mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity

Clément Aubert
School of Computer and Cyber Sciences, Augusta University, GA, USA

Thomas Rubiano
LIPN – UMR 7030 Université Sorbonne Paris Nord, France

Neea Rusch
School of Computer and Cyber Sciences, Augusta University, GA, USA

Thomas Seiller
LIPN – UMR 7030 Université Sorbonne Paris Nord, France
CNRS, Paris, France

Abstract

Implicit Computational Complexity (ICC) drives better understanding of complexity classes, but it also guides the development of resources-aware languages and static source code analyzers. Among the methods developed, the *mwp-flow analysis* [23] certifies polynomial bounds on the size of the values manipulated by an imperative program. This result is obtained by bounding the transitions between states instead of focusing on states in isolation, as most static analyzers do, and is not concerned with termination or tight bounds on values. Those differences, along with its built-in compositionality, make the mwp-flow analysis a good target for determining how ICC-inspired techniques diverge compared with more traditional static analysis methods. This paper's contributions are three-fold: we fine-tune the internal machinery of the original analysis to make it tractable in practice; we extend the analysis to function calls and leverage its machinery to compute the result of the analysis efficiently; and we implement the resulting analysis as a lightweight tool to automatically perform data-size analysis of `C` programs. This documented effort prepares and enables the development of certified complexity analysis, by transforming a costly analysis into a tractable program, that furthermore decorrelates the problem of deciding if a bound exist with the problem of computing it.

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Theory of computation → Complexity theory and logic; Theory of computation → Logic and verification

Keywords and phrases Static Program Analysis, Implicit Computational Complexity, Automatic Complexity Analysis, Program Verification

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.26

Related Version *Technical report*: `https://hal.archives-ouvertes.fr/hal-03596285` [4]

Supplementary Material *Software (Source Code)*: `https://github.com/statycc/pymwp`
archived at `swh:1:dir:22a4ab0cfad49138981ed25fc2abfe830fb7ccdf`
Software (Documentation and Demo): `https://statycc.github.io/pymwp`

Funding This research is supported by the Th. Jefferson Fund of the Embassy of France in the United States and the FACE Foundation, and has benefited from the research meeting 21453 "Static Analyses of Program Flows: Types and Certificate for Complexity" in Schloss Dagstuhl. Th. Rubiano and Th. Seiller are supported by the Île-de-France region through the DIM RFSI project "CoHOp".

Acknowledgements The authors wish to express their gratitude to Assya Sellak for her contribution to this work, to the reviewers of previous versions for their comments, and to the FSCD community: in particular, the reviews we received were extremely interesting, and generated new directions and questions, for which we are thankful.

1 Introduction: letting ICC drive the development of static analyzers

Certifying program resource usages is possibly as crucial as the specification of program correctness, since a guaranteed correct program whose memory usage exceeds available resources is, in fact, unreliable. The field of Implicit Computational Complexity (ICC) theory [15] pioneers in "embedding" in the program itself a guarantee of its resource usage, using e.g., bounded recursion [8, 27] or type systems [6, 26]. This field initiated numerous distinct and original approaches, primarily to characterize complexity classes in a machine-independent way, with increasing expressivity, but these approaches have rarely materialized into concrete programming languages or program analyzers: even if, as opposed to traditional complexity, its models are generally expressive enough to write down actual algorithms [30, p. 11], they rarely escape the sphere of academia or extend beyond toy languages, with a few exceptions [5, 22]. However, by abstracting away constant factors and insignificant orders of magnitude, it is frequently conjectured that ICC will allow sidestepping some of the difficult issues one usually has to face when inferring the resource usage of a concrete program.

This work reinforces this conjecture by adjusting, improving and implementing an existing ICC technique, the *mwp-bounds analysis* [23], which certifies that the values computed by an imperative program will be bounded by polynomials in the program's input. This flow analysis is elegant but computationally costly, and it missed an opportunity to leverage its built-in compositionality: we address both issues by revisiting and expanding the original flow calculus, and further make our point by implementing it on a subset of the `C` programming language. While the theory has been improved to allow analysis of function definitions and calls – including recursive ones, a feature not widely supported [21, p. 359] – , its integration into the implementation is underway, as we placed primary focus on developing an efficient and implementable technique for program analysis. Implementing a tool along the theory enabled testing improvements in real-life, which in return drove adjustments to the theory.

Our enhanced technique answers positively two questions asked by the authors of the original analysis [23, Section 1.2], namely

1. Can the method be extended to richer languages?
2. Can it lead to powerful and convenient tools?

It also supports the conjecture that ICC can be used to construct concrete tools, but highlights that doing so requires adjusting the theory to make it tractable in practice. This work also provides better insight into the original analysis, by e.g., separating the algorithm to decide the existence of a bound from its evaluation into a concrete bound; and by illustrating its plasticity: while our analysis conservatively extends the original one, it nevertheless greatly alters its internal machinery to ease its implementability. Last but not least, our technique is orthogonal to most static analysis methods, which focus on worst-case resource-usage complexity or termination, while ours establishes that the growth rate of variables values is at most polynomially related to their inputs.

Our paper starts by recalling the "original" *mwp-bounds analysis* [23] – to which we refer for a more gentle introduction – and discuss its limitations (Sect. 2). In Sect. 3, we motivate, introduce and justify two modifications to this original analysis, and state that this calculus can be reduced to the original one. We then extend this analysis along two axis (Sect. 4): we detail how functions calls can be analyzed, and how the structures we implemented allowed to speed up some very costly operations. Finally, Sect. 5 presents and discuss our implementation, and Sect. 6 concludes. The proofs, some additional details on semi-rings and the detail of our benchmarks are in appendix, with the exception of some tedious proofs relative to semi-rings that are only in our technical report [4].

2 Background: the original flow analysis

The original analysis [23] computes a polynomial bound – if it exists – on the sizes (of the value itself) of variables in an imperative `while` programming language, extended with a `loop` operator, by computing for each variable a vector that tracks how it depends on other variables – and the program itself gets assigned a matrix collecting those vectors. While this does not ensure termination, it provides a certificate guaranteeing that the program uses throughout its execution at most a polynomial amount of space, and as a consequence that *if* it terminates, it will do so in polynomial time.

2.1 Language analyzed: fragments of imperative language

▶ **Definition 1** (Imperative Language). *Letting natural number variables range over* `X` *and* `Y` *and boolean expressions over* `b`*, we define* expressions `e` *and* commands `C` *as follows:*

```
e ::=X ‖ X - Y ‖ X + Y ‖ X * Y
C ::=X = e ‖ if b then C else C ‖ while b do {C} ‖ loop X {C} ‖ C ; C
```

where `loop X {C}` *means "do* `C` `X` *times" and* `C;C` *is used for sequentiality ("do* `C`*, then* `C`*"). We write "program" for a series of commands composed sequentially.*

This language assumes that the program's inputs are the only variables, and that assigning a value to a variable inside the program is not permitted. Extending flow calculi to those operations has been discussed [23, p. 3] and proven possible [9], but we leave this for future work – in particular, our `C` examples will be of `foo` functions with their variables listed as parameters[1]. However, we disallow w.l.o.g. composed expressions of the form `X + Y * Y`, which can always be dealt with in the style of three-address code.

2.2 A flow calculus of mwp-bounds for complexity analysis

Flows characterize controls from one variable to another, and can be, in increasing growth rate, of type 0 – the absence of any dependency – *m*aximum, *w*eak polynomial and *p*olynomial. The bounds on programs written in the syntax of Sect. 2.1 are represented and calculated thanks to vectors and matrices whose coefficients are elements of the mwp semi-ring.

▶ **Definition 2** (The mwp semi-ring and matrices over it). *Letting* $\textsc{mwp} = \{0, m, w, p\}$ *with* $0 < m < w < p$*, and* α*,* β*,* γ *range over* $\textsc{mwp}$*, the* mwp semi-ring $(\textsc{mwp}, 0, m, +, \times)$ *is defined with* $+ = \max$*,* $\alpha \times \beta = \max(\alpha, \beta)$ *if* $\alpha, \beta \neq 0$*, and* 0 *otherwise.*

We denote $\mathbb{M}(\textsc{mwp})$ *the matrices over* $\textsc{mwp}$*, and, fixing* $n \in \mathbb{N}$*,* M *for* $n \times n$ *matrices over* $\textsc{mwp}$*,* M_{ij} *for the coefficient in the* i*th row and* j*th column of* M*,* $\oplus$ *for the componentwise addition, and* $\otimes$ *for the product of matrices defined in a standard way. The* 0*-element for addition is* $0_{ij} = 0$ *for all* i, j*, and the* 1*-element for product is* $1_{ii} = m$*,* $1_{ij} = 0$ *if* $i \neq j$*, and the resulting structure* $(\mathbb{M}(\textsc{mwp}), 0, 1, \otimes, \oplus)$ *is a semi-ring that we simply write* $\mathbb{M}(\textsc{mwp})$*. The closure operator* $\cdot^*$ *is* $M^* = 1 \oplus M \oplus (M^2) \oplus \ldots$*, for* $M^0 = 1$*,* $M^{m+1} = M \otimes M^m$*.*

[1] Our implementation allows to relax this condition, as exemplified in `inline_variable.c`, without losing any of the results expressed in this paper. Assuming a fixed number of variables, known ahead of time, is mostly a theoretical artifact used to simplify the analysis.

$$\frac{}{\vdash_{\text{JK}} \texttt{Xi} : \{{}^{m}_{\texttt{i}}\}}\ \text{E1} \qquad\qquad \frac{}{\vdash_{\text{JK}} \texttt{e} : \{{}^{w}_{\texttt{i}} \mid \texttt{Xi} \in \text{var}(\texttt{e})\}}\ \text{E2}$$

$$\star \in \{+,-\}\ \frac{\vdash_{\text{JK}} \texttt{Xi} : V_1 \quad \vdash_{\text{JK}} \texttt{Xj} : V_2}{\vdash_{\text{JK}} \texttt{Xi}\star\texttt{Xj} : pV_1 \oplus V_2}\ \text{E3} \qquad \star \in \{+,-\}\ \frac{\vdash_{\text{JK}} \texttt{Xi} : V_1 \quad \vdash_{\text{JK}} \texttt{Xj} : V_2}{\vdash_{\text{JK}} \texttt{Xi}\star\texttt{Xj} : V_1 \oplus pV_2}\ \text{E4}$$

