Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time (Extended Abstract)

Amir M. Ben-Amram[1,⋆], Neil D. Jones[2], and Lars Kristiansen[3]

[1] School of Computer Science, Tel-Aviv Academic College, Israel
[2] DIKU, the University of Copenhagen, Denmark
[3] Department of Mathematics, University of Oslo, Norway
amirben@mta.ac.il, neil@diku.dk, larskri@iu.hio.no

Abstract. We present a new method for inferring complexity properties for imperative programs with bounded loops. The properties handled are: polynomial (or linear) boundedness of computed values, as a function of the input; and similarly for the running time.

It is well known that complexity properties are undecidable for a Turing-complete programming language. Much work in program analysis overcomes this obstacle by relaxing the correctness notion: one does not ask for an algorithm that correctly decides whether the property of interest holds or not, but only for "yes" answers to be sound. In contrast, we reshaped the problem by defining a "core" programming language that is Turing-incomplete, but strong enough to model real programs of interest. For this language, our method is the first to give a certain answer; in other words, our inference is both sound and complete.

The essence of the method is that every command is assigned a "complexity certificate", which is a concise specification of dependencies of output values on input. These certificates are produced by inference rules that are compositional and efficiently computable. The approach is inspired by previous work by Niggl and Wunderlich and by Jones and Kristiansen, but use a novel, more expressive kind of certificates.

Keywords: implicit computational complexity, polynomial time complexity, linear time complexity, static program analysis.

1 Introduction

Central to the field of Implicit Computational Complexity (ICC) is the following fundamental observation: it is possible to restrict a programming language syntactically so that the admitted programs will possess a certain complexity, say polynomial time. Such results lead to a sweet dream: the "complexity-certifying compiler," that will warn us whenever we compile a non-polynomial algorithm.

Since (as is well known) deciding such a property precisely for any program in a Turing-complete language is impossible, the goal in this line of research is

⋆ Research performed while visiting DIKU, the University of Copenhagen, Denmark.

to extend the capabilities of the certifying compiler as much as possible, while taking into account the price: in other words, explore the tradeoff between the completeness of the method and the method's complexity. At any rate, we insist on the certification being sound—no bad programs shall pass.

The method presented in this paper is a step forward in this research program. It can be used to certify programs as having a running time that is polynomial in certain input values; we also present a variant for certifying linear time. Further, we determine which variables have values that are polynomially (or linearly) bounded. The latter is, in fact, the essential problem: since we will only consider bounded loops, deriving bounds on the running time amounts to deriving a bound on the counters that govern the loops.

Our work complements previous research by Kristiansen and Niggl [KN04], Niggl and Wunderlich [NW06], and Jones and Kristiansen [JK08]. All methods apply to a structured imperative language. The last two, in particular, take the form of a compositional calculus of certificates—to any command a "certificate" is assigned which encodes the certified properties of the command, and a certificate for a composite command is derived from those of its parts. We keep this elegant structure, but use a new kind of certificates designed to accurately discern phenomena such as *accumulator variables* in loops (think of a command `X := X+Y` within a loop) as well as *self-multiplying variables* (`X := X+X`).

Normally, in theoretical research such as this, one does not bother with algorithms for analysing a full-featured practical programming language, but considers a certain *core language* that embodies the features of algorithmic interest. It is well known that one can strip a lot of "syntactic sugar" out of any practical programming language to obtain such a core which is still Turing-complete, and once a problem is solved for the core, it is as good as solved for the full language (up to implementing the appropriate translations). We go a step further by proposing that if certain features are *beyond the scope* or our analysis, getting rid of them is best done in the passage to the core language. Here is the prototypical example: Very often, program analyses take a conservative approach to modelling conditionals: both branches are treated as possible. Since our analysis also does so, we include in our core language only the following form for the conditional command: `if ? then C1 else C2` .

