International Journal of Foundations of Computer Science
Vol. 23, No. 7 (2012) 1451–1464

DOI: 10.1142/S0129054112400588

ON THE EDGE OF DECIDABILITY IN COMPLEXITY ANALYSIS OF LOOP PROGRAMS*

AMIR M. BEN-AMRAM†

School of Computer Science, The Academic College of Tel-Aviv Yaffo
2 Rabenu Yeruham Str., Tel-Aviv, 68182, Israel
benamram.amir@gmail.com

LARS KRISTIANSEN†

Department of Mathematics, University of Oslo, P. O. Box 1053, Blindern
Oslo, NO-0316, Norway
larsk@math.uio.no

Received 31 December 2010
Accepted 18 December 2011
Communicated by Klaus Sutner

We investigate the decidability of the *feasibility problem* for imperative programs with bounded loops. A program is called *feasible* if all values it computes are polynomially bounded in terms of the input. The feasibility problem is representative of a group of related properties, like that of polynomial time complexity. It is well known that such properties are undecidable for a Turing-complete programming language. They may be decidable, however, for languages that are not Turing-complete. But if these languages are expressive enough, they do pose a challenge for analysis. We are interested in tracing the edge of decidability for the feasibility problem and similar problems.

In previous work, we proved that such problems are decidable for a language where loops are bounded but *indefinite* (that is, the loops may exit before completing the given iteration count). In this paper, we consider definite loops. A second language feature that we vary, is the kind of assignment statements. With ordinary assignment, we prove undecidability of a very tiny language fragment. We also prove undecidability with *lossy assignment* (that is, assignments where the modified variable may receive any value bounded by the given expression, even zero). But we prove decidability with *max assignments* (that is, assignments where the modified variable never decreases its value).

Keywords: Static analysis; loop programs; subrecursive programming languages.

*Dedicated to Neil D. Jones on the occasion of his 70th birthday.
†Part of this work was done when both authors were visiting DIKU, the Department of Computer Science at the University of Copenhagen, Denmark.

1. Introduction

1.1. *Background*

Devising algorithms that deduce complexity properties of a given program is a classic problem of program analysis [16, 19, 23], and has received considerable attention in recent years, e.g., see the LNCS volume [22]. Ideally, a static-analysis tool will warn us in compilation-time whenever our program fails a complexity specification. For instance, we could be warned of algorithms whose running time is not polynomially bounded, or algorithms that compute super-polynomially large values.

Since deciding such a properties precisely for any program in a Turing-complete language is impossible, we are led to investigate the decidability of the problem in restricted languages. In previous work [3], we showed the problem of *polynomial boundedness* to be decidable for a "core" imperative language L_{BJK} with a bounded loop command. The language only handled numeric variables, with the basic operations of addition and multiplication.

Figure 1 shows the syntax of the language; its semantics is almost self-explanatory, and will be further explained below. Figure 2 illustrates the analysis problem we shall focus on: deciding whether all values computed by a program are polynomially bounded.

An important aspect of the language L_{BJK} was the inclusion of non-deterministic commands: in particular, the `if` command has no conditional and is interpreted as a non-deterministic choice; the loop may (non-deterministically) exit before the loop count is exhausted; we also considered a non-deterministic kind of assignment.

The initial reason for introducing non-determinism was the expectation that if deterministic conditional branching was included, the language would defy a decision procedure; so we followed the conservative approach often used in program analysis, treating both branches as possible. This decision meant that our language could be used to model concrete programs by over-approximating their semantics. Having made this choice, we realized that additional non-deterministic constructs may be useful as over-approximations of concrete ones. Our bounded loops (like Pascal `for` loops), that allow a non-deterministic premature exit, capture programs that actually employ `while` loops (as long as an iteration bound can be given), and programs that use an early exit mechanism (like the C command `break`).

$$
\begin{array}{rcl}
\texttt{X} \in \text{Variable} & ::= & \texttt{X}_1 \mid \texttt{X}_2 \mid \texttt{X}_3 \mid \ldots \mid \texttt{X}_n \\
\texttt{e} \in \text{Expression} & ::= & \texttt{X} \mid \texttt{(e + e)} \mid \texttt{(e * e)} \\
\texttt{C} \in \text{Command} & ::= & \texttt{skip} \mid \texttt{X:=e} \mid \texttt{C}_1\texttt{;C}_2 \mid \texttt{?loop X \{C\}} \\
& \mid & \texttt{if ? then C else C}
\end{array}
$$

Fig. 1. Syntax of the language L_{BJK}.

Program 1:

```
loop X5 {
  if ? then { X3 := X1; X4 := X2 }
       else { X3 := X2; X4 := X1 };
  X1 := X3 + X4;
}
```

Program 2:

```
loop X4 {
  X2 := X1;
  X3 := X2;
  X1 := X3 + X2;
}
```

Fig. 2. In the first program, all variables are polynomially bounded, while in the second, exponential growth occurs.

As in this paper we contrast the two kinds of bounded loops, we employ the notation `!loop` for the *definite* type (which cannot exit early), and `?loop` for the *indefinite* type (which can).

The non-deterministic assignment ("lossy assignment") is another feature that could be used to represent (again, by over-approximation) concrete programs in richer languages. The lossy assignment statement $\mathtt{X} :\leq \mathtt{Y}$ sets variable `X` to some non-negative integer[a] bounded by `Y`. Thus, it is not really an assignment, but a constraint. Constraints are widely used in program analysis to represent sound over-approximations of program semantics.