(a) Rules for assigning vectors to expressions.

$$\frac{\vdash_{\text{JK}} \texttt{e} : V}{\vdash_{\text{JK}} \texttt{Xj = e} : 1 \overset{\texttt{j}}{\leftarrow} V}\ \text{A} \qquad \frac{\vdash_{\text{JK}} \texttt{C1} : M_1 \quad \vdash_{\text{JK}} \texttt{C2} : M_2}{\vdash_{\text{JK}} \texttt{C1; C2} : M_1 \otimes M_2}\ \text{C}$$

$$\frac{\vdash_{\text{JK}} \texttt{C1} : M_1 \quad \vdash_{\text{JK}} \texttt{C2} : M_2}{\vdash_{\text{JK}} \texttt{if b then C1 else C2} : M_1 \oplus M_2}\ \text{I}$$

$$\forall i, M^*_{ii} = m\ \frac{\vdash_{\text{JK}} \texttt{C} : M}{\vdash_{\text{JK}} \texttt{loop Xl \{C\}} : M^* \oplus \{{}^{p}_{\texttt{l}}\!\rightarrow j \mid \exists i, M^*_{ij} = p\}}\ \text{L}$$

$$\forall i, M^*_{ii} = m \text{ and } \forall i,j, M^*_{ij} \neq p\ \frac{\vdash_{\text{JK}} \texttt{C} : M}{\vdash_{\text{JK}} \texttt{while b do \{C\}} : M^*}\ \text{W}$$

(b) Rules for assigning matrices to commands.

Figure 1 Original non-deterministic ("Jones-Kristiansen") flow analysis rules.

Although not crucial to understand our development, details about (strong) semi-rings and the mwp semi-ring, and the construction of a semi-ring whose elements are matrices with coefficients in a semi-ring – so, in particular, $\mathbb{M}(\textsc{mwp})$ – are given in our technical report [4, A.1 and A.2] and sketched in appendix Appendix A.

Below, we let V_1, V_2 be column vectors with values in MWP, αV_1 be the usual scalar product, and $V_1 \oplus V_2$ be defined componentwise. We write $\{{}^{\alpha}_{i}\}$ for the vector with 0 everywhere except for α in its ith row, and $\{{}^{\alpha}_{i}, {}^{\beta}_{j}\}$ for $\{{}^{\alpha}_{i}\} \oplus \{{}^{\beta}_{j}\}$.

Replacing in a matrix M the jth column vector by V is denoted $M \overset{j}{\leftarrow} V$. The matrix M with $M_{ij} = \alpha$ and 0 everywhere else is written $\{{}^{\alpha}_{i}\!\rightarrow j\}$, and the set of variables in the expression `e` is written var(`e`). The assumption is made that exactly n different variables are manipulated throughout the analyzed program, so that n-vectors are assigned to expressions – in a non-deterministic way, to capture larger classes of programs [23, Section 8] – and $n \times n$ matrices are assigned to commands using the rules presented Fig. 1 [23, Section 5].

The intuition is that if $\vdash_{\text{JK}}$ `C` : M can be derived, then all the values computed by `C` will grow at most polynomially w.r.t. its inputs [23, Theorem 5.3], e.g., will be bounded by $\max(\vec{x}, p_1(\vec{y})) + p_2(\vec{z})$, where p_1 and p_2 are polynomials and $\vec{x}$ (resp. $\vec{y}$, $\vec{z}$) are m-(resp. w-, p-)annotated variables in the vector for the considered output. Since the derivation system is non-deterministic, multiple matrices and polynomial bounds – that sometimes coincide – may be assigned to the same program. Furthermore, the coefficient at $M_{\texttt{ij}}$ carries quantitative information about the way `Xi` depends on `Xj`, knowing that 0- and m-flows are harmless and without constraints, but that w- and p- flows are more harmful w.r.t. polynomial bounds and need to be handled with care, particularly in loops – hence the condition on the L and W rules. The derivation may fail – some programs may not be assigned a matrix – if at least one of the variables used in the body of a loop depends "too strongly" upon another, making it impossible to ensure polynomial bounds on the loop itself. We will use the following example as a common basis to discuss possible failure, non-determinism, and our improvements.

▶ **Example 3.** Consider `loop X3 {X2 = X1 + X2}`. The body of the `loop` command admits three different derivations, obtained by applying A to one of the three derivation of the expression `X1 + X2`, that we name π_0, π_1 and π_2:

$$\dfrac{\dfrac{}{\vdash_{\text{JK}} \texttt{X1} : \begin{pmatrix} m \\ 0 \\ 0 \end{pmatrix}}\,\text{E1} \qquad \dfrac{}{\vdash_{\text{JK}} \texttt{X2} : \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix}}\,\text{E1}}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} p \\ m \\ 0 \end{pmatrix}}\,\text{E3} \qquad \dfrac{\dfrac{}{\vdash_{\text{JK}} \texttt{X1} : \begin{pmatrix} m \\ 0 \\ 0 \end{pmatrix}}\,\text{E1} \qquad \dfrac{}{\vdash_{\text{JK}} \texttt{X2} : \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix}}\,\text{E1}}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}}\,\text{E4} \qquad \dfrac{}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}}\,\text{E2}$$

From π_0, the derivation of `loop X3 {X2 = X1 + X2}` can be completed using A and L, but since L requires having only m coefficients on the diagonal, π_1 cannot be used to complete the derivation, because of the p coefficient in a box below:

$$\dfrac{\dfrac{\begin{matrix} \vdots\, \pi_0 \\ \vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} p \\ m \\ 0 \end{pmatrix} \end{matrix}}{\vdash_{\text{JK}} \texttt{X2 = X1 + X2} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix}}\,\text{A}}{\vdash_{\text{JK}} \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & p & m \end{pmatrix}}\,\text{L} \qquad \dfrac{\begin{matrix} \vdots\, \pi_1 \\ \vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix} \end{matrix}}{\vdash_{\text{JK}} \texttt{X2 = X1 + X2} : \begin{pmatrix} m & m & 0 \\ 0 & \boxed{p} & 0 \\ 0 & 0 & m \end{pmatrix}}\,\text{A}$$

Similarly, using A after π_2 gives a w coefficient on the diagonal and makes it impossible to use L, hence only one derivation for this program exists.

2.3 Limitations and inefficiencies of the mwp analysis

Even if the proof techniques are far from trivial, with only 9 rules and skipping over boolean expressions (observe that the condition `b` has no impact in the rules I or W), the analysis is flexible and easy to carry out – at least mathematically. It also has inherent limitations: while the technique is sound, it is not complete and programs such as greatest common divisor fail to be assigned a matrix. We will discuss in Sect. 5.2, the benefits and originality of this analysis, but we would now like to stress how it is computationally inefficient, since the non-determinacy makes the analysis costly to carry out and can lead to memory explosions.

Abstracting Example 3, one can see that the base case of non-determinism – e.g., to assign a vector to `X1` $\star$ `X2`–yields vectors $\begin{pmatrix} p \\ m \end{pmatrix}$ (using E1 then E3), $\begin{pmatrix} m \\ p \end{pmatrix}$ (using E1 then E4) and $\begin{pmatrix} w \\ w \end{pmatrix}$ (using E2). Since none of those vectors is less than the others, only two strategies are available to analyze a larger program containing `X1` $\star$ `X2`: either the derivations for this base case are considered one after the other, or they are all stored in memory at the same time. Considering the derivations for the base case one after the other can lead to a time explosion, as a program of n lines can have 3^n different derivations – as exemplified by `explosion.c`, a simple series of applications – and it is possible that only one of them can be completed, so all must be explored. On the other hand, storing those three vectors and constructing all the matrices in parallel leads to a memory explosion: the analysis for two commands involving 6 variables, with 3 choices – which cannot be simplified as explained previously – would result in 9 matrices of size 6×6, i.e., 324 coefficients. All in all, a program of n lines with x different variables can require c_1^n different derivations, which can produce up to $(c_2 \times x)^2$ coefficients to store for some constants c_1, c_2.

Beyond inefficiency, there are additional limitations: while the analysis is naturally compositional, this feature is not leveraged in the original system; furthermore, an occurrence of non-polynomial flows in the matrix causes the analysis to simply stop, thus not capturing failure in a meaningful way. We will discuss our solutions to these deficiencies next.

$$\star \in \{+,-\}\ \frac{}{\vdash \texttt{Xi}\star\texttt{Xj} : (0 \mapsto \{{}^{m}_{\texttt{i}}, {}^{p}_{\texttt{j}}\}) \oplus (1 \mapsto \{{}^{p}_{\texttt{i}}, {}^{m}_{\texttt{j}}\}) \oplus (2 \mapsto \{{}^{w}_{\texttt{i}}, {}^{w}_{\texttt{j}}\})}\ \mathrm{E}^{\mathrm{A}}$$

$$\frac{}{\vdash \texttt{Xi * Xj} : \{{}^{w}_{\texttt{i}}, {}^{w}_{\texttt{j}}\}}\ \mathrm{E}^{\mathrm{M}} \qquad \frac{}{\vdash \texttt{Xi} : \{{}^{m}_{\texttt{i}}\}}\ \mathrm{E}^{\mathrm{S}}$$

(a) Rules for assigning vectors to expressions.

$$\frac{\vdash \texttt{e} : V}{\vdash \texttt{Xj = e} : 1 \xleftarrow{\texttt{j}} V}\ \mathrm{A} \qquad \frac{\vdash \texttt{C1} : M_1 \quad \vdash \texttt{C2} : M_2}{\vdash \texttt{C1; C2} : M_1 \otimes M_2}\ \mathrm{C} \qquad \frac{\vdash \texttt{C1} : M_1 \quad \vdash \texttt{C2} : M_2}{\vdash \texttt{if b then C1 else C2} : M_1 \oplus M_2}\ \mathrm{I}$$

$$\frac{\vdash \texttt{C} : M}{\vdash \texttt{loop Xl \{C\}} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{p}_{\texttt{l}} \to j \mid \exists i, M^*_{ij} = p\}}\ \mathrm{L}^{\infty}$$

$$\frac{\vdash \texttt{C} : M}{\vdash \texttt{while b do \{C\}} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{\infty}_{i} \to j \mid M^*_{ij} = p\}}\ \mathrm{W}^{\infty}$$

(b) Rules for assigning matrices to commands.