Here `C1`, `C2` represent commands, while the question mark represents that the conditional expression is hidden. In the core language, this command has a *non-deterministic* semantics. Thus, the passage to the core language is an *abstracting* translation: it abstracts away features that we overtly leave out of the scope of our analysis. Our point of view is that it is beneficial to separate the concern of parsing and abstracting practical programs (the *front end*) from the concern of analysing the core language (the *back end*). A simple-minded abstraction (e.g., really just hiding the conditionals) is clearly doable, so there should be no doubt that a front end for a realistic language *can* be built. Current *static analysis* technology allows the construction of sophisticated front ends. Our theoretic effort will concentrate on the "back end"—analysing the core language. The reader

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

Fig. 1. Syntax of the core language. Variables hold nonnegative integers.

may want to peek at Figure 1, showing the syntax of the language. Its semantics is almost self-explanatory, and is made precise in the next section.

The main result in this paper is a proof that the problems of polynomial and linear boundedness are decidable for the core language. Our certification method solves this problem completely: for example, for the problem of polynomial running time, we will certify a core-language program *if and only if* its time is polynomially bounded. Furthermore, the analysis itself takes *polynomial time.*

A brief comparison with previous work. Both previous work we mentioned do not use a core language to set a clear abstraction boundary. However, [JK08] treats (implicitly) the same core language. Its inferences are sound, but incomplete, and its complexity appears to be non-polynomial (this paper is, essentially, a journal version of [JK05]—where completeness was wrongly claimed). The (implicit) core language treated by [NW06] can be viewed as an extension of our core language. When applied to our language, their method too is sound but incomplete (its complexity is polynomial-time, like ours).

2 Problem Definition

The *syntax* of our core language is described in Figure 1. In a command `loop X {C}`, variable `X` is not allowed to appear on the left-hand side of an assignment in the loop body `C`.

Data. It is most convenient to assume that the only type of data is nonnegative integers. More generality is possible but will not be treated here.

Command semantics. As already explained, the core language is nondeterministic. The `if` command represents a nondeterministic choice. The *loop command* `loop` $\mathtt{X}_\ell$ `{C}` repeats `C` a number of times bounded by the value of $\mathtt{X}_\ell$. Thus, it is also nondeterministic, and may be used to model different kinds of loops (for-loops, while-loops) as long as a bounding variable can be statically determined.

While the use of bounded loops restricts the computable functions to the primitive recursive class, this is still rich enough to make the problem challenging (and it can still be pushed to undecidability, if we give up the abstraction and include conventional, deterministic conditionals, such as an equality test[1]).

[1] Undecidability for such a language can be proved by a reduction from Hilbert's 10th problem.

The formal semantics associates with every command $\mathtt{C}$ over variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$ a relation $[\![\mathtt{C}]\!] \subseteq \mathbb{N}^n \times \mathbb{N}^n$. In the expression $\vec{x}[\![\mathtt{C}]\!]\vec{y}$, vector $\vec{x}$ (respectively $\vec{y}$) is the store before (after) the execution of $\mathtt{C}$.

The semantics of `skip` is the identity. The semantics of an assignment leaves some room for variation: either the precise value of the expression is assigned, or a nonnegative integer bounded by that value. The latter definition is useful for abstracting non-arithmetic expressions that may appear in a real-life program. Because our analysis only derives monotone increasing value bounds, this choice does not affect the results. Finally, composite commands are described by the straight-forward equations:

$$\begin{aligned}
[\![\mathtt{C}_1; \mathtt{C}_2]\!] &= [\![\mathtt{C}_2]\!] \circ [\![\mathtt{C}_1]\!] \\
[\![\mathtt{if\ ?\ then\ C}_1\ \mathtt{else\ C}_2]\!] &= [\![\mathtt{C}_1]\!] \cup [\![\mathtt{C}_2]\!] \\
[\![\mathtt{loop}\ \mathtt{X}_\ell\ \{\mathtt{C}\}]\!] &= \{(\vec{x}, \vec{y}) \mid \exists i \le x_\ell : \vec{x}[\![\mathtt{C}]\!]^i \vec{y}\}
\end{aligned}$$

where $[\![\mathtt{C}]\!]^i$ represents $[\![\mathtt{C}]\!] \circ \cdots \circ [\![\mathtt{C}]\!]$ (i occurrences of $[\![\mathtt{C}]\!]$).