1.2. *Problems left open and new results*

One of the insights we gained regarding our work in [3], was that the key to success is the property of *monotonicity*.

Firstly, all the functions computed by the arithmetic operators we admitted (addition and multiplication) are monotone. Indeed, if we extend the language with subtraction of integers, decidability is lost, which should come as no surprise.

Secondly, monotonicity is also necessary in the semantics of commands. While for the `if` command and sequential composition, a suitable notion of monotonicity can be easily established, the loop command is more difficult. We found that *definite* loop semantics destroys monotonicity: a second iteration may erase the result of the first, so that increasing the iteration count does not increase the output. Our

[a]For simplicity, we restrict all data in this paper to non-negative integers. The positive results that we cite from [2, 3] are actually applicable to richer data types.

non-deterministic loop semantics restores monotonicity since increasing the loop variable only makes more outcomes possible.

Another limitation of our analysis [3] was that we could not handle constants, for instance the command `X:=0`.

The reader may feel that the problem is less severe if 0 is the *only* constant allowed, and this is indeed the case. In [2], L_{BJK} is extended with the constant 0 and decidability is shown. Extending the language with the constants 0 and 1 yields a variant whose analysis remains an open problem.

In this paper, we focus on the difficulty caused by the definite loop. For a language with precise addition and assignments, and the constants 0 and 1, including `!loop` leads to undecidability: The language can actually compute all primitive recursive functions, and the impossibility of deciding various properties in such a language is well known. We are thus interested in weakening the language, to find the edge of decidability. First, we consider lossy assignments. This kind of assignment statement sets the variable on the left-hand side non-deterministically to a value between 0 and the value on the right-hand side. This seems to take the edge off `!loop`. Surprisingly, we found that undecidability still holds. Furthermore we can (with both standard and lossy assignment) eliminate the constants. This culminates in a proof that a language with `!loop` and a single form of assignment statement, either $\texttt{X} := \texttt{Y+Z}$ or $\texttt{X} :\leq \texttt{Y+Z}$, suffices to make complexity analysis undecidable.

These proofs are given in Section 3. In Section 4, we go back to the decidable side of the line, by considering the *max assignment*. The semantics of $\texttt{X} :\overset{\max}{=} \mathit{exp}$ is to set `X` to the value of *exp* if this increases `X`; otherwise, `X` keeps its current value. We prove that under this assignment semantics, the language with `!loop` can be analyzed precisely (we consider the variant that has expressions `Y+Z` as well as the constant 0). This agrees very well with the intuition presented earlier, since the semantics of max assignments makes the sequence of computed values monotone.

Like the "trick" of super-approximating program semantics by introducing nondeterminism, the max assignment can also be found in the practice of program analysis, where the analysis often relies on *program invariants*. To derive invariants from the program semantics one goes through what is often called *a collecting semantics* [6] where the "value" of a variable becomes the set of all values it assumes throughout the computation. Thus, as assignment becomes *an addition to the set*. The set itself is further abstracted to a simpler structure, a classic example being intervals [4, 5]. The assignment $\texttt{X} :\overset{\max}{=} \mathit{exp}$ can be viewed as an instruction updating the limit of the interval.

1.3. *Related work*

The idea of studying programs based on a restricted type of loops, in order to make the complexity analysis problem more accessible — and even precisely decidable — goes back to the seminal paper of Meyer and Ritchie [17]. Their class of *loop programs* only has bounded loops and a static upper bound on their complexity can

be computed. While these upper bounds are tight for the class of programs as a whole, many programs of the class have a lower complexity, so we can try to analyze a given program more precisely. Works of this kind include [1, 10, 11, 14, 18]. With a single exception, these works proposed syntactic criteria, or analysis algorithms, that are sufficient for ensuring that the program lies in a desired class (say, polynomial-time programs), but are not both necessary and sufficient: thus, they do not address the decidability question (the exception is [14] which has a decidability result for a "core" language).

A key notion in the study of loop programs, starting with [11, 17], is the nesting depth of loops . Programs of nesting depth 2, called LOOP(2) programs, can compute all the Kalmar-elementary functions, which makes them powerful enough to defy decidability for many properties of interest. A prototypical example is the equivalence of two programs. However, as shown by Tsichritzis [21], this problem (and others) are decidable if loops are unnested (LOOP(1) programs).[b] We remark that, as such programs are always polynomial-time and feasible, the questions we study are trivial in such a restricted class. The analysis of LOOP(1) programs becomes more challenging as one expands the set of basic (non-looping) instructions, and several authors have studied decision problems (like equivalence) and the computational power of such programs [7–9, 12]. These works pointed out the connection of these programs to counter machines, and in this context it is interesting to note some similarity between our lossy assignments and the so-called *lossy counters* [15].

As previously mentioned, [2, 3] are the nearest ancestors of this work. To our best knowledge, the specific decidability questions studied in the current paper have not been studied before.

2. Basic Definitions

We will deal with languages that differ from L_{BJK} in certain respects. On one hand, we exclude nested expressions, the `skip` command and the `if` command; after all, we are going to prove a negative result, so we are interested in minimal fragments that suffice. On the other hand, we included the constants 0 and 1 (in the form of the reset instruction `X:=0` and the increment `X:=Y+1`).

