An extended and more practical mwp flow analysis

Clément Aubert, Thomas Rubiano, Neea Rusch, Thomas Seiller

An extended and more practical mwp flow analysis*

Clément Aubert[1][0000−0001−6346−3043], Thomas Rubiano[2], Neea Rusch[1], and Thomas Seiller[2,3][0000−0001−6313−0898]

[1] School of Computer and Cyber Sciences, Augusta University
[2] LIPN – UMR 7030 Université Sorbonne Paris Nord
[3] CNRS

Abstract. We improve and refine a method for certifying that the values' sizes computed by an imperative program will be bounded by polynomials in the program's inputs' sizes. Our work "tames" the non-determinism of the original analysis, and offers an innovative way of completing the analysis when a non-polynomial growth is found. We furthermore enrich the analyzed language by adding function definitions and calls, allowing to compose the analysis of different libraries and offering generally more modularity. The implementation of our improved method, discussed in a tool paper [4], also required to reason about the efficiency of some of the needed operations on the matrices produced by the analysis. It is our hope that this work will enable and facilitate static analysis of source code to guarantee its correctness with respect to resource usages.

Keywords: Static Program Analysis · Implicit Computational Complexity · Automatic Complexity Analysis · Program Verification

1 Introduction

This work takes a step further in the implementation of static analysis methods inspired from work in implicit computational theory [11], and more particularly the series of work from the so-called flow analysis of the "Copenhagen school", notably Neil Jones, Lars Kristiansen, and Amir Ben-Amram. The *Copenhagen school approach to implicit computational complexity* initiates in the celebrated *size-change principle* of Ben Amram [8] to characterise termination of programs, and evolved in more precise techniques to capture resource usage and more particularly growth rate between variables' sizes. The overall flow analysis approach is related in spirit to *abstract interpretation* [9,10]; as noted by Jones [15] it bounds *transitions* between states (e.g. commands) instead of states. A first work implemented this technique to develop a static analysis tool detecting loop quasi-invariants [22,23].

* This material is based upon research supported by the Thomas Jefferson Fund of the Embassy of France in the United States and the FACE Foundation.

One landmark result in this series of work is the precise and detailed analysis of the relationship between the resource requirements of a computation and the way data might flow during the computation [14]. Thanks to a typing system resting on matrices with coefficients in the so-called mwp semi-ring, programs in a basic imperative language could be guaranteed to have variables growing at most polynomially with respect to their inputs. While this does not ensure termination, it provides a certificate that *if* the program terminates, it will do so in polynomial time and using at most polynomial space. We here continue in this direction and implement this "mwp-bounds analysis" [14] on a fragment of the `C` programming language [4].

Our contribution is both of practical and theoretical interest: a `python` program analyzing `C` source code is currently being developed [4], and documented at `https://seiller.github.io/pymwp/`. This implementation largely benefited from the improvements discussed in the current paper and triggered the development of a modified analysis allowing for the use of more efficient algorithms to carry out the analysis. However, our theoretical contributions can be read independently from this implementation, and answers some of the questions asked by Jones and Kristiansen [14, Section 1.2], notably pushing further their method. Two of those questions are 1. Can the method be extended to richer languages? 2. How powerful and convenient is this method? We answer the first question positively, by adding treatment of function definition and calls, and by implementing the analysis on an actual programming language instead of a simple imperative language. Our work suggests that to answer the second question precisely, a lot of care is needed: the analysis uses matrices in a non-deterministic way to compute the influence of variables on each others, resulting in objects growing exponentially in number. By altering the semi-ring at the core of the original analysis, we show not only that more parsimonious methods can be used, but also that the mathematical machinery can be substituted. While our alternative approach essentially carries out the same analysis, we improved the implementability (and in fact implemented it already), re-usability and efficiency of the techniques while illustrating that the general method could be adapted easily for different types of analysis.