For every command we also define its *step count* (informally referred to as running time). For simplicity, the step count of an atomic command is defined as 1. The step count of a loop command is the sum of the step counts of the iterations taken. Because of the nondeterminism in `if` and `loop` commands, the step count is also a relation. We also refer to the *iteration count*, which only grows by one each time a loop body is entered. The iteration count is linearly related to the step count, but is easier to analyse.

Goals of the analysis. Our *polynomial-bound calculus* (Section 3) reveals, for any given command, which variables are bounded throughout any computation by a polynomial in the input variables[2]. The *linear-bound calculus* (Section 4) identifies linearly-bounded variables instead. Finally, Section 5 extends these methods to characterize commands where the maximum step count is bounded polynomially (respectively, linearly).

As a by-product, the analysis reveals which inputs influence any specific output variable.

An example. In the following program, all variables are polynomially bounded (we invite the reader to check); this is not recognized by the previous methods. In the next section we explain how the difficulty illustrated by this example was overcome.

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

[2] Thus, as pointed out by one of the reviewers, the title of this paper is imprecise: we distinguish polynomial growth from *super-polynomial* one, be it exponential or worse.

3 A Calculus to Certify Polynomial Bounds

Our calculus can be seen as a set of rules for *abstract interpretation* of the core language, in the sense of [Cou96][3]. The maximal output value resulting of a given command `C` is some function f of the input values $x_1, \ldots, x_n$. There are infinitely many possible functions; we map each one into an *abstract value* of which there are finitely many. Each abstract value V is associated with a *concretisation* $\gamma(V)$ which is a (possibly infinite) set of functions such that f is bounded by one of them.

Let $\mathbb{D} = \{0, 1, 1^+, 2, 3\}$ with order $0 < 1 < 1^+ < 2 < 3$. Informally, $\mathbb{D}$ is a set of *dependency types*, describing how a result depends on an input, as follows:

value	3	2	1^+	1	0
dependency type	at least exponential	polynomial	additive	copy	none

The notation $[x = y]$ below denotes the value 1 if $x = y$ and 0 otherwise.

Definition 1. *Let $V \in \mathbb{D}^n$. The concretisation $\gamma(V)$ includes all functions defined by the following rules, and none others. (1) If there is an i such that $V_i = 1$, then $\gamma(V)$ includes $f(\vec{x}) = x_i$. (2) $\gamma(V)$ includes all polynomials of form $(\sum_i a_i x_i) + P(\vec{x})$, where $a_i \le [V_i = 1^+]$, and P is a polynomial of non-negative coefficients depending only on variables x_i such that $V_i = 2$. (3) If there is an i such that $V_i = 3$, then $\gamma(V)$ is the set of all n-ary functions over $\mathbb{N}$.*

A core-language expression `e` obviously describes a polynomial and it is straight-forward to obtain a minimal vector $\alpha(\mathtt{e})$ such that $\gamma(\alpha(\mathtt{e}))$ includes that polynomial.

A basic idea (going back to [NW06]) is to approximate the relation $\vec{x}[\![\mathtt{C}]\!]\vec{y}$ by a set of vectors $V_1, \ldots, V_n$ that describe the dependence of $y_1, \ldots, y_n$ respectively on $\vec{x}$. We combine the vectors into a matrix $M \in \mathbb{D}^{n \times n}$ where column j is V_j. Thus, M_{ij} is the dependency type of y_j on x_i. A complementary and useful view is that M_{ij} describes a *data-flow* from x_i to x_j. In fact, M can be viewed as a bipartite, labeled digraph where the left-hand (source) side represents the input variables and the right-hand (target) side represents the output. The set of arcs $\mathbf{A}(M)$ is the set $\{i \to j \mid M_{ij} \ne 0\}$. A list of arcs may also be more readable than a matrix. For example, consider the command `loop` $\mathtt{X}_3$ `{`$\mathtt{X}_1$`:=` $\mathtt{X}_1$`+` $\mathtt{X}_2$`}` or the command $\mathtt{X}_1$:= $\mathtt{X}_1$+$\mathtt{X}_3$*$\mathtt{X}_2$. Both are described (in the most precise way) by the following collection of arcs: $\mathtt{X}_2 \xrightarrow{1} \mathtt{X}_2,\ \mathtt{X}_3 \xrightarrow{1} \mathtt{X}_3,\ \mathtt{X}_1 \xrightarrow{1^+} \mathtt{X}_1,\ \mathtt{X}_2 \xrightarrow{2} \mathtt{X}_1,\ \mathtt{X}_3 \xrightarrow{2} \mathtt{X}_1$ or, as a matrix, $\begin{bmatrix} 1^+ & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}$.