Definition 1. *We will work with fragments and variants of the language whose syntax is given in Fig.* 3*. Programs in this language manipulate non-negative integers according to the standard semantics expected from the syntax. The following "cleanliness" restrictions are imposed on the syntax:*

- *in* `!loop X {C}`*, the variable* `X` *is not allowed to be assigned to within* `C`
- *in* `X:=Y`*,* `X:=Y+Z` *and* `X:=Y+1`*, the variables on the right-hand side are distinct from the left-hand side variable.*

We denote subsets of the language in Fig. 3 *by the letter L followed by a list of the right-hand sides of assignment commands. Hence, the full language shown is*

[b]The proof is exhibited among the *Gems of Theoretical Computer Science* [20].

$$\begin{aligned} \texttt{X},\texttt{Y},\texttt{Z} \in \text{Variable} &::= \texttt{X}_1 \mid \texttt{X}_2 \mid \texttt{X}_3 \mid \ldots \mid \texttt{X}_n \\ \texttt{C} \in \text{Command} &::= \texttt{X:=Y} \mid \texttt{X:=Y+Z} \mid \texttt{X:=0} \mid \texttt{X:=Y+1} \\ &\quad \mid \texttt{C}_1\texttt{;C}_2 \mid \texttt{!loop X \{C\}} \end{aligned}$$

Fig. 3. Syntax of a language we consider.

$L[\texttt{Y},\texttt{Y+Z},\texttt{0},\texttt{Y+1}]$*; and* $L[\texttt{Y},\texttt{Y+Z},\texttt{0}]$ *is the language shown without assignment of the form* $\texttt{X} := \texttt{Y+1}$*; and so on.*

We study three variants of the assignment command:

- *The standard assignment:* $\texttt{X} := \textit{exp}$*. This command sets* X *to the value of exp.*
- *The lossy assignment:* $\texttt{X} :\leq \textit{exp}$*. This command sets* X *to a non-deterministically chosen value that is less than or equal to exp.*
- *The max assignment:* $\texttt{X} :\stackrel{max}{=} \textit{exp}$*. This command sets* X *to the value of exp if this increases* X*; otherwise,* X *keeps its current value.*

We use $L^{\leq}[\ldots]$ *and* $L^{\max}[\ldots]$ *to denote the variants of the language* $L[\ldots]$ *that use, respectively, lossy assignments and max assignments in place of standard assignments.*

Let p *be a program over the variables* $\texttt{X}_1,\ldots,\texttt{X}_n$*, and let* $x_1,\ldots,x_n$ *be the values of respectively* $\texttt{X}_1,\ldots,\texttt{X}_n$ *when an execution of* p *starts. The program* p *is* feasible *if there exists a polynomial* p *such that* $p(x_1,\ldots,x_n)$ *bounds any value computed during the execution. We define the* feasibility problem for the language L *by*

- *input: an* L*-program* p*; question: is* p *feasible?*

We proved in [3] that the feasibility problem for L_{BJK} is decidable in polynomial time. In [2], we proved that the feasibility problem for L_{BJK} still is decidable when L_{BJK} is extended with the constant 0, moreover, we proved that this problem is PSPACE complete. We also argued that these decidability results hold for a few close variants of the feasibility problem:

- The *polynomial-bound problem* where the question is whether the value of a designated output variable upon program termination is polynomially bounded in the inputs.
- The *polynomial step-count problem* where the question is whether the number of primitive steps in the execution of the program is polynomially bounded in the inputs.

The decidability and undecidability results of the current paper do also apply to these variants, but we leave the proofs to the interested reader.

3. Undecidability Results

The next theorem should be well known, but we include a brief proof for the reader's convenience.

Theorem 2. *The feasibility problem for* $L[\mathtt{Y}, \mathtt{0}, \mathtt{Y+1}]$ *is undecidable.*

Proof. First we note that we indeed can subtract in this language. The following program sets `Y` to the the value of `X` minus 1 (when `X` is strictly greater than 0).

$$\mathtt{Y{:=}0;\ Z{:=}0;\ !loop\ X\ \{\ Y{:=}Z;\ Z{:=}Y{+}1\ \}}\,.$$

Hence, it should be obvious that the language is strong enough to compute any primitive recursive function.

Assume some standard enumeration of Turing machines taking no input, and let $T(e, s)$ be the predicate that holds if and only if the e'th Turing machine halts within $\log_2 s$ steps.[c] This primitive recursive predicate is of low complexity and can be decided by a feasible $L[\mathtt{Y}, \mathtt{0}, \mathtt{Y+1}]$-program. Let $\mathtt{p}_e$ be the $L[\mathtt{Y}, \mathtt{0}, \mathtt{Y+1}]$-program given by the informal code

$$\mathtt{if}\ T(e, \mathtt{X})\ \mathtt{then}\ \text{compute}\ 2^{\mathtt{X}}\ \mathtt{else}\ \text{do nothing}\,.$$

Now, $\mathtt{p}_e$ is feasible if and only if the e'th Turing machine does not halt. This proves that the halting problem for Turing machines is reducible to the feasibility problem for $L[\mathtt{Y}, \mathtt{0}, \mathtt{Y+1}]$, and thus, this feasibility problem is undecidable. □

Now, what happens if the programs apply lossy assignments in place of standard assignments? It turns out the feasibility problem is still undecidable, as we show in the next theorem.