1.1 Complexity, resource growth and implementations: a brief tour

Our approach is conceptually guided by implicit computational complexity, that generally focus on restricting e.g. recursion [7,18] or type systems [5,17] to insure that a programming language captures a particular complexity class, or perform amortized resource analysis [13]. The particular domain concerned here, *data-flow analysis*, more specifically focuses on measuring or restricting loops in imperative programs [14,16,24] and was implemented on e.g. low level assembly-like programs [21].

However, our work is probably best compared with approaches coming from other communities sharing the same goal of finding worst case resource consumption. Complexity analyzers of different languages, such as SPEED [12] for `C++`, COSTA [2] for `Java` bytecode, RaML for `OCaml` [20] or Cerco [3] for `C`

all attempts to generate (possibly certified) cost annotation on (subsets of) programming languages in use.

1.2 Contribution: a different take on implementing the theory

We would like to argue that the "mwp" approach we are extending and making more practical is different from the previously mentioned implementations in four respects: 1. it is focused on the *growth* of variables instead of focusing on their possible values, 2. it is modular, in the sense that the internal machinery can be altered – as we illustrate in this paper – without the need to re-develop large chunks of the theory, 3. it is at the same time language-independent, as it reasons abstractly on imperative languages, and easy to port, as we illustrate with our implementation [4], 4. it focuses on characterizations of "chunks" of any size of the program allowing to abstract values and their encoding.

2 Background: the original flow analysis

We here quickly recall the original syntax of the imperative language, that we will extend with function call and definition in Sect. 4, then the original analysis by Jones and Kristiansen and its mathematical machinery.

2.1 Language analyzed: fragments of imperative language

We will be using the following imperative programming language, where variables range over `R`, `X`, `X'`, `Y`, `Z` and `Xi` for `i` $\in \mathbb{N}$, and need not to be declared, and given the binary operations on expression $-$, $+$, and $\times$, and on booleans $\bullet$ (such as $\wedge$, $\vee$, etc.), and a collection of comparison operators between expressions$\square$.

```
e ::= X | e - e | e + e | e * e                          (Expression)
b ::= e□e | b•b                                  (Boolean expression)
C ::= X = e | if b then C else C | while b do {C}
      loop X {C} | C ; C                                 (Command)
```

The semantics is straightforward, with `loop X {C}` meaning "do `C` `X` times" and `C;C` being used for sequentiality ("do `C`, then `C`"). We generally write "program" for a series of commands composed sequentially.

2.2 A Flow Calculus of mwp-Bounds for Complexity Analysis

The original paper [14] studies *flows* between variables in imperative programs, that are of three types: *m*aximum, *w*eak polynomial and *p*olynomial flows characterize the three forms of controls from one variable to another, with increasing

growth rate[4]. The programs are written in (a variation on) the language presented in Sect. 2.1, and the bounds are represented and calculated thanks to vector and matrices whose coefficients are elements of the mwp semi-ring.

Definition 1 (mwp semi-ring, matrix algebra). *Letting* $\text{MWP} = \{0, m, w, p\}$ *with* $0 < m < w < p$*, and* α*,* β*,* γ *range over* MWP*, the* mwp semi-ring $(\text{MWP}, 0, m, +, \times)$ *is defined with* $+ = \max$*,* $\alpha \times \beta = \max(\alpha, \beta)$ *if* $\alpha, \beta \neq 0$*, and* 0 *otherwise.*

Fixing a natural number n*, we use* M*,* A*,* B*,* C*, ... to denote* $n \times n$ *matrices over* MWP*,* M_{ij} *for the coefficient in the* i*th row and* j*th column of* M*,* $+$ *for the component wise addition, and* $\times$ *for the product of matrices defined in a standard way. The* 0*-element for the 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* $\mathbb{M}(\text{MWP})$ *is a semi-ring. Finally,* $M^0 = 1$*,* $M^{m+1} = M \times M^m$ *and the closure operator* $\cdot^*$ *is defined as* $M^* = 1 + M + (M^2) + \ldots$.

Although not crucial to understand our development, details about strong semi-rings and the mwp semi-ring can be found in Sect. A.1, and the general construction of a semi-ring whose elements are matrices with coefficients in a different semi-ring – so, in particular, $\mathbb{M}(\text{MWP})$ – is given in Sect. A.2.