Figure 2 Deterministic improved flow analysis rules.

3 A deterministic, always-terminating, declension of the mwp analysis

The problem of finding a derivation in the original calculus is in NP [23, Theorem 8.1]. But since all the non-determinism is in the rules to assigning a vector, the potentially exponential number of derivations are actually extremely similar. Hence, instead of having the analysis stop when failing to establish a derivation and re-starting from scratch, storing the different vectors and constructing the derivation while keeping all the options open seems to be a better strategy, but, as we have seen, this causes a memory blow-up. We address it by fine-tuning the internal machinery: to represent non-determinism, we let the matrices take as values either functions from choices to coefficients in MWP or coefficients in MWP, so that instead of mapping choices to derivations, all the derivations are represented by the same matrix that internalizes the different choices. Sect. 3.1 discusses this improvement, which results in a notable gain: getting back to the example of Sect. 2.3, a program involving 6 variables, with 3 choices, would now be assigned a (unique) 6×6 matrix that requires 66 coefficients instead of the 324 we previously had – this is because 30 coefficients are "simple" values in MWP, and 6 are functions from a set of choices $\{0, 1, 2\}$ to values in MWP, each represented with 6 coefficients.

For the choices that give coefficients fulfilling the side condition of L or W, the derivation can proceed as usual, but when a particular choice gives a coefficient that violates it, we decided against simply removing it. Instead, to guarantee that all derivations always terminate, we mark that choice by indicating that it would not provide a polynomial bound. This requires extending the MWP semi-ring with a special value ∞ that represents failure in a local way, marking non-polynomial flows, and is detailed in Sect. 3.2. As a by-product, this enables fine-grained information on programs that *do not* have polynomially bounded growth, since the precise dependencies that break this growth rate can be localized.

Taken together (Sect. 3.3), our improvements ensure that exactly one matrix will always be assigned to a program while carrying over the correctness of the original analysis. We give in Fig. 2 the deterministic system we are introducing in full, but will gently introduce it though the remaining parts of this section: note that the rules A, C and I are unchanged, up to the fact that the matrices, sum and product are in a different semi-ring.

3.1 Internalizing non-determinism: the choice data flow semi-rings

Internalizing the choice requires altering the semi-ring used in the analysis: we want to replace the three vectors over $\textsc{mwp}$ that can be assigned to an expression by a single vector over $\{0,1,2\} \to \textsc{mwp}$ that captures the same three choices. For a program needing to decide p times between the 3 available choices, this means replacing the $3 \times p$ different matrices in $\mathbb{M}(\textsc{mwp})$ by a single matrix in $\mathbb{M}(\{0,1,2\}^p \to \textsc{mwp})$. For any strong semi-ring $\mathbb{S}$ and family of sets $(A_i)_{i=1,\dots,p}$, both $A_i \to \mathbb{S}$ and $\mathbb{M}(\prod_{i=1}^p A_i \to \mathbb{S})$ are semi-rings, using the usual cartesian product of sets, and there exists an isomorphism $\mathbb{M}(\prod_{i=1}^p A_i \to \mathbb{S}) \cong \prod_{i=1}^p A_i \to \mathbb{M}(\mathbb{S})$ [4, A.3]. This dual nature of the semi-ring considered is useful:

- the analysis will now assign an element M of $\mathbb{M}(\prod_{i=1}^p A_i \to \textsc{mwp})$ to a program;
- representing M as an element of $\prod_{i=1}^p A_i \to \mathbb{M}(\textsc{mwp})$ allows one to use an *assignment* $\vec{a} = (a_1, \dots, a_p) \in \prod_{i=1}^p A_i$ to produce a matrix $M[\vec{a}] \in \mathbb{M}(\textsc{mwp})$, recovering the mwp-flow that would have been computed by making the choices $a_1, \dots, a_p$ in the derivation.

▶ Remark 4. As the unique degree of non-determinism to assign a matrix to commands is 3, our modification of the analysis flow consists simply of recording the different choices by letting $A_i = \{0,1,2\}$ for all $i = 1, \dots, p$ where p is the number of times a choice had to be taken. Starting with Sect. 4, function calls will require potentially different sets A_i.

▶ Notation 5. In the following and in the implementation alike, we will denote a function $(a_1^0 \times \cdots \times a_p^0 \mapsto \alpha_0) + \cdots + (a_1^k \times \cdots \times a_p^k \mapsto \alpha_k)$ in $A^p \to \textsc{mwp}$ with $\mathrm{Card}(A) = k$ by, omitting the product, $(\alpha_0 \delta(a_1^0, 0) \cdots \delta(a_p^0, p)) + \cdots + (\alpha_k \delta(a_1^k, 0) \cdots \delta(a_p^k, p))$, with $\delta(i,j) = m$ if the jth choice is i, 0 otherwise. Example 8 will justify and explain this choice.

Our derivation system replaces the E3 and E4 rules with a single rule E^{A} ("additive"), and splits E2 in two exclusive rules, E^{M} for "multiplicative" and E^{S} for "simple" (atomic) expressions – Theorem 11 will prove how they are equivalent.

▶ **Example 6.** We represent the vectors $\left(\begin{smallmatrix} p \\ m \\ 0 \end{smallmatrix}\right)$, $\left(\begin{smallmatrix} m \\ p \\ 0 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} w \\ w \\ 0 \end{smallmatrix}\right)$ from Example 3 with a single vector $\left(\begin{smallmatrix} p\delta(0,0)+m\delta(1,0)+w\delta(2,0) \\ m\delta(0,0)+p\delta(1,0)+w\delta(2,0) \\ 0 \end{smallmatrix}\right)$, that can be read as $\left(\begin{smallmatrix} \{0\mapsto p, 1\mapsto m, 2\mapsto w\} \\ \{0\mapsto m, 1\mapsto p, 2\mapsto w\} \\ 0 \end{smallmatrix}\right)$, where we write 0 for $\{0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0\}$[2]. Since in particular[3], $\mathbb{M}(\{0,1,2\} \to \textsc{mwp}) \cong \{0,1,2\} \to \mathbb{M}(\textsc{mwp})$, the obtained vector can be rewritten as $0 \mapsto \left(\begin{smallmatrix} p \\ m \\ 0 \end{smallmatrix}\right), 1 \mapsto \left(\begin{smallmatrix} m \\ p \\ 0 \end{smallmatrix}\right), 2 \mapsto \left(\begin{smallmatrix} w \\ w \\ 0 \end{smallmatrix}\right)$.

3.2 Internalizing failure: de-correlating derivations and bounds

The original analysis stops when detecting a non-polynomial flow, puts an end to the chosen strategy (i.e., set of choices) and restarts from scratch with another one. We adapt the rules so that every derivation can be completed even in the presence of non-polynomial flows, thanks to a new top element, ∞, representing failure in a local way.

Ignoring our previous modification in this subsection, the semi-ring $\textsc{mwp}^\infty$ we need to consider is $(\textsc{mwp} \cup \{\infty\}, 0, m, +^\infty, \times^\infty)$, with $\infty > \alpha$ for all $\alpha \in \textsc{mwp}$, $+^\infty = \max$ as before, and $\alpha \times^\infty \beta = 0$ if $\alpha, \beta \neq \infty$ and α or β is 0, $\max(\alpha, \beta)$ otherwise. This different condition in the definition of $\times^\infty$ ensures that once non-polynomial flows have been detected, they cannot be erased (as $\infty \times^\infty 0 = \infty$).

[2] The implementation supports both coefficients from $\textsc{mwp}$ *and* coefficients from $\{0,1,2\}^p \to \textsc{mwp}$, cf. e.g., a simple assignment example `assign_expression.c`.

[3] This is a variant of Lemma 21 [4, A.3]. While the latter lemma applies to algebras of square matrices, a similar result holds for rectangular matrices of a fixed size; the algebraic structure is no longer that of a semi-ring as rectangular matrices do not possess a proper multiplication, but the proof can be adapted to show the existence of an isomorphism of modules between the considered spaces.

The only cases where the original analysis may fail is if the side conditions of L or W (Fig. 1) are not met. We replace those by L^∞ and W^∞ (Fig. 2), which replace the problematic coefficients with ∞, marking non-polynomial dependencies, and carry on the analysis.

▶ **Example 7.** The program from Example 3 would now receive three derivations (omitting the one obtained from π_0, as the resulting matrix is identical):

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi_1}{\vdash \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}}}{\vdash \texttt{X2 = X1 + X2} : \begin{pmatrix} m & m & 0 \\ 0 & p & 0 \\ 0 & 0 & m \end{pmatrix}}\;\mathrm{A}}{\vdash \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & p & 0 \\ 0 & \infty & 0 \\ 0 & p & m \end{pmatrix}}\;\mathrm{L}^\infty \qquad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{X1 + X2} : \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}}\;\mathrm{E2}}{\vdash \texttt{X2 = X1 + X2} : \begin{pmatrix} m & w & 0 \\ 0 & w & 0 \\ 0 & 0 & m \end{pmatrix}}\;\mathrm{A}}{\vdash \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & w & 0 \\ 0 & \infty & 0 \\ 0 & 0 & m \end{pmatrix}}\;\mathrm{L}^\infty$$

Of course, neither of those two derivations would yield polynomial bound – since they contain ∞ coefficients – but it becomes possible to determine that the last one is "better" – since $\begin{pmatrix} p \\ \infty \\ p \end{pmatrix} > \begin{pmatrix} w \\ \infty \\ 0 \end{pmatrix}$ – and to observe how their "failure" would propagate in larger programs, possibly establishing that one fares better than the other in terms of non-polynomial growths. This could imply, for instance, that particular programs without polynomial bounds could still be considered "reasonnable" if they are exponential only in some variables that are known to have smaller values in input.