Definition 3. *Let* `H` *be a variable not occurring in the program* `p`. *We use* $\mathcal{H}(\mathtt{p})$ *to denote the prorgam* `p` *where each assignment* $\mathtt{X} :\leq \mathit{exp}$ *is replaced by* $\mathtt{!loop\ H\{\,X} :\leq \mathit{exp}\,\mathtt{\}}$.

Observe that all our assignment commands are idempotent, that is, repeating them several times does not change their effect. Thus, the program $\mathcal{H}(\mathtt{p})$ computes the same values as `p` does if the initial value of `H` is nonzero. If the initial value of `H` is zero, the program will execute without changing the value of any variable. Moreover, if we insert an assignment into $\mathcal{H}(\mathtt{p})$ that sets `H` to 0, the program will be inhibited from computing any larger values from the moment where this assignments is executed.

To prove undecidability in weaker and weaker languages, we introduce *macros*, that is, pieces of code in the weaker language that simulate (in some sense) the missing commands. We use this technique to deal with the limitation of lossy assignments.

[c]This is essentially the predicate T defined by Kleene in Sect. 42 of [13], the definition adjusted to ensure polynomial complexity.

Definition 4. *The* macro `X :≤ Y-1` *stands for*

```
X:≤0; Z:≤0; !loop Y { X:≤Z; Z:≤X+1 }
```

where `Z` *is a fresh variable* (*that can be shared by all invocations of the macro*). *The* macro `X :≤ X-1` *stands for*

```
Z:≤X-1; X:≤Z
```

where `Z` *is a fresh variable* (*that can be shared by all invocations of the macro*).

Lemma 5. *The effect of the macro* `X :≤ Y-1` *is to set* `X` *to a value less than or equal to* $\mathtt{Y}-1$ (*and, importantly, equality is possible*).

Definition 6. *The* macro `X:=Y+1` *stands for*

```
X:≤Y+1; Z:≤X; !loop Y { Z:≤Z-1 }; H:≤Z
```

where `Z` *is a fresh variable* (*that can be shared by all invocations of the macro*).

Lemma 7. *The effect of the macro* `X:=Y+1` *is to set* `X` *to a value that is at most* $\mathtt{Y}+1$ (*any value less than or equal to* $\mathtt{Y}+1$ *is possible*). *Moreover, the macro also sets* `H` *to a value. This new value of* `H` *can be nonzero only if the new value of* `X` *is precisely* $\mathtt{Y}+1$.

The reader may now see how the macro `X:=Y+1` interacts with the transformation $\mathcal{H}$. If the macro fails to compute $\mathtt{Y}+1$ precisely, it sets `H` to zero, and thereby, inhibits the program.

Definition 8. *The* macro `X:=Y` *stands for*

```
X:≤Y; Z:≤X; !loop Y { H:≤Z; Z:≤H-1 }
```

where `Z` *is a fresh variable* (*that can be shared by all invocations of the macro*).

Lemma 9. *The effect of the macro* `X:=Y` *is to set* `X` *to a value that is at most* `Y` (*any value less than or equal to* `Y` *is possible*). *Moreover, the macro also sets* `H` *to a value. This new value of* `H` *can be nonzero only if the new value of* `X` *is precisely* `Y`.

Theorem 10. *Feasibility is undecidable for* $L^{\leq}[\mathtt{Y}, 0, \mathtt{Y+1}]$.

Proof. By reduction from feasibility for $L[\mathtt{Y}, 0, \mathtt{Y+1}]$. Given a program `p` in $L[\mathtt{Y}, 0, \mathtt{Y+1}]$, we form $\mathcal{H}(\mathtt{p})$, and then, we go on to replace all assignments by the corresponding macros, that either simulate them precisely or inhibit the rest of the computation from producing any larger values. The result is an $L^{\leq}[\mathtt{Y}, 0, \mathtt{Y+1}]$-program that is feasible if and only if `p` is feasible. □

Next, we will prove that the feasibility problem for programs with definite loops and lossy assignments remains infeasible when the constants 0 and 1 are removed from the language.

Theorem 11. *Feasibility is undecidable for* $L^{\leq}[\mathtt{Y}, \mathtt{Y+Z}, 0]$.

Proof. We have proved that the feasibility problem for $L^{\leq}[\mathtt{Y}, 0, \mathtt{Y+1}]$ is undecidable. Thus, the the feasibility problem for $L^{\leq}[\mathtt{Y}, \mathtt{Y+Z}, 0, \mathtt{Y+1}]$ will also be undecidable. We will reduce the feasibility problem for $L^{\leq}[\mathtt{Y}, \mathtt{Y+Z}, 0, \mathtt{Y+1}]$ to the the feasibility problem for $L^{\leq}[\mathtt{Y}, \mathtt{Y+Z}, 0]$.