Such matrices/graphs make an elegant abstract domain (or "certificates") for commands because of the ease in which the certificate for a composite command

[3] Abstract interpretation is a well-developed theoretical framework, which can shed light on our algorithm. However in this paper we avoid relying on prior knowledge of abstract interpretation.

can be derived. Let $\sqcup$ denote the LUB operation on $\mathbb{D}$, and extend it to matrices (elementwise). If M_1, M_2 describe commands $\mathtt{C}_1, \mathtt{C}_2$, it is not hard to see that $M_1 \sqcup M_2$ describes `if ? then` $\mathtt{C}_1$ `else` $\mathtt{C}_2$. For transferring the sequential composition of commands to matrices, we define the following operation: $a \cdot b$ is 0 if either a or b is 0, and otherwise is the largest of a and b. Intuitively, composition should be represented by matrix product using $\cdot$ as "multiplication." The problem, though, is what result an "addition" $\oplus$ of 1^+s should be. Consider the commands

$$\begin{array}{lcl} \mathtt{C1} & = & \mathtt{X}_2 \ \mathtt{:=}\ \mathtt{X}_1\mathtt{;}\ \mathtt{X}_3\ \mathtt{:=}\ \mathtt{X}_1 \\ \mathtt{C2} & = & \mathtt{X}_1\ \mathtt{:=}\ \mathtt{X}_2\ \mathtt{+}\ \mathtt{X}_3 \end{array}$$

The graph representing `C1` has arcs $\mathtt{X}_1 \xrightarrow{1} \mathtt{X}_2,\ \ \mathtt{X}_1 \xrightarrow{1} \mathtt{X}_3$ and the graph for `C2` has $\mathtt{X}_2 \xrightarrow{1^+} \mathtt{X}_1,\ \ \mathtt{X}_3 \xrightarrow{1^+} \mathtt{X}_1$. Hence entry M_{11} of the matrix for `C1; C2` has value $1 \cdot 1^+ \oplus 1 \cdot 1^+ = 1^+ + 1^+$. And for this example, the right answer is 2, because the command doubles $\mathtt{X}_1$ (and in a loop, $\mathtt{X}_1$ will grow exponentially). However, the graph for $\mathtt{C1}' =$ `if ? then` $\mathtt{X}_2$ `:=` $\mathtt{X}_1$ `else` $\mathtt{X}_3$ `:=` $\mathtt{X}_1$ also includes the arcs $\mathtt{X}_1 \xrightarrow{1} \mathtt{X}_2,\ \ \mathtt{X}_1 \xrightarrow{1} \mathtt{X}_3$, but when $\mathtt{C1}'$ is combined with `C2`, no doubling occurs.

Our conclusion is that matrices are just not enough, and in order to allow for compositional computation, our certificates retain additional information. Basically, we add to the matrices another piece of data which distinguishes between a pair of 1's that arises during a single computation path (as in `C1`) and a pair that arises as alternatives (as in $\mathtt{C1}'$). We next move to the formal definitions.

3.1 Data Flow Relations

A few notations: We use $\mathbf{A^1}(M)$ to denote the set of arcs labeled by $\{1, 1^+\}$. For any set S, $C_2(S)$ is the set of 2-sets (unordered pairs) over S. For $M \in \mathbb{D}^{n\times n}$, we define $r(M)$ to be $C_2(\mathbf{A^1}(M))$. The identity matrix I has 1 on the diagonal and 0 elsewhere.

