 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. [5]). 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. In particular, we strictly improve on the best-known class of constants with polynomial time algorithms.

1 Motivation

Constant propagation (*CP*) is one of the most widely used optimisations in practice (cf. [1, 3, 6]). 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 [3] and by Reif and Lewis [10]. 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 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 r from

[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.

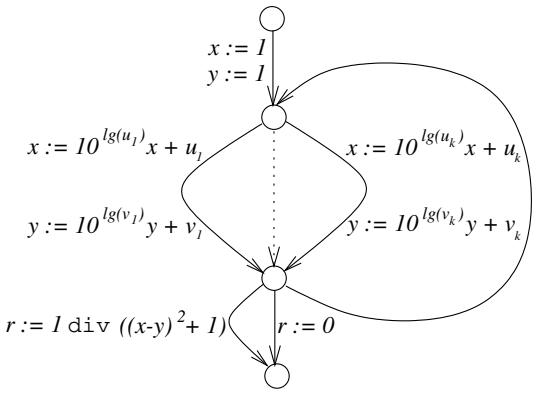

Fig. 1. Undecidability of constant propagation: reduction of the Post correspondence problem.

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 recently shown to be Co-NP-hard [5]. 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. [2]). 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 $r2$ 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.

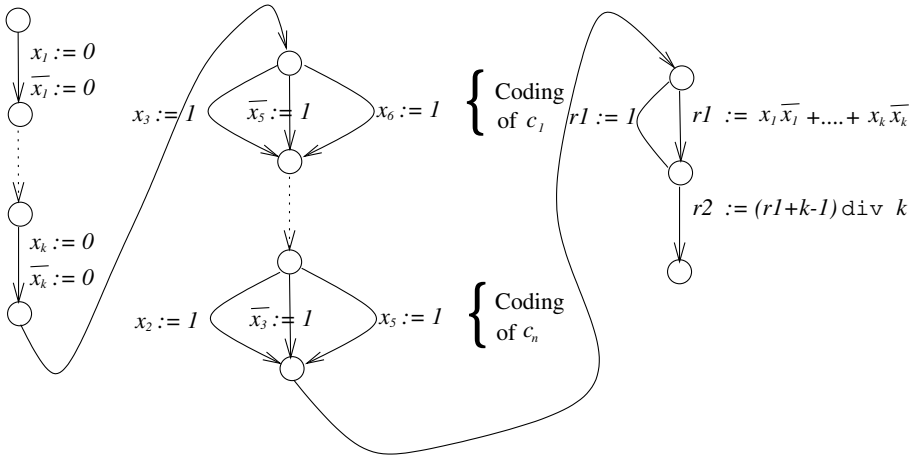

Fig. 2. Co-NP-hardness of constant propagation for acyclic programs: reduction of co-3-SAT.

Contributions This paper aims at examining the borderline of intractability and undecidability more closely. To this end, we investigate the constant propagation problem for integers with respect to a three-dimensional taxonomy. The first dimension is given by the distinction between arbitrary and loop-free programs. The second dimension in 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 $+, -, *, \texttt{div}, \texttt{mod}$. 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 [8] 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. 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. May-constancy information is also useful for reasoning on aliasing of arrays.

2 The Taxonomy and Complexity Results

2.1 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

		Must-Constants	May-Constants	
			single value	multiple value
acyclic control flow	Copy Constants	**P**	**P**	**NP-complete** Muth & Debray [8]
	Linear Constants	**P** Sharir & Pnueli [12]		
	Pressburger Constants			
	+,-,* Constants			
	Full Constants	**Co-NP hard** Knoop & Rüthing [5]		
unrestricted control flow	Copy Constants	**P**	**P**	**PSPACE-compl.** Muth & Debray [8]
	Linear Constants	**P** Sharir & Pnueli [12]		
	Presburger Constants			
	+,-,* Constants			
	Full Constants	**undecidable** Hecht [3]		

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

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 [3], 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 [12]. Hence the class of linear constants defines a distributive data flow problem, which guarantees that the standard maximum fixed point iteration strategy over $\mathbb{Z} \cup \{\bot, \top\}$ computes the exact solution in polynomial time.[3]

On side of the may-constancy problem the class of copy-constant has recently been examined by Muth and Debray [8]. 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

[3] Sagiv, Reps and Horwitz [11] 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.

can essentially keep track of any possible constant value at a program point by 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 [8].