Let $\mathtt{p}$ be a $L^{\leq}[\mathtt{Y}, \mathtt{Y+Z}, 0, \mathtt{Y+1}]$-program, and let $\mathtt{K1}$ be a fresh variable. The preamble

$$\mathtt{pre} \equiv \mathtt{H{:}\leq 0;\ !loop\ K1\ \{\ H{:}\leq\ Z;\ Z{:}\leq 0\ \}}$$

will inhibit the program $\mathcal{H}(\mathtt{p})$, unless the initial value of $\mathtt{K1}$ is 1 and the initial value of $\mathtt{Z}$ is nonzero. This allows us to simulate the command $\mathtt{X} :\leq \mathtt{Y+1}$ by $\mathtt{X} :\leq \mathtt{Y+K1}$. Hence, let $\mathtt{p}'$ be $\mathtt{p}$ where each assignment $\mathtt{X} :\leq \mathtt{Y+1}$ is replaced by $\mathtt{X} :\leq \mathtt{Y+K1}$, and let $\mathtt{q} \equiv \mathtt{pre}; \mathcal{H}(\mathtt{p}')$. Now, if the initial value of $\mathtt{K1}$ should be different from 1, or if initial value of $\mathtt{Z}$ should be 0, then the values computed by $\mathtt{q}$ will obviously be bounded by polynomials in the inputs; otherwise, for any variable $\mathtt{X}$ occurring $\mathtt{p}$, the program $\mathtt{q}$ will compute the same values into $\mathtt{X}$ as $\mathtt{p}$ does. Hence, $\mathtt{q}$ is an $L^{\leq}[\mathtt{Y}, \mathtt{Y+Z}, 0]$-program that is feasible if and only if $\mathtt{p}$ is feasible. □

Next, we we eliminate the constant $\mathtt{0}$, and thereby reduce the number of expression forms in assignments to one.

Theorem 12. *Feasibility is undecidable for* $L^{\leq}[\mathtt{Y+Z}]$.

Proof. We reduce from the feasibility problem for $L^{\leq}[\mathtt{Y}, \mathtt{Y+Z}, 0]$. Given $\mathtt{p}$ in the latter language, pick a fresh variable $\mathtt{K0}$, and replace each assignment $\mathtt{X} :\leq 0$ by $\mathtt{X} :\leq \mathtt{K0+K0}$ and each assignment $\mathtt{X} :\leq \mathtt{Y}$ by $\mathtt{X} :\leq \mathtt{Y+K0}$. If the initial value of $\mathtt{K0}$ is zero, this preserves the effect of the program, and reduces the language to $L^{\leq}[\mathtt{Y+Z}]$.

Now, as $\mathtt{K0}$ can take any initial value, we modify the program further by replacing any command $\mathtt{X} :\leq \mathtt{Y+Z}$ by

$$\mathtt{X{:}\leq Y+Z;\ !loop\ K0\ \{\ X{:}\leq\ K0+K0\ \}}\,.$$

This preserves the effect of the program if the initial value of $\mathtt{K0}$ is zero, while if the initial value of $\mathtt{K0}$ is nonzero, any value computed during an execution will be bounded by $2(x_1 + \ldots + x_n)$ where $x_1, \ldots, x_n$ are the inputs.

This proves that for any $L^{\leq}[\mathtt{Y}, \mathtt{Y+Z}, 0]$-program $\mathtt{p}$ we can construct an $L^{\leq}[\mathtt{Y+Z}]$-program $\mathtt{q}$ such that $\mathtt{q}$ is feasible if and only if $\mathtt{p}$ is feasible. □

The reductions in the proofs of Theorem 11 and 12 work equally well with exact assignments. Hence, we also have

Theorem 13. *Feasibility is undecidable for* $L[\mathtt{Y+Z}]$.

4. Decidability with Max Assignments

It turns out that the feasibility problems that we proved undecidable in the previous section become decidable if the programs apply max assignments in place of lossy assignments.

Let $?L[\ldots]$ denote the language $L[\ldots]$ equipped with indefinite loops in place of definite loops. The language $?L[\mathtt{Y}, \mathtt{Y+Z}]$ is a sub-language of L_{BJK}, and it follows from the results in [3] that the feasibility problem for $?L[\mathtt{Y}, \mathtt{Y+Z}]$ is decidable. We will prove that the feasibility problem for $L^{\max}[\mathtt{Y}, \mathtt{Y+Z}, \mathtt{0}]$ is decidable by reduction to the feasibility problem for $?L[\mathtt{Y}, \mathtt{Y+Z}]$.

Definition 14. *Let* $\mathcal{A}(\mathtt{p})$ *denote the* $?L^{\max}[\mathtt{Y}, \mathtt{Y+Z}]$*-program we obtain when each assignment* $\mathtt{X} :\overset{max}{=} \mathtt{0}$ *in the* $L^{\max}[\mathtt{Y}, \mathtt{Y+Z}, \mathtt{0}]$*-program* p *is replaced by the assignment* $\mathtt{X} :\overset{max}{=} \mathtt{X}$.

The next lemma is obvious.

Lemma 15. *Let* p *be any* $L^{\max}[\mathtt{Y}, \mathtt{Y+Z}, \mathtt{0}]$*-program. Then,* p *is feasible if and only if* $\mathcal{A}(\mathtt{p})$ *is feasible.*

Definition 16. *Let* $?(\mathtt{p})$ *denote the* $?L^{\max}[\mathtt{Y}, \mathtt{Y+Z}]$*-program we obtain when each definite loop* `!loop X {....}` *in the* $L^{\max}[\mathtt{Y}, \mathtt{Y+Z}]$*-program* p *is replaced by an indefinite loop* `?loop X {....}`.