Below, we let V_1, V_2 be column vectors with values in MWP, αV_1 to be the usual scalar product, and $V_1 + V_2$ to be defined component-wise. We write $\{{}^{\alpha}_{i}\}$ for the vector with 0 everywhere except for α in its ith row, and $\{{}^{\alpha}_{i}{}^{,\beta}_{j}\}$ for $\{{}^{\alpha}_{i}\} + \{{}^{\beta}_{j}\}$. Given a matrix M and a vector V, $M \overset{j}{\leftarrow} V$ is M with the jth column vector replaced by V. We write $\{{}^{\alpha}_{i}{\rightarrow}\, j\}$ for the matrix M with $M_{ij} = \alpha$ and 0 everywhere else, and var(`e`) for the set of variables in the expression `e`. In the analysis, the assumption is made that exactly n different variables are manipulated throughout the analyzed program, n-vectors are assigned to expressions and $n \times n$ matrices are assigned to commands using rules reminded in Figure 1 [14, Section 5].

The intuition is that if $\vdash_{\text{JK}} \texttt{C} : M$ can be derived using these rules, then all the values computed by `C` will grow at most polynomially w.r.t. its inputs: this is the core and powerful result of the original paper [14, Theorem 5.3]. 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 when used in loops – hence the condition on the L and W rules. Although simple in appearance, the proof techniques are far from trivial, but the relative simplicity of the derivation and of the matrices manipulated make the analysis flexible and easy to carry.

In fact, the assignment may fail in case of loops—that is, some programs may not be assigned a matrix—, as at least one of the variables used in the body of the loop may depend "too strongly" upon another, making it impossible to ensure polynomial bounds on the loop itself (as iterating the body can lead to super-polynomial dependencies).

[4] Note that 0 is also a possible type, corresponding to the absence of any dependency.

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

$$\star \in \{+,-\}\ \frac{\overline{\vdash_{\text{JK}} \texttt{e1} : V_1} \quad \overline{\vdash_{\text{JK}} \texttt{e2} : V_2}}{\vdash_{\text{JK}} \texttt{e1}\star\texttt{e2} : pV_1 + V_2}\ \text{E3} \qquad \star \in \{+,-\}\ \frac{\overline{\vdash_{\text{JK}} \texttt{e1} : V_1} \quad \overline{\vdash_{\text{JK}} \texttt{e2} : V_2}}{\vdash_{\text{JK}} \texttt{e1}\star\texttt{e2} : V_1 + pV_2}\ \text{E4}$$

(a) Rules for assigning vectors to expressions

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

$$\frac{\vdash_{\text{JK}} \texttt{C1} : A \quad \vdash_{\text{JK}} \texttt{C2} : B}{\vdash_{\text{JK}} \texttt{if b then C1 else C2} : A + B}\ \text{I}$$

$$\forall i, M^*_{ii} = m\ \frac{\vdash_{\text{JK}} \texttt{C} : M}{\vdash_{\text{JK}} \texttt{loop Xl \{C\}} : M^* + \{^{p}_{1} \to j \mid \exists i, M^*_{ij} = p\}}\ \text{L}$$

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

(b) Rules for assigning matrices to commands

Fig. 1: Original ("Jones-Kristiansen") rules

To capture a larger class of programs, the calculus used to assign a matrix to a program – that corresponds to a proof search in a derivation system – is non-deterministic. As a consequence, multiple matrices—hence, multiple polynomial bounds—may be assigned to the same program.

We will use the following example (of "iteration-dependent" loop [14, Example 3.4]) as a common basis to discuss our improvements.

Example 1. Consider the command `loop X3{X2= X1 + X2}`. The body of the loop – the expression `X1 + X2` – admits 3 different derivations that we name π_0, π_1 and π_2:

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

$$\frac{}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}}\ \text{E2}$$

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

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

Similarly, because of the w coefficient on the diagonal after applying A, π_2 cannot be used to complete the derivation either, and hence only one derivation for this command holds. Note that in general, multiple derivations can exist and that this "indeterminacy" [14, Section 8] is needed to capture as many programs as possible.