3.3 Merging the improvements: illustrations and proofs

We prove that our system captures the original system in the sense that set aside ∞ coefficients, both systems agree (Theorem 11), but also that exactly one matrix is produced per program (Theorem 10) – i.e., that we can analyze as many programs as originally, and still be correct regarding the bounds. Before doing so, we would like to give more specifics on our system, by combining the semi-rings and intuitions from the previous two subsections. We have discussed our "axiomatic" (E^{A}, E^{M}, E^{S}) and "loop" rules (L^∞ and W^∞), but remain to discuss the rules for assignment (A), `if` (I) and composition (C) – which is where both improvements meet. Mathematically speaking, adopting the semi-ring defined over matrices with coefficients in $\{0, 1, 2\}^p \to \textsc{mwp} \cup \{\infty\}$ is straightforward, and we simply write $\oplus$ and $\otimes$ the operations resulting from merging the two transformations. We discuss in Sect. 4.3 how, however, those operations are computationally costly and how we address this challenge.

▶ **Example 8.** Using our deterministic system presented in Fig. 2, consider the following:

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{X1 + X2} : V}\;\mathrm{E}^{\mathrm{A}}}{\vdash \texttt{X1 = X1 + X2} : 1 \xleftarrow{1} V}\;\mathrm{A} \qquad \dfrac{\dfrac{}{\vdash \texttt{X1 - X3} : V'}\;\mathrm{E}^{\mathrm{A}}}{\vdash \texttt{X1 = X1 - X3} : 1 \xleftarrow{1} V'}\;\mathrm{A}}{\vdash \texttt{if b then \{X1 = X1 + X2\} else \{X1 = X1 - X3\}} : (1 \xleftarrow{1} V) \oplus (1 \xleftarrow{1} V')}\;\mathrm{I}$$

with

$$\begin{aligned}
V &= 0 \mapsto \{{}^{m}_{1}, {}^{p}_{2}\} \oplus 1 \mapsto \{{}^{p}_{1}, {}^{m}_{2}\} \oplus 2 \mapsto \{{}^{w}_{1}, {}^{w}_{2}\} \\
V' &= 0 \mapsto \{{}^{m}_{1}, {}^{p}_{3}\} \oplus 1 \mapsto \{{}^{p}_{1}, {}^{m}_{3}\} \oplus 2 \mapsto \{{}^{w}_{1}, {}^{w}_{3}\} \\
1 \xleftarrow{1} V &\cong \begin{pmatrix} (0{\mapsto}m) \oplus (1 \mapsto p) \oplus (2{\mapsto}w) & 0 & 0 \\ (0{\mapsto}p) \oplus (1 \mapsto m) \oplus (2{\mapsto}w) & m & 0 \\ 0 & 0 & m \end{pmatrix} = \begin{pmatrix} m\delta(0{,}0) \oplus p\delta(1{,}0) \oplus w\delta(2{,}0) & 0 & 0 \\ p\delta(0{,}0) \oplus m\delta(1{,}0) \oplus w\delta(2{,}0) & m & 0 \\ 0 & 0 & m \end{pmatrix} \\
1 \xleftarrow{1} V' &\cong \begin{pmatrix} (0{\mapsto}m) \oplus (1 \mapsto p) \oplus (2{\mapsto}w) & 0 & 0 \\ 0 & m & 0 \\ (0{\mapsto}p) \oplus (1 \mapsto m) \oplus (2{\mapsto}w) & 0 & m \end{pmatrix} = \begin{pmatrix} m\delta(0{,}1) \oplus p\delta(1{,}1) \oplus w\delta(2{,}1) & 0 & 0 \\ 0 & m & 0 \\ p\delta(0{,}1) \oplus m\delta(1{,}1) \oplus w\delta(2{,}1) & 0 & m \end{pmatrix}
\end{aligned}$$

Some care is needed to perform the addition for the I rule: the choices in the left and right branches are independent, so we must use coefficients in $\{0,1,2\}^2 \to \textsc{mwp}$ for the 2^3 choices. While the mapping notation would require to use positions to describe which choice is being refereed to, the δ notation makes it immediate, as it encodes in the second value of δ that two choices are considered, numbering the choice in the left branch 0. Hence we can sum the coefficients and obtain the matrix that can be observed in our implementation by analyzing `example7.c`.

▶ **Example 9.** Our deterministic system now assigns to `loop X3 {X2 = X1 + X2}` from Example 3 the unique matrix

$$\begin{pmatrix} m & (0{\mapsto}p)\oplus(1 \mapsto m)\oplus(2{\mapsto}w) & 0 \\ 0 & (0{\mapsto}m)\oplus(1 \mapsto\infty)\oplus(2{\mapsto}\infty) & 0 \\ 0 & (0{\mapsto}p)\oplus(1 \mapsto 0)\oplus(2{\mapsto}0) & m \end{pmatrix} = \begin{pmatrix} m & p\delta(0{,}0)\oplus m\delta(1{,}0) \oplus w\delta(2{,}0) & 0 \\ 0 & m\delta(0{,}0)\oplus\infty\delta(1{,}0)\oplus\infty\delta(2{,}0) & 0 \\ 0 & p\delta(0{,}0)\oplus 0\delta(1{,}0)\oplus 0\delta(2{,}0) & m \end{pmatrix}$$

where we observe that

1. only one choice, one assignment, 0, gives a matrix without ∞ coefficient, corresponding to the fact that, in the original system, only π_0 could be used to complete the proof,
2. the choice impacts the matrix locally, the coefficients being mostly the same, independently from the choice,
3. the influence of `X2` on itself is where possible non-polynomial growth rates lies, as the ∞ coefficient are in the second column, second row.

We are now in possession of all the material and intuitions needed to state the correspondence between our system and the original one of Jones and Kristiansen.

▶ **Theorem 10** (Determinancy and termination). *Given a program P, there exists unique $p \in \mathbb{N}$ and $M \in \mathbb{M}(\{0,1,2\}^p \to \textsc{mwp}^\infty)$ such that $\vdash P : M$.*

Proof. The existence of the matrix is guaranteed by the completeness of the rules, as any program written in the syntax presented in Sect. 2.1 can be typed with the rules of Fig. 2. The uniqueness of the matrix is given by the fact that no two rules can be applied to the same command. Details are provided in Appendix B. ◀

▶ **Theorem 11** (Adequacy). *If $\vdash P : M$, then for all $\vec{a} \in A^p$, $\vdash_{\text{JK}} P : M[\vec{a}]$ iff $\infty \notin M[\vec{a}]$.*

Proof. The proof uses that P cannot be assigned a matrix in the original calculus iff the deterministic calculus introduce a ∞ coefficient, and from the fact that both calculus coincide in all the other cases. Details are provided in Appendix B. ◀

▶ **Corollary 12** (Soundness). *If $\vdash P : M$ and there exists $\vec{a} \in A^p$ such that $\infty \notin M[\vec{a}]$, then every value computed by P is bounded by a polynomial in the inputs.*

Proof. This is an immediate corollary of the original soundness theorem [23, Theorem 5.3] and of Theorem 11. ◀

This proves that the two analyses coincide, when excluding ∞, and that we can re-use the original proofs. However, our alternative definition should be understood as an important improvement, as it enables a better proof-search strategy while optimizing the memory usage, and hence enables the implementation (Sect. 5). It also lets the programmer gain more fine-grained feedback, and illustrates the flexibility of the analysis: the latter will also be demonstrated by the improvements we discuss in the next section.

4 Extending and improving the analysis: functions and efficiency

To improve this analysis, one could try to extract a tight bound, to certify it, or to port it to a compiler's intermediate representation. Adding constant values is arguably immediate [23, p. 3] but handling pointers, even if technically possible, would probably require significant work. This illustrates at the same time the flexibility of the analysis, and the distance separating ICC-inspired techniques from their usage on actual programs. We decided to narrow this gap along two axes: the first one consists of allowing function definitions and calls in our syntax. It is arguably a small improvement, but illustrates nicely the compositionality of the analysis, and includes recursively defined functions. The second extension intersects the theory and the implementation: it details how our semi-ring structure can be leveraged to maintain a tractable algorithm to compute costly operations on our matrices, and to separate the problem of deciding if a bound exists from computing its form.

4.1 Leveraging compositionality to analyze function calls

Thanks to its compositionality, this analysis can easily integrate functions and procedures, by re-using the matrix and choices of a program implementing the function called. We begin by adding to the syntax the possibility of defining multiple functions and calling them:

▶ **Definition 13** (Functions). *Letting `R` (resp. `f`) range over variables (resp. function names), we add* function calls[4] *to the commands (Def. 1) and allow* function declarations*:*

`C` $:=$ `Xi = f(X1, ..., Xn)`   `F` $:=$ `f(X1, ..., Xn){C; return R}`

In a function declaration, `f(X1, ..., Xn)` is called the header*, and the* body *is simply `C` (i.e., `return R` is not part of the body). A* program *is now a series of function declarations such that all the function calls refer to previously declared functions – we deal with recursive calls in Sect. 4.2 – and a* chunk *is a series of commands.*

Now, given a function declaration computing f, we can obtain the matrix M_f by analyzing the body of f as previously done. It is then possible to store the assignments $\vec{a}_0, \dots, \vec{a}_k$, for which no ∞ coefficients appear[5], and to project the resulting matrices to only keep the vector at `R` that provides quantitative information about all the possible dependencies of the output variable `R` w.r.t. input values, possibly merging choices leading to the same result. After this, we are left with a family $(M_f[\vec{a}_0])|_{\mathtt{R}}, \dots, (M_f[\vec{a}_k])|_{\mathtt{R}}$ of vectors – as the syntax here is restricted to functions with a single output value, even if accommodating multiple return values would be dealt with the same way – that we can re-use when calling the function.