Lemma 17. *Let* p *be any* $L^{\max}[\mathtt{Y}, \mathtt{Y+Z}]$*-program. Then,* p *is feasible if and only if* $?(\mathtt{p})$ *is feasible.*

Proof. An execution of the deterministic program p corresponds to a particular execution of the non-deterministic program $?(\mathtt{p})$. Hence, p is feasible if $?(\mathtt{p})$ is.

Now, assume that p is feasible. Let $\mathcal{A} = \mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_\ell$ be the sequence of assignment statements performed when p is executed on some inputs $x_1, \ldots, x_n$. When $?(\mathtt{p})$ is executed, a loop `?loop Z [...]` will be executed at most Z times, in contrast to exactly Z times in p. Let $\mathcal{A}^? = \mathtt{p}_1^?, \ldots, \mathtt{p}_k^?$ be the sequence of assignment statements performed when $?(\mathtt{p})$ is executed on the same inputs $x_1, \ldots, x_n$. We claim that the statements $\mathcal{A}^?$ form a subsequence of $\mathcal{A}$. Further, we claim that the values of variables after any prefix of $\mathcal{A}^?$ are bounded by the values of the respective variables after a corresponding prefix of $\mathcal{A}$. This is easy to prove by induction on the length of the prefix, based on the fact that variables cannot be decreased. We conclude that $?(\mathtt{p})$ is feasible if p is feasible. □

Definition 18. *Let* p *be a* $?L^{\max}[\mathtt{Y}, \mathtt{Y+Z}]$*-program, and let* W *be a fresh variable. We use* $\mathcal{C}(\mathtt{p})$ *to denote the* $?L[\mathtt{Y}, \mathtt{Y+Z}]$*-program we obtain when each assignment* $\mathtt{X} :\overset{max}{=} exp$ *in* p *is replaced by* `?loop W {` X := *exp* `}`.

Note that the command `?loop W {` X := *exp* `}` either performs no assignments, or performs the assignment X := *exp* a number of times. Thus, as our standard

assignment commands are idempotent, we can without loss of generality assume the that program either does nothing, or performs the assignment $\mathtt{X} := \mathit{exp}$ exactly once.

Lemma 19. *Let* $\mathtt{p}$ *be any* $?L^{\max}[\mathtt{Y}, \mathtt{Y{+}Z}]$*-program. Then,* $\mathtt{p}$ *is feasible if and only if* $\mathcal{C}(\mathtt{p})$ *is feasible.*

Proof. Any execution of $\mathtt{p}$ corresponds to a particular execution of $\mathcal{C}(\mathtt{p})$, namely the execution that does nothing whenever the execution of $\mathtt{p}$ performs $\mathtt{X} :\overset{\max}{=} \mathit{exp}$ and $\mathtt{X} \geq \mathit{exp}$; and performs $\mathtt{X} := \mathit{exp}$ whenever the execution of $\mathtt{p}$ performs $\mathtt{X} :\overset{\max}{=} \mathit{exp}$ and $\mathtt{X} < \mathit{exp}$. Hence, $\mathcal{C}(\mathtt{p})$ is infeasible if $\mathtt{p}$ is infeasible.

Now, assume that $\mathtt{p}$ is feasible. Let $\mathcal{C} = \mathtt{p}_1^c, \ldots, \mathtt{p}_\ell^c$ be the sequence of assignment statements performed in a certain computation of $\mathcal{C}(\mathtt{p})$ with some inputs $x_1, \ldots, x_n$. We claim that there is a computation of $\mathtt{p}$ on the same inputs $x_1, \ldots, x_n$ that performs a sequence of assignments $\mathcal{A} = \mathtt{p}_1, \ldots, \mathtt{p}_k$, such that the sequence $\mathcal{C}$ forms a subsequence of $\mathcal{A}$. Further, we claim that the values of variables after any prefix of $\mathcal{C}$ are bounded by the values of the respective variables after a corresponding prefix of $\mathcal{A}$. This is easy to prove by induction on the length of the prefix, noting the following cases regarding the command `?loop W {`$\mathtt{X} := \mathit{exp}$`}`: (1) no assignment takes place; in $\mathtt{p}$, a corresponding assignment is performed, which might only increase the value of $\mathtt{X}$, maintaining the invariant. (2) An assignment takes place: a corresponding assignment would be performed by $\mathtt{p}$. There may be a difference in the result, but the invariant will be maintained, because the $\mathtt{X} :\overset{\max}{=} \mathit{exp}$ command, performed in $\mathtt{p}$, can only result in a value at least as big as the one assigned to $\mathtt{X}$ in $\mathcal{C}(\mathtt{p})$.

Since there are polynomials bounding the values computed by $\mathtt{p}$ in terms of the input, we deduce that the same polynomials bound the results of $\mathcal{C}(\mathtt{p})$, which is therefore feasible. □

Theorem 20. *The feasibility problem for* $L^{\max}[\mathtt{Y}, \mathtt{Y{+}Z}, 0]$ *is decidable. Moreover, the problem is in PTIME.*

Proof. Let $\mathtt{p}$ be an $L^{\max}[\mathtt{Y}, \mathtt{Y{+}Z}, 0]$-program. By the preceding lemmas, $\mathtt{p}$ is feasible if and only if the program $\mathcal{C}(?(\mathcal{A}(\mathtt{p})))$ is feasible. Moreover, $\mathcal{C}(?(\mathcal{A}(\mathtt{p})))$ is a $?L[\mathtt{Y}, \mathtt{Y{+}Z}]$-program and the feasibility problem for $?L[\mathtt{Y}, \mathtt{Y{+}Z}]$ is decidable.