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.

2.2 New Undecidability Results

Fortunately, Hecht's construction can easily be adapted for proving undecidability of Presburger may-constants. The only modification necessary for this is to replace the assignment to r in Figure 1 by an 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 multiplications with constants 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 loop.

2.3 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 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 to 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) [9]. 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 3.

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 a variable holds 0 if and only if the corresponding state 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.

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

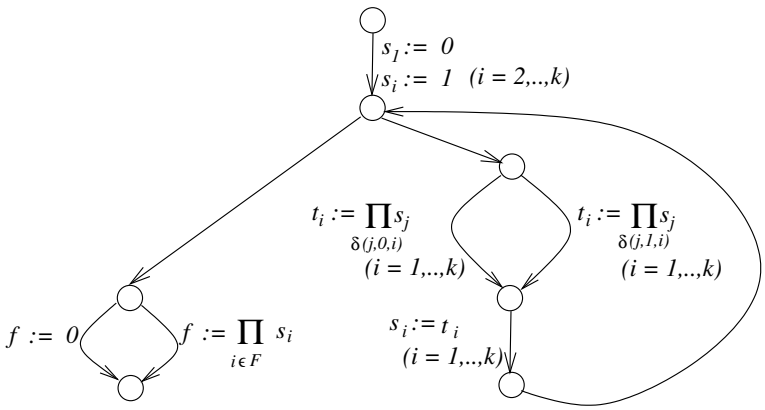

Fig. 3. PSPACE-hardness of must-constant propagation for $+, -, *$-constants.

instance, for 0 then, for each i, an auxiliary state variable t_i is set to 0 by the assignment $\overline{s}_i := \prod_{\delta(j,0,i)} s_j$, if and only if one of its 0-predecessors is recognized reachable.[4] 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 and nine variables, the decimal number 1100 would indicate that clause three and four are satisfied, while clause one and two 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" coresponding to satisfied clauses.

To illustrate our reduction let us assume an instance of 3-SAT with Boolean variables $\{b_1, \ldots, b_k\}$ and clauses $c_1, \ldots, c_n$, 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 4. 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

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

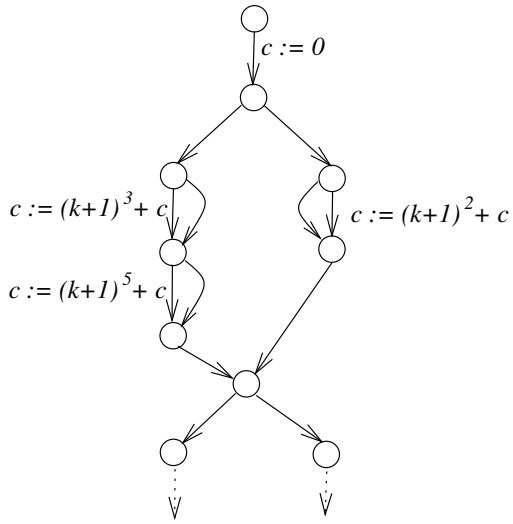

Fig. 4. NP-hardness of may-constant propagation for linear constants.

made true by another literal. It is now easy to see that r is a may-constant of value

$$\underbrace{1 \ldots 1}_{n},$$

(in $(k+1)$-adic number representation) if and only if the underlying instance of 3-SAT is satisfiable.

On the other hand, it is easy to see that detecting may-constancy is in NP for acyclic control flow, since a nondeterminininistic 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.*

2.4 New Polynomial-Time Algorithms

In this section we fill the last fields in our taxonomy by providing a new algorithm that is able to detect all Presburger constants in polynomial time.

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).[5] 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 [9].

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 path leading to n and compute the value d of x on this path.

Stage 2: Solve the backward data flow analysis where 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.[6]

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 contraint 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

[5] A forward data flow analysis based on affine vector spaces has first been proposed by Karr [4]. However, he dows not show that his method is complete with respect to the detection of Presburger constants. We are currently examining this question.