A *dataflow relation*, or DFR, is a pair (M, R) where $M \in \mathbb{D}^{n\times n}$ (and has the meaning described above) and $R \subseteq C_2(\mathbf{A^1}(M))$. Thus, R consists of pairs of arcs.

For compactness, instead of writing $\{i \to j, i' \to j'\} \in R$ we may write $R(i, j, i', j')$.

Definitions: Operations on matrices and DFRs.

1. $A \otimes B$ is $(\sqcup, \cdot)$ matrix product over $\mathbb{D}$.
2. $(M_1, R_1) \sqcup (M_2, R_2) \overset{def}{=} (M_1 \sqcup M_2, (R_1 \cup R_2) \cap C_2(\mathbf{A^1}(M_1 \sqcup M_2)))$.
3. $(M, R) \cdot (M', R') \overset{def}{=} (M'', R'')$, where:

$$\begin{aligned} M'' &= (M \otimes M') \sqcup \{i \xrightarrow{2} j \mid \exists s \neq t.R(i, s, i, t) \wedge R'(s, j, t, j)\} \\ R'' &= \{\{i \to j, i' \to j'\} \in C_2(\mathbf{A^1}(M'')) \mid \exists s, t.R(i, s, i', t) \wedge R'(s, j, t, j')\} \\ &\quad \cup \{\{i \to j, i \to j'\} \in C_2(\mathbf{A^1}(M'')) \mid \exists s.(i, s) \in \mathbf{A^1}(M) \wedge R'(s, j, s, j')\} \\ &\quad \cup \{\{i \to j, i' \to j\} \in C_2(\mathbf{A^1}(M'')) \mid \exists s.R(i, s, i', s) \wedge (s, j) \in \mathbf{A^1}(M')\}. \end{aligned}$$

Observe how the rule defining M'' uses the information in the R-parts to identify computations that double an input value by adding two copies of it, a situation described by *the diamond*:

$$\begin{tikzcd}
& s \arrow[dr] & \\
i \arrow[ur] \arrow[dr] & & j \\
& t \arrow[ur] &
\end{tikzcd}$$

The reader may have guessed that if this situation occurs in analysing a program, the two meeting arcs will necessarily be labeled with 1^+.

Proposition 1. *The product is associative and distributes over* $\sqcup$, *i.e.,*
$(M,R)\cdot((M_1,R_1)\sqcup(M_2,R_2)) = ((M,R)\cdot(M_1,R_1))\sqcup((M,R)\cdot(M_2,R_2))\,.$

4. Powers: defined by $(M,R)^0 = (I, r(I))$ and $(M,R)^{i+1} = (M,R)^i\cdot(M,R)$.
5. Loop Correction: for a DFR (M,R), define $LC_\ell(M,R) = (M',R')$ where M' is identical to M except that:
   (a) For all j such that $M_{jj}\geq 2$, $M'_{\ell j} = 3$;
   (b) For all j such that $M_{jj} = 1^+$, $M'_{\ell j} = M_{\ell j}\sqcup 2$;
   and $R' = R\cap C_2(\mathbf{A^1}(M'))$.
   Remarks: Rule (a) reflects the exponential growth that results from multiplying a value inside a loop. If $\mathtt{X}_j$ is doubled, it will end up multiplied by 2^{x_ℓ}. Rule (b) reflects the behaviour of *accumulator variables.* Intuitively, $M_{jj} = 1^+$ reveals that some quantity y is added to $\mathtt{X}_j$ in the loop. Therefore, the effect of the loop will be to add $x_\ell\cdot y$, hence the correction to $M_{\ell j}$.
6. Loop Closure: For a given ℓ, the loop closure with respect to $\mathtt{X}_\ell$ is the limit (M^*,R^*) of the series (M_i,R_i) where:

$$(\star)\qquad \begin{aligned}(M_0,R_0) &= (I, r(I))\\ (M_1,R_1) &= (M_0,R_0)\sqcup(M,R)\\ (M_{i+1},R_{i+1}) &= (LC_\ell(M_i,R_i))^2\end{aligned}$$

   This closure can be computed in a finite (polynomial) number of steps because the series is an increasing chain in a semilattice of polynomial height. A more natural specification of the closure may be the following: (M^*,R^*) is the smallest DFR that is at least $(I, r(I))$ and is fixed under both multiplication by (M,R) and under Loop Correction (the meaning of "smallest" and "at least" has yet to be made precise). However, the definition above (as a limit) is the useful one from the algorithmic point of view.