The analysis of the command calling f is then dealt with the F rule below:

$$\frac{}{\vdash \mathtt{Xi = F(X1,\dots, Xn)} : 1 \overset{\mathtt{i}}{\leftarrow} (((M_f[\vec{a}_0])|_{\mathtt{R}})\delta(0,c) \oplus \cdots \oplus ((M_f[\vec{a}_k])|_{\mathtt{R}})\delta(k,c))}\ \text{F}$$

This rule introduces a choice c over k possible matrices, and it is possible that $k \neq 3$, but this is not an issue, since our semi-ring construction can accommodate any set of choice A.

[4] Function calls that discard the output – procedures – could also be dealt with easily, but are vacuous in our effect-free, in particular pointer-free, language

[5] Allowing ∞ coefficients would not change the method described nor its results, but it does not seem relevant to allow calling functions that are not polynomially bounded.

▶ **Example 14.** Consider the following two programs `Q` and `P`:

```
    int f(X1, X2){                    int foo(X1, X2){
Q =   while b do {X2=X1+X1};      P =   X2=X1+X1;
      return X2;                        X1=f(X2, X2);
    }                                 }
```

We first have $\vdash$ `X2 = X1 + X1` $: V$ for $V = \left(\begin{smallmatrix} m & p\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) \\ 0 & 0 \end{smallmatrix}\right)$, and since $V^* = \left(\begin{smallmatrix} m & p\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) \\ 0 & m \end{smallmatrix}\right)$, applying W^∞ gives $\vdash$ `Q` $: \left(\begin{smallmatrix} m & \infty\delta(0,0)\oplus\infty\delta(1,0)\oplus w\delta(2,0) \\ 0 & m \end{smallmatrix}\right)$. Noting that only one choice gives an ∞-free matrix, we can now carry on the analysis of `P`:

$$\dfrac{\dfrac{\vdots}{\vdash \texttt{X2 = X1 + X1} : V} \qquad \dfrac{}{\vdash \texttt{X1 = f(X2, X2)} : 1 \overset{1}{\leftarrow} ((\begin{smallmatrix} w \\ m \end{smallmatrix})\,\delta(0,c))}\ \mathrm{F}}{\vdash \texttt{P} : V \otimes 1 \overset{1}{\leftarrow} ((\begin{smallmatrix} w \\ m \end{smallmatrix})\,\delta(0,c))}\ \mathrm{C}$$

In this particular case, the c choice can be discarded, since only one option is available.

Now, to prove that the F rule faithfully extends the analysis (Theorem 17), i.e., preserves Corollary 12, we prove that the analysis of the program "inlining" the function call – as defined below – is, up to some bureaucratic variable manipulation and ignoring some ∞ coefficients, the same as the analysis resulting from using our rule. Intuitively, this mechanism provides the expected result because the choices in the function *do not* affect the program calling it, and because their sets of variables are disjoint – except for the return variable.

▶ **Definition 15** (In-lining function calls). *Let P be a chunk containing a call to the function f, and F be the function declaration computing the function f. The* context *$P[\cdot]$, a chunk containing a slot $[\cdot]$, is obtained by replacing in P the function call `Xi=f(X1, ..., Xn)`, with `X'1=X1; ...; X'n=Xn;` $[\cdot]$ `Xi=R`, for `R`, `X'1, ..., X'n` fresh variables added to the header containing the chunk.*

The chunk $\tilde{F}$ is obtained from the body of F by renaming the input variables to `X'1, ...`, `X'n`, and the variable returned by F to `R`. The code $P[F]$ is finally obtained by computing the chunk $\tilde{F}$, and inserting it in place of the symbol $[\cdot]$ in $P[\cdot]$.

That P and $P[F]$ have, at the end of their executions, the same values stored in the variables of P is straightforward in our imperative programming language.

▶ **Example 16.** The in-lining of `Q` in `P` from Example 14 would give the following chunk $\tilde{\texttt{Q}}$ and context `P`$[\cdot]$, `P[Q]` being obtained by replacing in the latter $[\cdot]$ with the former:

```
                                              int foo(X1, X2, X'1, R){
                                                X2=X1+X1;
Q̃ = while b do {R=X'1+X'1};       P[·] =        X'1=X2;
                                                [·]
                                                X1=R;
                                              }
```

The analysis of `P` (excluding the function call) and `Q` is implemented at `example15a.c`, and of `P[Q]` at `example15b.c`: this latter diverges with Example 14 only up to projection and ∞-coefficients that are removed by F but not when in-lining the function call.

Now, we need to prove that the matrices $M(P)$ – obtained by analyzing P and using the F rule for `Xi=f(X1, ..., Xn);` – and $M(P[F])$ – obtained by analyzing the inlined $P[F]$ – are the same. However, to avoid conflict with the variables and to project the matrices on the relevant values, some bureaucracy is needed: we write $\Pi_P(M(P[F]))$ (resp. $(1 - \Pi_P)(M(P[F]))$) the projection of $M(P[F])$ onto the variables in (resp. *not* in) P. Some non-deterministic choices may appear within the (modified) chunk $\tilde{F}$ inside $P[F]$, i.e.,

- the coefficients of $M(P)$ are elements of the semi-ring $\prod_{i=1}^{p+1} A_i \to \mathbb{M}(\textsc{mwp})$, with one particular choice corresponding to the F rule – we write the corresponding index i_0;
- the coefficients of $M(P[F])$ are elements of the semi-ring $\prod_{i=1}^{p+k} B_i \to \mathbb{M}(\textsc{mwp})$, where k choices are made within the chunk $\tilde{F}$ – we write the corresponding indexes $j_1, j_2, \dots, j_k$ (note these are in fact consecutive indexes).

We note $\pi : \{1, \dots, p+k\} \to \{1, \dots, p+1\}$ the projection of the choices in $P[F]$ onto the corresponding choices in P, i.e., $\pi(j) = \begin{cases} j & \text{if } j < j_1 \\ i_0 & \text{if } j_1 \leqslant j < j_k \\ j-k+1 & \text{if } j_k < j \end{cases}$. We note that each matrix used as axiom in the function call corresponds to a specific assignment on indexes $j_1, \dots, j_k$. We write $\Psi : A_{i_0} \to \prod_{i=j_1}^{j_k} B_i$ the corresponding injection, extended to $\bar{\Psi} : \prod_{i=1}^{p+1} A_i \to \prod_{i=0}^{p+k} B_i$ straightforwardly.

▶ **Theorem 17.** *For all $\vec{a}$ in $\prod_{i=1}^{p+1} A_i$, $(M(P))[\vec{a}] = (1 - \Pi_P)(M(P[F]))[\bar{\Psi}(\vec{a})]$, and for all β in $\prod_{i=0}^{p+k} B_i$ not in the image of $\bar{\Psi}$, $(1 - \Pi_P)(M(P[F])[\beta])$ contains ∞.*

Proof. It is sufficient to prove it for the simplest chunk P containing only one command `Xi = f(X1, ..., Xn)`. This comes from the compositional nature of the analysis, as a sequence of commands is assigned the product of the matrices of each individual command. Then, checking the theorem in this case is a straightforward, though tedious (due to keeping track of all indices), computation. ◀

4.2 Integrating recursive calls, the easy way

The question of dealing with self-referential, or recursive, calls, naturally arises when extending to function calls. It turns out that our approach makes such cases easy to handle.

A program implementing a function `rec` calling itself cannot use the F rule presented above as is, since the result of the analysis of `rec` is precisely what we are trying to establish. However, if `rec` takes two input variables `X1` and `X2` and its return value is assigned to a third variable `X3`, then we already know that the vector at 3 will need to be replaced by the vector capturing the dependency between `X1`, `X2`, and the return variable of `rec` (which we will take to be `X3` in our example). The solution consists in replacing the actual values in this vector by variables α, β ranging over values in $\textsc{mwp}^\infty$, terminating the analysis with those variables, and then to resolve the equation – which is easy given the small size of the $\textsc{mwp}^\infty$ semiring.

As an example[6], consider the following program and compute the corresponding matrix:

```
int rec(X1, X2){
  X1 = X1 + X2;
  X3 = rec(X1, X2);
  return X3;
}
```

$$\begin{pmatrix} m\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) & 0 & 0 \\ p\delta(0,0)\oplus m\delta(1,0)\oplus w\delta(2,0) & m & 0 \\ 0 & 0 & m \end{pmatrix} \otimes 1 \xleftarrow{3} \begin{pmatrix} \alpha \\ \beta \\ 0 \end{pmatrix}$$
$$= \begin{pmatrix} m\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) & 0 & \alpha m\delta(0,0)\oplus\alpha p\delta(1,0)\oplus\alpha w\delta(2,0) \\ p\delta(0,0)\oplus m\delta(1,0)\oplus w\delta(2,0) & m & \alpha p\delta(0,0)\oplus\alpha m\delta(1,0)\oplus\alpha w\delta(2,0)\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$$

Using the assignments 0, 1 and 2 gives $\begin{pmatrix} m & 0 & \alpha m \\ p & m & \alpha p\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$, $\begin{pmatrix} p & 0 & \alpha p \\ m & m & \alpha m\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$ and $\begin{pmatrix} w & 0 & \alpha w \\ w & m & \alpha w\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$, and since the third vector should be equal to $\begin{pmatrix} \alpha \\ \beta \\ 0 \end{pmatrix}$, this gives three systems of equations:

[6] Where we use variables that are not parameters, following footnote 1, and where our recursive call does not terminate: we are focusing on growth rates and not on termination, and keep the example compact.

$$\begin{cases} \alpha m &= \alpha \\ \alpha p \oplus \beta &= \beta \end{cases} \qquad \begin{cases} \alpha p &= \alpha \\ \alpha m \oplus \beta &= \beta \end{cases} \qquad \begin{cases} \alpha w &= \alpha \\ \alpha w \oplus \beta &= \beta \end{cases}$$

The smaller solution to the first (resp. second, third) equational system is $\{\alpha = m; \beta = p\}$ (resp. $\{\alpha = p; \beta = p\}$, $\{\alpha = w; \beta = w\}$), and as a consequence, we find two meaningful solutions (all others being larger than those): $\begin{pmatrix} m \\ p \\ 0 \end{pmatrix}$ and $\begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$.