3 "Taming" non-determinism and non-termination

The first two improvements over the existing analysis we offer are to:

1. "internalize" the non-determinism, so that at most one matrix per command is produced,
2. "internalize" the failure, so that at least one matrix per command is produced.

These changes were introduced first to obtain an efficient (or, actually, to simply enable an) implementation, but they came with by-products. Indeed, the naive approach consisting in producing a list of all possible matrices corresponding to all the non-deterministic choices (and removing those matrices for which the analysis fails) would result in a very slow implementation even for small programs.

To represent non-determinism, we use in the matrices *functions from choices to coefficients in* MWP instead of simply coefficients in MWP. This is explained by the following remark – made formal in Sect. A.3: the overall analysis produces a function from a space of choices C to the space $\mathbb{M}(\text{MWP})$ of matrices over the mwp semi-ring, i.e. it results in a function $C \to \mathbb{M}(\text{MWP})$. But there is a semi-ring isomorphism between $C \to \mathbb{M}(\text{MWP})$ and $\mathbb{M}(C \to \text{MWP})$, i.e. matrices whose coefficients are functions from choices to the mwp semi-ring. We use this, together with a clever representation of the space $C \to \mathbb{M}(\text{MWP})$ to provide an alternative formalism allowing for more efficient implementation. Moreover, compacting all the possible derivations into one matrix results in a gain of space and time as different matrices obtained from different choices are *more or less* the same, i.e. they usually differ only on a few coefficients, leading to a quite compact representation. As a side-product, this also allows the user to be presented with different polynomial bounds, so that they can pick the one that suits their needs.

Concerning failure, we extend the mwp semi-ring with a special value ∞; one key point is that the resulting structure is *not* a *strong* semi-ring–as opposed to mwp or $\mathbb{M}(\text{MWP})$–because the latter structure requires the equality $0 \times \infty = 0$ to hold while we need $0 \times \infty = \infty$ to avoid overlooking some super-polynomial computations: if part of the program computes an exponential value but then

throws it away, $0 \times \infty = 0$ would hide the super-polynomial computation, resulting in an incorrect analysis[5]. This way of representing failure also has the advantage of being local, so that which input variable impacts which variable in a non-polynomial way can be precisely pinpointed. We believe this feature can be of crucial use in a situation where some variables are known to be of small size, hence where a non-polynomial bound *on particular input variables* is acceptable.

Taken together, our improvements insure that exactly one matrix will always be assigned to a program, but also gives an opportunity to chose between "the lesser of two evils" when it fails: if two derivations produce ∞ coefficients *on different flows*, the user could decide to privilege one over the other based on knowledge about the inputs' sizes.

We give in Figure 2 the alternative system we are introducing in full, but will gently discuss it though the remaining parts of this section and in Sect. 4: note that the A, C and I rules are unchanged (even if the sum and product are in a different semi-ring) and that the call rule is new.

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

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

(a) New rules for assigning vectors to expressions

$$\frac{\vdash \texttt{e} : V}{\vdash \texttt{Xj = e} : 1 \overset{\texttt{j}}{\leftarrow} V}\ \mathrm{A} \qquad \frac{\vdash \texttt{C1} : A \quad \vdash \texttt{C2} : B}{\vdash \texttt{C1 ; C2} : A \times B}\ \mathrm{C} \qquad \frac{\vdash \texttt{C1} : A \quad \vdash \texttt{C2} : B}{\vdash \texttt{if b then C1 else C2} : A + B}\ \mathrm{I}$$

$$\frac{\vdash \texttt{C} : M}{\vdash \texttt{loop Xl \{C\}} : M^* + \{^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} + \{^{p}_{1} \to j \mid \exists i, M^*_{ij} = p\}}\ \mathrm{L}^{\infty}$$

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

$$\frac{}{\vdash \texttt{Xi = F(X1,..., XN)} : 1 \overset{\texttt{i}}{\leftarrow} ((0 \mapsto M(f)_0) + \ldots + (k \mapsto M(f)_k))}\ \text{call}$$

(b) New rules for assigning matrices to commands

Fig. 2: New rules

[5] Here we can be a bit more detailed: while throwing away the infinite coefficient would hide the super-polynomial computation, it would not contradict the *ultimately* polynomial dependency of the values w.r.t. the inputs. As such, $0 \times \infty = 0$ could still be used to bounds values, at the cost of losing the bounds on time and space usage for terminating programs. A modular implementation allowing to decide which structure to use in under progress.