Calculation of DFRs

The following inference rules associate a DFR with every core-language command. The association of (M,R) with command `C` is expressed by the judgment $\vdash \mathtt{C} : M, R$.

(Skip) $$\frac{}{\vdash \mathtt{skip} : I, r(I)}$$

(Assignment) $$\frac{\alpha(\texttt{e}) = V}{\vdash \texttt{X}_i\texttt{:=}\ \texttt{e} : M, r(M)}$$

where M is obtained from I by replacing the ith column with V.

(Choice) $$\frac{\vdash \texttt{C}_1 : M_1, R_1 \quad \texttt{C}_2 : M_2, R_2}{\vdash \texttt{if?then}\,\texttt{C}_1\,\texttt{else}\,\texttt{C}_2 : (M_1, R_1) \sqcup (M_2, R_2)}$$

(Sequence) $$\frac{\vdash \texttt{C}_1 : M_1, R_1 \quad \texttt{C}_2 : M_2, R_2}{\vdash \texttt{C}_1; \texttt{C}_2 : (M_1, R_1) \cdot (M_2, R_2)}$$

(Loop) $$\frac{\vdash \texttt{C} : M, R}{\vdash \texttt{loop}\,\texttt{X}_\ell\{\texttt{C}\} : (M^*, R^*)}$$

where (M^*, R^*) is the loop closure of (M, R) with respect to $\texttt{X}_\ell$.

The DFR for a command can always be computed in time polynomial in the size of the command (i.e., the size of its abstract syntax tree). This is done bottom up, so (since the calculus is deterministic) every node is treated once. The work per node is the application of one of the above rules, each of which is polynomial-time.

4 Certifying Linear Bounds

We present an adaption of the method to certify linear bounds on variables. Essentially the only change is that when something is deduced to be non-linear, it is labeled by a 3. Thus, 2's only describe linear bounds. This is summarized here:

value	3	2	1^+	1	0
dependency type	nonlinear	linear	additive	copy	none

For example: the command $\texttt{X}_1\texttt{:=}\texttt{X}_1\texttt{+2*}\texttt{X}_2$ is described by $\texttt{X}_2 \xrightarrow{1} \texttt{X}_2$, $\texttt{X}_2 \xrightarrow{2} \texttt{X}_1$, $\texttt{X}_1 \xrightarrow{1^+} \texttt{X}_1$.

The calculus $\vdash_{lin}$ for linear bounds deviates from the polynomiality calculus (judgments $\vdash$) only as follows.

1. In abstracting expressions into vectors $V \in \{0, 1, 1^+, 2\}^n$, we treat linear expressions as before, while every occurrence of multiplication $\texttt{e}_1 * \texttt{e}_2$ creates a Type-3 dependence on all variables in $\texttt{e}_1$ and $\texttt{e}_2$. Formally, the abstraction and concretisation functions α, γ are replaced with appropriate α^{lin} and γ^{lin}.
2. For a DFR (M, R), define $LC_\ell^{lin}(M, R) = (M', R')$ where M' is identical to M except that: for all j such that $M_{jj} \in \{1^+, 2\}$, $M'_{\ell j} = 3$; and $R' = R \cap C_2(\mathbf{A^1}(M'))$. LC^{lin} replaces LC in the calculation of the loop closure.

5 Analysing Running Time

Recall that the *iteration count* grows by one each time a loop body is entered. To certify that it is polynomially (or linearly) bounded, it suffices to include an extra variable in the program that sums up the values of loop counters, and certify that this variable is so bounded. We can extend our calculi to implicitly include this variable, so that there is no need to actually modify the program. This is achieved as follows.