The feasibility problem for $?L[\mathtt{Y}, \mathtt{Y{+}Z}]$ is in PTIME [3]. The program $\mathcal{C}(?(\mathcal{A}(\mathtt{p})))$ can obviously be constructed from $\mathtt{p}$ by a Turing machine working in polynomial time. Hence, the feasibility problem for $L^{\max}[\mathtt{Y}, \mathtt{Y{+}Z}, 0]$ is also in PTIME. □

We conjecture that the feasibility problem for $L^{\max}[\mathtt{Y}, \mathtt{Y{+}Z}, 0, \mathtt{Y{+}1}]$ is decidable as well. Let $L_{\text{BJK}} \cup \{\mathtt{0}, \mathtt{Y{+}1}\}$ denote the language L_{BJK} extended with assignments of the forms $\mathtt{X} := 0$ and $\mathtt{X} := \mathtt{Y{+}1}$. We do not know if the decidability problem for $L_{\text{BJK}} \cup \{\mathtt{0}, \mathtt{Y{+}1}\}$ is decidable. But if it is, that would imply decidability of the feasibility problem for $L^{\max}[\mathtt{Y}, \mathtt{Y{+}Z}, 0, 1]$ by a reduction alike the one shown earlier.

However, the feasibility problem for $L_{\text{BJK}} \cup \{\texttt{0}, \texttt{Y+1}\}$ seems to be significantly more difficult then the feasibility problem for $L^{\max}[\texttt{Y}, \texttt{Y+Z}, \texttt{0}, \texttt{1}]$.

5. Open Problems

Definition 21. *Let* $\texttt{p}$ *be a program over the variables* $\texttt{X}_1, \ldots, \texttt{X}_n$*, and let* $x_1, \ldots, x_n$ *be the values of respectively* $\texttt{X}_1, \ldots, \texttt{X}_n$ *when an execution of* $\texttt{p}$ *starts. The program* $\texttt{p}$ *is* feasible almost always (a.a.) *if there exist a polynomial* p *and* $k \in \mathbb{N}$ *such that* $p(x_1, \ldots, x_n)$ *bounds any value computed during the execution when* $x_1, \ldots, x_n > k$*. We define the* feasibility almost always (a.a.) problem for L *by*

- *input: an L-program* $\texttt{p}$*; question: is* $\texttt{p}$ *feasible (a.a.)?*

A feasible program will also be feasible (a.a.), but there exists infeasible programs that are feasible (a.a.), e.g. the program given by the informal code

$$\texttt{!loop Z \{X := 0\}; !loop X \{}\,\text{compute } 2^{\texttt{Y}}\,\texttt{\}}\,.$$

Our proof of undecidability of the feasibility problem for $L[\texttt{Y+Z}]$ depends on the behaviour of programs when certain inputs are small, specifically, when certain inputs are 0 or 1. Thus, this proof will not work for feasibility (a.a.). Neither will it work for the variant of feasibility given in the next definition.

Definition 22. *Let* $\texttt{p}$ *be a program over the variables* $\texttt{X}, \texttt{Y}_1, \ldots, \texttt{Y}_n$*, and let* $x, 0, 0, \ldots, 0$ *be the values of respectively* $\texttt{X}, \texttt{Y}_1, \ldots, \texttt{Y}_n$ *when the execution of* $\texttt{p}$ *starts. The program* $\texttt{p}$ *is* feasible with single input (w.s.i.) *if there exist a polynomial* p *such that* $p(x)$ *bounds any value computed during the execution. We define the* feasibility with single input (w.s.i.) problem for L *by*

- *input: an L-program* $\texttt{p}$*; question: is* $\texttt{p}$ *feasible (w.s.i.)?*

It is easy to see that the (a.a.) variant and the (w.s.i.) variant of the feasibility problem for $L[\texttt{Y}, \texttt{Y+1}, \texttt{0}]$ are undecidable. (The program $\texttt{p}_e$ constructed in the proof of Theorem 2 is feasible (a.a.) if and only if the e'th Turing machine does not halt. Moreover, the same program $\texttt{p}_e$ is feasible if and only if the e'th Turing machine does not halt, even when all variables, except a particular one, are initiated to 0.) However, we think that the feasibility (a.a.) problem, or the feasibility (w.s.i.) problem, or both problems, might be decidable for the language $L[\texttt{Y}, \texttt{Y+Z}, \texttt{0}]$. But this is a conjecture, and of course, if any of these two problems should turn out to be undecidable, the proof for this has to be very different from the proof of undecidability of the feasibility problem for $L[\texttt{Y+Z}]$. Thus, we pose two open problems:

- Is the feasibility (a.a.) problem for $L[\texttt{Y}, \texttt{Y+Z}, \texttt{0}]$ decidable?
- Is the feasibility (w.s.i.) problem for $L[\texttt{Y}, \texttt{Y+Z}, \texttt{0}]$ decidable?