4.3 Taking advantage of polynomial structure to compute efficiently

Ensuring that the analysis is tractable is an important part of our contribution. For a program accepting n different derivations and having k different derivations that cannot be completed, the original flow calculus must run at most $k+1$ times to find *one* derivation, while our analysis outputs the $k+n$ different derivations in one run, and then sorts them – as discussed next – by listing all the evaluations and looking for ∞ values. In this task, the C rule, that lets building programs from commands, is obviously crucial and consists simply in multiplying two matrices: however, since we are internalizing the choices, those matrices contain a mixture of functions from choices to coefficients in $\textsc{mwp}^\infty$ and of coefficients in $\textsc{mwp}$. Multiplying such matrices is more costly, but also essential: an 8-line program such as `explosion.c` requires to multiply elements of its matrix 34,992 times[7]. This forces to represent and manipulate the elements of $\prod_{i=1}^{p} A_i \to \mathbb{M}(\textsc{mwp})$ – setting aside ∞ coefficients for a moment – cleverly: simple comparison showed that the improved algorithm presented below made the analysis roughly *five times* faster (Sect. C.3).

As discussed in Notation 5, elements of this semi-ring are represented as *polynomials* w.r.t. the generating set given by the functions $\delta(i,j) : \prod_{i=1}^{p} A_i \to \textsc{mwp}$ defined by $\delta(i,j)(a_1,\dots,a_p) = m$ if $a_j = i$ and $\delta(i,j)(a_1,\dots,a_p) = 0$ otherwise, i.e., an element of $\prod_{i=1}^{p} A_i \to \textsc{mwp}$ is represented as a polynomial $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ with $\alpha_i \in \textsc{mwp}$.

This basis has an important property: the *monomials* $\alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ in a polynomial can be ordered so that the product with another monomial is ordered, i.e., if $\alpha \leqslant \beta$ and both $\alpha \times \gamma$ and $\beta \times \gamma$ are non-zero, then $\alpha \times \gamma \leqslant \beta \times \gamma$. This order is leveraged to obtain efficient algorithms, similar to what is done using Gröbner bases for computation of standard polynomials [35]. For instance, the algorithm for multiplication of polynomials uses this property to compute the product of an ordered polynomial P with $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$:

1. compute the products $P_i = P \times \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ for all i;
2. compare and order a list L of all the first elements of those polynomials;
3. append the smallest element to the result and remove it from the corresponding P_i;
4. insert the (new) first element of P_i to the list L if it exists;
5. if L is non-empty, go back to step 3.

When adding or multiplying polynomials, which consist of monomials, we check if a monomial is contained or included by another, and exclude all redundant cases (cf. `contains` or `includes`). This is also done when inserting monomials. Thus we keep polynomials free of implementation choices that we would otherwise have to handle during evaluation.

[7] The need to optimize functions is made even more obvious when we discuss benchmarking in Sect. 5.1.

4.4 Deciding the existence of a bound faster thanks to delta graphs

Adopting the $\prod_{i=1}^{p} A_i \to \text{MWP}^{\infty}$ semi-ring permits to complete all derivations simultaneously, but remains to determine if there exists an assignment $\vec{a} \in \prod_{i=1}^{p} A_i$ s.t. the resulting matrix is ∞-free, to decide whenever a program accepts a polynomial bound: this is the *evaluation* step. Despite the optimizations detailed above that simplifies the task, this phase remains particularly costly, since the number of assignment grows exponentially w.r.t. the number of choice, which is linear in the number of variables. While this step is necessary (in one form or another) if one wishes to produce the actual mwp matrices certifying polynomial bounds, we implemented a specific data structure to keep track of assignments resulting in ∞ coefficients on the fly, thus allowing the analysis to provide a qualitative answer quickly. This section details how those *delta graphs* allow to immediately determines whenever a polynomial bound exists without having to compute the corresponding matrix, something that was not possible in the original, non-deterministic, calculus.

A delta graph is a graph whose vertices are monomials. The graph is populated during the analysis by adding those monomials that appear with an infinite coefficient – i.e., possible choices leading to ∞ in the resulting matrix. This graph is structured in layers: each layer corresponds to the size of the monomials (the number of deltas) it contains. The intuition is that a monomial – or rather a list of deltas $\delta(\_,\_)$ – defines a subset of the space $\prod_{i=1}^{p} A_i$; the less deltas in the monomial, the greater the subspace represented[8]. As we populate the delta graph, we create edges within a given layer to keep track of differences between monomials: we add an edge labeled i between two monomials if and only if they differ only on one delta $\delta(\_, i)$ (i.e., one is obtained from the other by replacing the first index of $\delta(\_, i)$). This is used to implement a `fusion` method on delta graphs, which simplifies the structure: as soon as a monomial m in layer n has $\text{Card}(A_i) - 1$ outgoing edges labelled i, we can remove all these monomials and insert a shorter monomial in layer $n - 1$, obtained from m by simply removing $\delta(\_, i)$. This implements the fact that $\sum_{k=0}^{\text{Card}(A_i)-1} m\delta(k, j) = m$.

Now, remember the delta graph represents the subspace of assignments for which an ∞ appears. If at some point the delta graph is completely simplified (i.e., "fusions" to the graph with a unique monomial consisting in an empty list of $\delta(\_,\_)$), it means the whole space of assignments is represented and no mwp-bounds can be found. On the contrary, if the analysis ends with a delta graph different from the completely simplified one, at least one assignment exists for which no infinite coefficients appear, and therefore at least one mwp-bound exists. This allows one to answer the question "Is there at least one mwp-bound?" *without actually computing said bounds.* Based on the information collected in the delta graph and the matrix with polynomial coefficients, one can however recover all possible matrix assignments by going through all possible valuations.

This last part is implemented with a specific iterator that leverages the information collected in the delta graph to skip large sets of valuations in a single step. For instance, suppose the monomial $\delta(1, 1)$ lies in the delta graph – i.e., that an infinite coefficient will be reached if the second index is equal to 1. When asked the valuation after $(0, 0, 2, 2)$ (and supposing that $\text{Card}(A_i) = 3$ for all i), our `delta_iterator` will jump directly to $(0, 2, 0, 0)$, skipping all intermediate valuation of the form $(0, 1, a, b)$ in a single step. Similarly, it will jump from $(1, 0, 2, 2)$ to $(1, 2, 0, 0)$, again skipping several valuations at a time, providing a

[8] Our intuitions here come from the standard topological structure of spaces of infinite sequences, where such a monomial represents a "cylinder set", i.e., an element of the standard basis for open sets.

faster analysis. Note that the implementation required care, to correctly jump when given additional informations from the delta graph, e.g., to produce $(2,0,1,0)$ as the successor of $(0,0,2,2)$ if $\delta(0,0)$, $\delta(1,1)$ and $\delta(0,2)$ all belong to the delta graph.

5 Implementing, testing and comparing the analysis

Demonstrating the implementability of the improved and extended mwp-bounds analysis requires an implementation. Our open-source solution, packaged through Python Package Index (PyPI) as `pymwp`, is a standalone command line tool, written in `Python`, that automatically performs growth-rate analysis on programs written in a subset of the `C` programming language. For programs that pass the analysis, it produces a matrix corresponding to the input program and a list of valid derivation choices; and for programs that do not have polynomial bounds, it reports infinity. Our motivation for choosing `C` as the language of analysis resulted from its central role and similarity with the original `while` language. `Python` was an ideal choice for the implementation because of its plasticity, collection of libraries, and because it allowed partial reuse of a previous flow analysis tool [3, 31, 32]. The source code is available on Github, along with an online demo, and detailed documentation [33] describing its current supported features and functionality. We now discuss how we tested and assesed it, and how it compares (or, rather *does not* compare) to other similar approaches.

5.1 Experimental evaluation

We allocated extensive focus and effort on testing and profiling our implementation, to ensure the correctness and efficiency of the analysis, and with the terminal objective of obtaining a usable tool. The test suite includes 42 `C` programs, carefully designed to exercise different aspects of the analysis, ranging from basic derivations, to ones producing worst-case behavior (by yielding e.g., dense matrices or exponential number of derivations), and classical examples such as computing the greatest common divisor or exponentiation.

We refer to our benchmarks (presented in Appendix C) for measured analysis results for each program. The most salient aspect is that our analysis is extremely fast (the time is measured in *milli*seconds) despite important numbers of function calls (in the 10k range, excluding builtin Python language calls, for 10-lines programs). Even examples tailored to stress our implementation cannot make the analysis go over *4 seconds*. We cannot compare our implementation with implementations of the original analysis, since it has never been implemented, and (according to our attempts) cannot be implemented in any realistic manner.

5.2 Related tools and incompatible metrics

This work was inspired by the series of works of the flow analysis from the “Copenhagen school” [11, 24]. The overall flow analysis approach is related in spirit to abstract interpretation [13, 14]; that bounds *transitions* between states (e.g., commands) instead of states [24]. This approach shaped the implementation of tools detecting loop quasi-invariants [31, 32].

Other communities share a similar goal of inferring resource-usage. Complexity analyzers such as SPEED [19] for `C++`, COSTA [1] for `Java` bytecode, ComplexityParser [21] for `Java`, Resource Aware ML for `OCaml` [29] or Cerco [2] and Verasco [25] for `C` generate (certified) cost or runtime analysis on (subsets of) imperative programming languages. Embracing such a large diversity is difficult, but our technique is different from existing implementations and tools: most of them focus on worst-case resource-usage complexity or termination, while we are interested in upper-bounds on the final values of program variables, i.e., we focus on *growth* instead of actual values. This makes the comparison with our approach difficult, but highlights at the same time its uniqueness in today's landscape of static analyzers.

Further, our approach provides other desirable properties:

1. it is compositional, which allows one to "hot-plug" bounds of previously analyzed functions without additional work,
2. it is modular, as the internal machinery can be altered – as in this paper – without having to re-develop the theory,
3. it is language-independent, as it reasons abstractly on imperative languages, but can be applied to real programs, as our implementation illustrates, and should extend to more complex languages,
4. it is lightweight and programmer-friendly, as it is fast, does not require annotations or to record value ranges,
5. it studies growth independently from e.g., iteration bounds, thus sidestepping difficult cases that worst-case analysis has to tackle, and
6. it may enable tight bounds on programs, as it has been done recently [10] for a similar analysis [11].