1. Matrices become of order $(n+1)\times(n+1)$. Thus the identity matrix I includes the entry $\mathtt{X}_{n+1}\xrightarrow{1}\mathtt{X}_{n+1}$ which reflects preservation of the extra variable.
2. The inference rule for the loop command is modified to reflect an implicit increase of the extra variable. The rule thus becomes:

   (Loop) $$\frac{\vdash \mathtt{C} : M, R}{\vdash \mathtt{loop}\,\mathtt{X}_\ell\{\mathtt{C}\} : M', R'}$$

   where M', R' are obtained from the loop closure (M^*, R^*) by:

   $$\begin{aligned} M' &= M^* \sqcup \{\ell \xrightarrow{1^+} (n+1),\ (n+1) \xrightarrow{1^+} (n+1)\} \\ R' &= R^* \cup \{\{i \to j, \ell \to n+1\} \mid i \to j \in \mathbf{A^1}(M^*)\} \end{aligned}$$

We assume that the reader can see that the implicit $\mathtt{X}_{n+1}$ is treated by these rules just as a variable that actually accumulates the loop counters.

6 Concluding Remarks

We have presented a new method for inferring complexity bounds for imperative programs by a compositional program analysis. The analysis applies to a limited, but non-trivial *core language* and proves that the properties of interest are decidable. We believe that this core-language framework is important for giving a robust yardstick for a project: we have a completeness result, so we can say that we achieved our goal (which, in fact, evolved from the understanding that previous methods did not achieve completeness for such a language).

This work is related, on one hand, to the very rich field of program analysis and abstract interpretation. These are typically targeted at realistic, Turing complete languages, and integrating our ideas with methods from that field is an interesting direction for further research, whether one considers expanding our approach to a richer core language, or creating clever front-ends for realistic languages.

Another connection is with Implicit Computational Complexity. In this field, a common theme is to capture complexity classes. Our core language cannot be really used for computation, but one can define a more complete language that has a simple, complexity-preserving translation to the core language. For example, let us provide the language with an input medium in the form of a binary string, and with operations to read it, as well as reading its length; and then with a richer arithmetic vocabulary, including appropriate conditionals. We obtain a "concrete" programming language $L_{concrete}$ that has an evident abstracting translation $\mathcal{T}$ into the core language, and it is not hard to obtain results such as: $f : \mathbb{N} \to \mathbb{N}$ is PTIME-computable if and only if it can be computed by an L program p such that $\mathcal{T}p$ is polynomial-time (and hence recognized so by our method). We leave the details to the full paper. Note that no procedure for inferring complexity will be complete for $L_{concrete}$ itself, precisely because it is powerful enough to simulate Turing machines.

Finally, let us point out some directions for further research:

- Extending the core language. For instance, our language does not include constants as do the languages considered in [KN04, NW06]. Neither does it include data types of practical significance such as strings (these too are present in the latter works).
- Investigating the design of front ends for realistic languages.
- Extending the set of properties that can be decided beyond the current selection of linear and polynomial growth rates and running times.

Acknowledgement. We are grateful to the CiE reviewers for some very detailed and helpful reviews.

References

[Cou96] Cousot, P.: Abstract interpretation. ACM Computing Surveys 28(2), 324–328 (1996)

[JK05] Jones, N.D., Kristiansen, L.: The flow of data and the complexity of algorithms. In: Cooper, S.B., Löwe, B., Torenvliet, L. (eds.) CiE 2005. LNCS, vol. 3526, pp. 263–274. Springer, Heidelberg (2005)

[JK08] Jones, N.D., Kristiansen, L.: A flow calculus of mwp-bounds for complexity analysis. ACM Trans. Computational Logic (to appear)

[KN04] Kristiansen, L., Niggl, K.-H.: On the computational complexity of imperative programming languages. Theor. Comput. Sci. 318(1-2), 139–161 (2004)

[NW06] Niggl, K.-H., Wunderlich, H.: Certifying polynomial time and linear/polynomial space for imperative programs. SIAM J. Comput 35(5), 1122–1147 (2006)