We remark that both problems indeed are undecidable for $L^{\leq}[\texttt{Y+Z}]$. The reason is that we can use lossy assignments to generate small values. The proofs in Section 3

works equally well if, e.g., the dedicated variable `K1` receives the critical value 1 from a lossy assignment `K1` $:\leq$ `X`, in stead of receiving this value as an input.

The feasibility (w.s.i.) problem for $L^{\max}[\texttt{Y+Z}, \texttt{0}]$ is decidable as it can be reduced to the feasibility problem for L_{BJK} extended with the constant `0` — a problem proven to be decidable in [2]. We conjecture that the feasibility (w.s.i.) and feasibility (a.a.) problems for $L^{\max}[\texttt{Y+Z}, \texttt{0}, \texttt{1}]$ are decidable as well.

6. Conclusion

The next table summarizes our results on decidability of the feasibility problem for various kinds of loop programs. The parameters are: type of loop and assignment constructs, and which constants are allowed. In the decidable cases, a multiplication `X:=Y*Z` can also be included [2, 3].

expressions:	`X+Y`	`X+Y`,0	`X+Y`,0, `Y+1`
indefinite loops all types of assignments	PTIME [3]	PSPACE [2]	?
definite loops max assignments	PTIME	PTIME	?
definite loops lossy/standard assignment	undecidable	undecidable	undecidable

To this we can add the (mostly open) problems regarding feasibility *almost always* or *with a single input*, as described in Section 5.

References

[1] A. Adachi, T. Kasai and E. Moriya, A theoretical study of the time analysis of programs, in *Mathematical Foundations of Computer Science 1979*, ed. J. Bečvář (volume 74 of LNCS, Springer-Verlag), pp. 201–207

[2] A. M. Ben-Amram, On decidable growth-rate properties of imperative programs, in *International Workshop on Developments in Implicit Computational complExity (DICE 2010)*, ed. P. Baillot (volume 23 of EPTCS, ArXiv.org, 2010), pp. 1–14.

[3] A. M. Ben-Amram, N. D. Jones and L. Kristiansen, Linear, polynomial or exponential? Complexity inference in polynomial time, in *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe (CiE 2008)*, eds. A. Beckmann, C. Dimitracopoulos and B. Löwe (volume 5028 of LNCS, Springer, 2008), pp. 67–76.

[4] P. Cousot and R. Cousot, Static determination of dynamic properties of programs, in *Proceedings of the Second International Symposium on Programming*, (Dunod, Paris, France, 1976), pp. 106–130.

[5] P. Cousot and R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Fourth ACM Symposium on Principles of Programming Language (POPL)* (ACM Press, New York, 1977), pp. 238–252.

[6] P. Cousot and R. Cousot, Abstract interpretation frameworks, *Journal of Logic and Computation*. **2**(1992) 511–547.

[7] E. M. Gurari and O. H. Ibarra, The complexity of the equivalence problem for simple programs, *J. ACM.* **28**(1981) 535–560.
[8] E. M. Gurari, Decidable problems for powerful programs, *J. ACM.* **32**(1985) 466–483.
[9] O. H. Ibarra, B. S. Leininger and L. E. Rosier, A note on the complexity of program evaluation, *Theory of Computing Systems.* **17**(1984) 85–96.
[10] N. D. Jones and L. Kristiansen, A flow calculus of mwp-bounds for complexity analysis, *ACM Trans. Computational Logic.* **10**(2009) 1–41.
[11] T. Kasai and A. Adachi, A characterization of time complexity by simple loop programs, *Journal of Computer and System Sciences.* **20**(1980) 1–17.
[12] A. J. Kfoury, Analysis of simple programs over different sets of primitives, in *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, USA, 1980), pp. 56–61.
[13] S. C. Kleene, *Mathematical Logic* (John Wiley & Sons, New York, 1967; republished by Dover, 2002).
[14] L. Kristiansen and K.-H. Niggl, On the computational complexity of imperative programming languages, *Theor. Comp. Sci.* **318**(2004) 139–161.
[15] R. Mayr. Undecidable problems in unreliable computations. *Theor. Comput. Sci*, **297**(2003) 337–354.
[16] D. Le Métayer, ACE: an automatic complexity evaluator, *ACM Trans. Program. Lang. Syst.* **10**(1988) 248–266.
[17] A. R. Meyer and D. M. Ritchie, The complexity of loop programs, in *Proc. 22nd ACM National Conference* (Washington, DC, 1967), pp. 465–469.
[18] K.-H. Niggl and H. Wunderlich, Certifying polynomial time and linear/polynomial space for imperative programs, *SIAM J. Comput*, **35**(2006) 1122–1147.
[19] M. Rosendahl, Automatic complexity analysis, in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'89)* (ACM, 1989), pp. 144–156.
[20] U. Schöning and R. Pruim, *Gems of Theoretical Computer Science* (Springer-Verlag, Berlin, 1998).
[21] D. Tsichritzis, The equivalence problem of simple programs, *J. ACM.* **17**(1970) 729–738.
[22] M. van Eekelen and O. Shkaravska (eds.), *International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA '09)* (volume 6324 of LNCS, Springer, 2010).
[23] B. Wegbreit, Mechanical program analysis, *Communications of the ACM.* **18**(1975) 528–539.