In particular compositionality is a highly desirable property – because otherwise the analysis needs to be re-run on programs or API whenever embedded into different pieces of software – yet difficult to achieve by most other approaches, as discussed and partially remedied recently [12]. While we suppose one approach could be used to derive the result obtained by the other, we do believe the originality of our pioneering ICC-based approach may inspire new and original directions in static program analysis.

6 Conclusion: limitations, strengths and future work

This work attempts to illustrate the usefulness and applicability of ICC results, but also the need to refine and adapt them. We showed that the mwp-flow analysis as originally described cannot scale to programs in a real programming language: while the considered analysis is definitely powerful and elegant, its mathematical nature let some costly operations go unchecked. However we have shown that, extended and coupled to optimizations techniques, its result enable the development of a novel and original static analysis technique on imperative programs, focused on *growth* rather than on termination or worst-case bounds.

This work is a proof of concept and it has limitations, both theoretical and practical: the theory is missing memory uses, pointers, and arrays and the supported feature set of the implementation could be extended. But instead of focusing on what this analysis *cannot* perform, we would like to stress that all the tools are in place to perform similar analysis on intermediate representations of code in compilers, which will naturally simplify the task of fitting richer program syntax to our analysis, and brings this technique yet another step closer to practical use cases.

One of our next steps include certifying the analysis using the Coq proof assistant [34], and implementing the analysis in certified tools such as the Compcert compiler [28] (or, more precisely, its static single assignment version [7]) or certified-llvm [36]. The plasticity of both compilers and of the implemented analysis should facilitate porting our results and approaches to support further programming languages in addition to `C`. As complexity analysis is notably difficult in Coq [20], we believe a push in this direction would be welcome, and that ICC provides all the needed tools for it.

Another direction is to explore the possibility for our analysis to focus on the *final* values of variables instead of tracking them throughout the whole program. Indeed, recall from Sect. 3.2 that our semi-ring is such that $\infty \times^{\infty} 0 = \infty$, but another valid choice would have been to pick $\infty \times^{\infty} 0 = 0$ [4, A.4]. In this case, it seems that if some non-polynomial growth is caused by a variable that is then "thrown away" (overridden), then a program could still pass the analysis: whether this lead gives relevant results is yet to determine, but it would be another nice illustration of the plasticity of this analysis.

Last but not least, working on finer comparison with other static analyzers [18] could be useful. We have stressed in Sect. 5.2 how such comparison was uneasy, as the finality of our tool is not directly comparable with any other analyzer we know of. However, some tools such as AProVE [17] or CoFloCo [16] provide polynomial upper bounds for `C` programs, and we could assess e.g., on the Termination Problems Data Base whether `pymwp` can analyze as many problems and how often all three analyzers agree. The motivation for the original mwp-analysis was to develop resource analysis for distinguishing feasible problems and to work at the boundary of undecidability, but this could actually be one of `pymwp`'s strength as a *pre-processor* to other static analyzers, to save them from running costly analysis on programs known to be unfeasible. This fast analysis and the compositionality of our tool could also, on longer term, be useful to construct IDE plug-ins that provide low-latency feedback to the programmer.

References

1 Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: design and implementation of a cost and termination analyzer for java bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *LNCS*, pages 113–132. Springer, 2007. `doi:10.1007/978-3-540-92188-2_5`.

2 Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified complexity (cerco). In Ugo Dal Lago and Ricardo Peña, editors, *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*, volume 8552 of *LNCS*, pages 1–18. Springer, 2013. `doi:10.1007/978-3-319-12466-7_1`.

3 Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. Lqicm on c toy parser. URL: `https://github.com/statycc/LQICM_On_C_Toy_Parser`.

4 Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. mwp-analysis improvement and implementation: Realizing implicit computational complexity. Preliminary technical report, March 2022. URL: `https://hal.archives-ouvertes.fr/hal-03596285`.

5 Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP):43:1–43:29, 2017. `doi:10.1145/3110287`.

6 Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, pages 266–275. IEEE Computer Society, 2004. `doi:10.1109/LICS.2004.1319621`.

7 Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, 2014. `doi:10.1145/2579080`.

8 Stephen J. Bellantoni and Stephen Arthur Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *STOC*, pages 283–93. ACM, 1992. `doi:10.1145/129712.129740`.

9 Amir M. Ben-Amram. On decidable growth-rate properties of imperative programs. In Patrick Baillot, editor, *Proceedings International Workshop on Developments in Implicit Computational complExity, DICE 2010, Paphos, Cyprus, 27-28th March 2010*, volume 23 of *EPTCS*, pages 1–14, 2010. doi:10.4204/EPTCS.23.1.

10 Amir M. Ben-Amram and Geoff W. Hamilton. Tight polynomial worst-case bounds for loop programs. *Log. Meth. Comput. Sci.*, 16(2), 2020. doi:10.23638/LMCS-16(2:4)2020.

11 Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In Arnold Beckmann and Costas Dimitracopoulos andBenedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability inEurope, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *LNCS*, pages 67–76. Springer, 2008. doi:10.1007/978-3-540-69407-6_7.

12 Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 467–478. ACM, 2015. doi:10.1145/2737924.2737955.

13 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

14 Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts: Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, St. Andrews, NB, Canada, August 1-5, 1977*, pages 237–278. North-Holland, 1977.

15 Ugo Dal Lago. A short introduction to implicit computational complexity. In Nick Bezhanishvili and Valentin Goranko, editors, *ESSLLI*, volume 7388 of *LNCS*, pages 89–109. Springer, 2011. doi:10.1007/978-3-642-31485-8_3.

16 Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *LNCS*, pages 254–273, 2016. doi:10.1007/978-3-319-48989-6_16.

17 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Jera Fuhs, Carstenand Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, and René Swiderski, Stephanie andThiemann. Analyzing program termination and complexity automatically with aprove. *J. Autom. Reasoning*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.

18 Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. The termination and complexity competition. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *LNCS*, pages 156–166. Springer, 2019. doi:10.1007/978-3-030-17502-3_10.

19 Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 127–139, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1480881.1480898.

20 Armaël Guéneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes)*. PhD thesis, Inria, Paris, France, 2019. URL: https://tel.archives-ouvertes.fr/tel-02437532.

21 Emmanuel Hainry, Emmanuel Jeandel, Romain Péchoux, and Olivier Zeyen. Complexityparser: An automatic tool for certifying poly-time complexity of Java programs. In Antonio Cerone and

Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *LNCS*, pages 357–365. Springer, 2021. `doi:10.1007/978-3-030-85315-0_20`.

22 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*, pages 781–786. Springer, 2012. `doi:10.1007/978-3-642-31424-7_64`.

23 Neil D. Jones and Lars Kristiansen. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4):28:1–28:41, 2009. `doi:10.1145/1555746.1555752`.

24 Neil D. Jones and Flemming Nielson. Abstract interpretation: A semantics-based tool for program analysis. In Samson Abramsky, Dov M. Gabbay, and Thomas Stephen Edward Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 527–636. Oxford University Press, 1995.

25 Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium onPrinciples of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015. `doi:10.1145/2676726.2676966`.

26 Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1):163–180, 2004. `doi:10.1016/j.tcs.2003.10.018`.

27 Daniel Leivant. Stratified functional programs and computational complexity. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 325–333. ACM Press, 1993. `doi:10.1145/158511.158659`.

28 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. `doi:10.1145/1538788.1538814`.

29 Benjamin Lichtman and Jan Hoffmann. Arrays and references in resource aware ML. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 26:1–26:20. Schloss Dagstuhl, 2017. `doi:10.4230/LIPIcs.FSCD.2017.26`.

30 Jean-Yves Moyen. *Implicit Complexity in Theory and Practice.* Habilitation thesis, University of Copenhagen, 2017. URL: `https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf`.

31 Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk detection. In Deepak D'Souza and K. Narayan Kumar, editors, *ATVA*, volume 10482 of *LNCS*. Springer, 2017. `doi:10.1007/978-3-319-68167-2_7`.

32 Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk motion by peeling with statement composition. In Guillaume Bonfante and Georg Moser, editors, *Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2017, Uppsala, Sweden, April 22-23, 2017*, volume 248 of *EPTCS*, pages 47–59, 2017. `doi:10.4204/EPTCS.248.9`.

33 pymwp's documentation, 2021. URL: `https://statycc.github.io/pymwp/`.

34 Coq Team. Coq documentation, 2022. URL: `https://coq.github.io/doc/`.

35 Joris van der Hoeven and Robin Larrieu. Fast Gröbner basis computation and polynomial reduction for generic bivariate ideals. *Applicable Algebra in Engineering, Communication and Computing*, 30(6):509–539, December 2019. `doi:10.1007/s00200-019-00389-9`.

36 Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 175–186. ACM, 2013. `doi:10.1145/2491956.2462164`.

A Technical appendix on semi-rings (abridged)

This is an abridged version of the technical development on semi-ring that is exposed in full details in our technical report [4, A.1 and A.2].

▶ **Lemma 18** (mwp semi-ring). *The tuple* $(\{0, m, w, p\}, 0, m, +, \times)$, *with*

- $0 < m < w < p$,
- $\alpha + \beta = \begin{cases} \alpha & \text{if } \alpha \geqslant \beta \\ \beta & \text{otherwise} \end{cases}$
- $\alpha \times \beta = \begin{cases} \alpha + \beta & \text{if } \alpha \neq 0 \text{ and } \beta \neq 0 \\ 0 & \text{otherwise} \end{cases}$

is a strong *semi-ring.*

▶ **Lemma 19** (Matrix semi-ring). *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$, *the tuple* $\mathbb{M} = (M, 0, 1, \oplus, \otimes)$, *with*

- M *the set of all* $n \times n$ *matrices over* S, *for all* $n \in \mathbb{N}$,
- 0 *defined by* $M = 0$ *iff* $M_{ij} = 0$ *for all* i *and* j,
- 1 *defined by* $M = 1$ *iff* $M_{ij} = 1$ *for* $i = j$, $M_{ij} = 0$ *otherwise,*
- $\oplus$ *defined by* $C = A \oplus B$ *iff* $C_{ij} = A_{ij} + B_{ij}$,
- $\otimes$ *defined by* $C = A \otimes B$ *iff* $C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$,

is a strong semi-ring.