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

effectively computes the weakest (liberal) precondition of the assertion $x = d$ at program point n. Thus 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. As the local steps that use Gauss-elimination are of order $\mathcal{O}(k^3)$ [9], the overall effort at a node can be estimated by $\mathcal{O}(k^4)$. Thus the complexity for the complete data flow analysis w.r.t. a single occurrence of a program variable is $\mathcal{O}(n\,k^4)$, where n denotes the number of nodes in the flow graph. For an exhaustive analysis that computes must-constancy information for any left-hand side occurrence of a variable the estimation becomes $\mathcal{O}(n^2\,k^4)$. Summarizing, we have:

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

3 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 in the supplemented table. At the moment we do not have upper bounds for the class of $+,-,*$-must-constants and neither 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 [7].

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. 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.
3. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
4. Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
5. J. Knoop and O. Rüthing. Constant propagation on the value graph: Simple constants and beyond. In *Proc. 9^{th} Conference on Compiler Construction (CC)*, Lecture Notes in Computer Science 1781, pages 94 – 109, Berlin, Germany, 2000. Springer-Verlag.

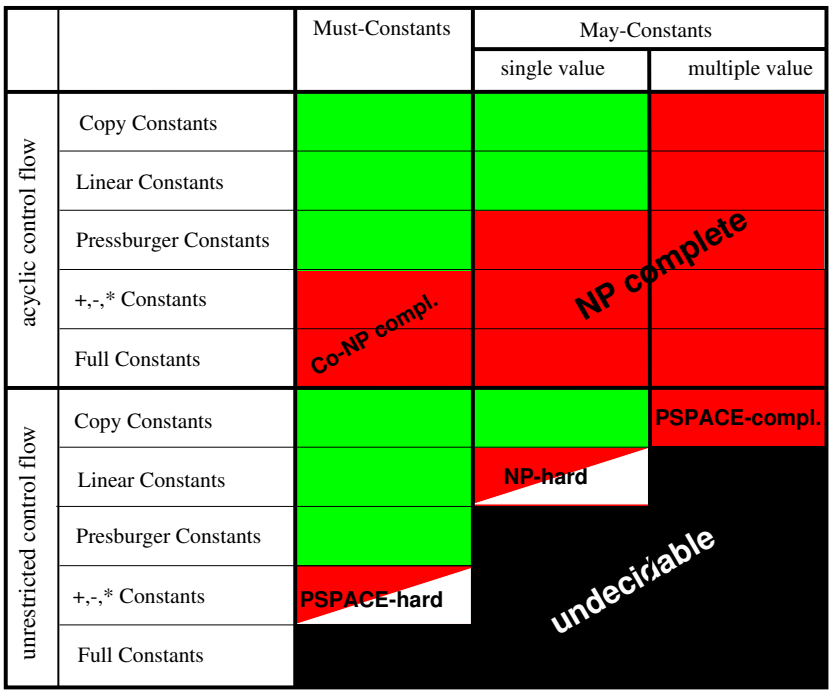

Table 2. Complexity classification of a taxonomy of constant propagation: summarizing the results

6. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.
7. M. Müller-Olm. On the complexity of copy constant detection in parallel programs. In W. Goerigk, editor, *Proceedings of the 17th Annual Workshop of the GI–FG 2.1.4 "Programmiersprachen und Rechenkonzepte" mit Schwerpunkt Softwarekomponenten, Physikzentrum Bad Honnef (Germany), May 8 –10*, Technical Report 2007, Institut für Informatik und Praktische Mathematik, 2000. This volume.
8. R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analysis. In *Conf. Record of the 27^{th} ACM Symposium on the Principles of Programming Languages* (*POPL*), pages 67 – 81, Boston, MA, January 2000.
9. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
10. J. H. Reif and R. Lewis. Symbolic evaluation and the gobal value graph. In *Conf. Record of the 4^{th} ACM Symposium on the Principles of Programming Languages* (*POPL*), pages 104 – 118, Los Angeles, CA, 1977.
11. Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 30 October 1996.
12. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189 – 233. Prentice Hall, Englewood Cliffs, NJ, 1981.