For simplicity, we will write $\mathbb{M}$ as $\mathbb{M}(\mathbb{S}) = (M(S), 0, 1, \oplus, \otimes)$.

▶ **Lemma 20** (Choices semi-ring). *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$ *and a set* A, *the tuple* $\mathbb{F} = (F, 0, 1, \boxplus, \boxtimes)$, *with*

- F *the set of functions from* A *to* S,
- 0 *the constant function* $0(a) = 0$ *for all* $a \in A$,
- 1 *the constant function* $1(a) = 1$ *for all* $a \in A$,
- $\boxplus$ *defined componentwise:* $(f \boxplus g)(a) = (f(a)) + (g(a))$, *for all* f, g *in* F *and* $a \in A$,
- $\boxtimes$ *defined componentwise:* $(f \boxtimes g)(a) = (f(a)) \times (g(a))$, *for all* f, g *in* F *and* $a \in A$,

is a strong semi-ring.

For simplicity, we will write $\mathbb{F}$ as $A \to \mathbb{S} = (A \to S, 0, 1, +, \times)$.

▶ **Lemma 21.** *For all set* A *and strong semi-ring* $\mathbb{S}$, $\mathbb{M}(A \to \mathbb{S}) \cong A \to \mathbb{M}(\mathbb{S})$.

▶ **Lemma 22.** *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$ *and an element* $\perp \notin S$, $\mathbb{S}^{\perp} = (S \cup \{\perp\}, 0, 1, +^{\perp}, \times^{\perp})$ *with, for all* a, $b \in S \cup \{\perp\}$,

$$a +^{\perp} b = \begin{cases} a + b & \text{if } a, b \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

$$a \times^{\perp} b = \begin{cases} a \times b & \text{if } a, b \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

is a semi-ring.

Proof. The proof is immediate, but note that $\mathbb{S}^{\perp}$ is not strong, as $\perp \times 0 = \perp$. ◀

B Omitted Proofs

▶ **Theorem 10** (Determinancy and termination). *Given a program P, there exists unique $p \in \mathbb{N}$ and $M \in \mathbb{M}(\{0,1,2\}^p \to \text{MWP}^\infty)$ such that $\vdash P : M$.*

Proof. The proof proceeds by induction on the length of the program P, expressed in number of commands. We let p be the number of variables in P, but observe that any program P can be treated as manipulating $p' > p$ different variables, by simply adding $p' - p$ additional rows and columns to the matrix, and leaving them unchanged by the derivation of P. While a complete proof would need to constantly account for the number of actual and potential variables used by P, we will simply assume that the reader understands that accounting for this technicality obfuscate more than it clarifies the proof, and we will freely resize the matrices to account for additional variables when needed.

If P is of length 1 Then we know P is of the form `X = e`, and only the rule A can be applied. But then we need to prove that all expression `e` can be typed with exactly one vector. An expression `e` is either a variable `X`, or a composed expression `X * Y`, `X - Y`, or `X + Y`. But then, respectively, only E^S, E^M or E^A (for addition and substraction) can be applied, and this case is proven.

If P is of length $n > 1$ Then we proceed by case on the structure of the command:

- If P is of the form `if b then P1  else P2`, then by induction we know for $i \in \{1,2\}$ there exists p_i and M_i of size $p_i \times p_i$ such that $\vdash$ `Pi` $: M_i$. If $p_1 \neq p_2$, then letting `Mj` being the smaller matrix, it is easy to rewrite `Pj`'s derivation to account for $|p_1 - p_2|$ additional variables, and as $\oplus$ is uniquely defined, we know that $M_1 \oplus M_2$ results in a unique matrix of size $\max(p_1, p_2)$.
- If P is of the form `while b do P'`, this is immediate by induction hypothesis on `P'`, considering that only W^∞ can be applied, and that this rule produces a unique matrix.
- If P is of the form `loop X {P'}`, this case is similar to the previous one, using L^∞ instead of W^∞.
- If P is of the form `P1;P2`, this case is similar to the `if` case, with the possible need to resize one of the matrix obtained by induction, and using that $\otimes$ is uniquely defined. ◀

▶ **Theorem 11** (Adequacy). *If $\vdash P : M$, then for all $\vec{a} \in A^p$, $\vdash_{\text{JK}} P : M[\vec{a}]$ iff $\infty \notin M[\vec{a}]$.*

Proof. The proof proceeds by induction on the length of the program P, expressed in number of commands.

If P is of length 1 Then we know P is of the form `X = e`, and only the rule A can be applied, in both systems. Hence, we need to prove that all expression `e` can be typed the same way in both systems. A careful comparison of Figures 1 and 2 shows that if `e` is of the form `Xi`, then there is a small mismatch. In the original system, we can use either E2, and obtain $\vdash_{\text{JK}}$ `Xi` $: \{{}^{w}_{\texttt{i}}\}$, or E1, and obtain $\vdash_{\text{JK}}$ `Xi` $: \{{}^{m}_{\texttt{i}}\}$, while the only derivation in the deterministic system is using E^S to get $\vdash_{\text{JK}}$ `Xi` $: \{{}^{m}_{\texttt{i}}\}$. As $m < w$, we argue that the deterministic system cannot obtain a derivation that is not useful anyway, and hence that it can be ignored.

As for the other cases, if `e` is a composed expression `X * Y`, `X - Y`, or `X + Y`, it is easy to observe that E^A and E^M encapsulates all the possible combinations of E2 and of E1 followed by E3 or E4 that can be used.

If P is of length $n > 1$ Then the result holds by induction, once we observed that L^∞ and W^∞ are introducing ∞ coefficients *only if* L and W cannot be applied. ◀

C Benchmarks

C.1 Descriptions of program groups

- *Basics* – C programs performing operations corresponding to simple derivation trees.
- *Implementation paper* – example programs presented in this paper.
- *Original paper* – examples taken from or inspired by the original analysis [23].
- *Infinite* – programs whose matrices always contain infinite coefficients.
- *Polynomial* – programs whose matrices do not always contain infinite coefficients.
- *Other* – other C programs of interest.

C.2 Results

The benchmarks are categorized and grouped to distinguish the type of system behavior they exercise. For each program we capture in Table 1

1. program variable count
2. the lines of code in the source program (LOC column)
3. clock time taken by the full analysis (excluding saving result to file, which is otherwise default behavior),
4. number of function calls excluding builtin Python language calls, and
5. the result of the analysis.

Collectively the LOC, time, and function calls columns provide insight into the behavior of the analysis as different aspects of the system are being stress-tested. From the results column we report expected results on each benchmarked program. In the benchmarks table a passing result is represented with ✓and ∞ otherwise. We do not report manually computed bounds as comparison, because the analysis is carried out on individual variables, thus calculating them on multivariate programs is tedious and futile. However, for simple programs such as while_2.c, it is straightforward through visual inspection to verify the obtained 2×2-matrix is indeed the correct result.

These benchmarks were obtained using Python's built-in cProfile utility, extended in `pymwp` implementation to enable batch profiling. The clock times are slight overestimates because the utility adds minor runtime overhead. The number of function calls includes primitive calls, but exclude built-in Python language calls. Full detailed results are viewable in the source code repository: https://github.com/statycc/pymwp/releases/tag/profile-latest

C.3 Comparison

It is not really meaningful or possible to compare those results with any other static analyzer, and impossible to compare it with any other implementation of this type of flow analysis. While we could, in theory, analyze our examples with other static analyzers, their results would be incomparable, as they would produce guarantees on termination or worst case resource usage, which are both orthogonal to our polynomial bounds on value growth. To our knowledge, the only static analyzer using similar metrics [5] was developed only for functional languages, thus preventing comparison. As for implementations of the original analysis, our first attempts showed that a naive implementation would likely fail to handle the memory or time explosions. We did, however, compare the gains resulting from the optimizations described in Sect. 4.3. In a nutshell, our improved algorithm for adding and multiplying polynomials resulted in the analysis being roughly *five times faster* for two programs that we estimate to be representative.

Table 1 Benchmark results produced by `pymwp` on `C` programs.

Program name	Variables	LOC	Time (ms)	Function calls	Bound
Basics					
assign_expression	2	8	133	81614	✓
assign_variable	2	9	115	81238	✓
if	2	9	118	82046	✓
if_else	2	7	118	82928	✓
inline_variable	2	9	118	81979	✓
while_1	2	7	117	82934	✓
while_2	2	7	117	83964	✓
while_if	3	9	122	91572	✓
Implementation paper					
example7	3	10	122	86898	✓
example15_a	2+2	25	122	88763	✓
example15_b	4	16	137	122016	✓
Original paper					
example3_1_a	3	10	110	85286	✓
example3_1_b	3	10	120	87637	✓
example3_1_c	3	11	121	89173	✓
example3_1_d	2	12	116	80002	∞
example3_2	3	12	118	83182	∞
example3_4	5	18	134	108890	∞
example5_1	2	10	116	81185	✓
example7_10	3	10	119	86053	✓
example7_11	4	11	139	119379	✓
Infinite					
exponent_1	4	16	127	99893	∞
exponent_2	4	13	123	92846	∞
infinite_2	2	6	143	128275	∞
infinite_3	3	9	120	89880	∞
infinite_4	5	9	3274	5924420	∞
infinite_5	5	11	369	529231	∞
infinite_6	4	14	1624	2836726	∞
infinite_7	5	15	631	964189	∞
infinite_8	6	23	880	1444782	∞
Polynomial					
notinfinite_2	2	4	119	86174	✓
notinfinite_3	4	9	131	104826	✓
notinfinite_4	5	11	169	168242	✓
notinfinite_5	4	11	174	176179	✓
notinfinite_6	4	16	195	215765	✓
notinfinite_7	5	15	1161	1961806	✓
notinfinite_8	6	22	1893	3172293	✓
Other					
dense	3	16	157	151428	✓
dense_loop	3	17	269	353068	✓
explosion	18	23	1296	2327071	✓
gcd	2	12	114	84914	∞
simplified_dense	2	9	118	85098	✓