3.1 Choice data flow semi-rings

The first step towards our "internalization of choice" is to design the correct semi-ring. We start by reasoning abstractly, the detail of this construction is given in Sect. A.3. Given a strong semi-ring $\mathbb{S}$, we define $\mathbb{M}(\mathbb{S})$ to be the strong semi-ring whose elements are matrices with coefficients in $\mathbb{S}$ (Lemma 3), similarly to the matrix algebra of Definition 1. We also define $A \to \mathbb{S}$ to be the strong semi-ring whose elements are functions from a set (of choices) A to $\mathbb{S}$ (Lemma 4). We furthermore observe (Lemma 5) that for all set A and strong semi-ring $\mathbb{S}$, $\mathbb{M}(A \to \mathbb{S})$ and $A \to \mathbb{M}(\mathbb{S})$ are isomorphic (Definition 3). By chosing $A = \prod_{i=1}^{p} A_i$, it follows that there exists an isomorphism

$$\mathbb{M}(\prod_{i=1}^{p} A_i \to \mathbb{S}) \cong \prod_{i=1}^{p} A_i \to \mathbb{M}(\mathbb{S})$$

for all family of sets $(A_i)_{i=1,\dots,p}$, using the usual cartesian product of sets. This dual nature of the semi-ring considered will be useful:

- we implement the analysis by assigning elements of $\mathbb{M}(\prod_{i=1}^{p} A_i \to \text{MWP})$, this allows for a more efficient implementation by using some clever representation of elements of $\prod_{i=1}^{p} A_i \to \text{MWP}$ detailed in Sect. 5;
- we use the representation of the resulting matrix M as an element of $\prod_{i=1}^{p} A_i \to \mathbb{M}(\text{MWP})$ to produce, from an *assignment* $\alpha = (a_1, a_2, \dots, a_p) \in \prod_{i=1}^{p} A_i$, a matrix $M[\alpha] \in \mathbb{M}(\text{MWP})$, recovering the *mwp*-flow that would have been computed by making the choices $a_1, a_2, \dots$ in the derivation.

Remark 1. As the unique degree of non-determinism in the rules to assign a matrix to commands is 3 at this point (cf. Example 1), our modification of the analysis flow consists simply (for the moment) in 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. Note that in a later section, other sets A_i will be used in order to deal with *function calls*.

Example 2. Re-using the derivations π_0, π_1 and π_2 from Example 1, we can now represent the three vectors $\begin{pmatrix} p \\ m \\ 0 \end{pmatrix}$, $\begin{pmatrix} m \\ p \\ 0 \end{pmatrix}$ and $\begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$ with a single vector

$$\begin{pmatrix} \{0 \mapsto p, 1 \mapsto m, 2 \mapsto w\} \\ \{0 \mapsto m, 1 \mapsto p, 2 \mapsto w\} \\ 0 \end{pmatrix}$$

Where we make the abuse of notation of writing 0 for $\{0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0\}$.[6] Since, in particular[7], $\mathbb{M}(\{0, 1, 2\} \to \text{MWP}) \cong \{0, 1, 2\} \to \mathbb{M}(\text{MWP})$, the obtained vector can be rewritten as $0 \mapsto \begin{pmatrix} p \\ m \\ 0 \end{pmatrix}, 1 \mapsto \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}, 2 \mapsto \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$.

[6] The implementation supports both coefficients from MWP *and* coefficients from $\{0, 1, 2\}^m \to \text{MWP}$, cf. e.g. a simple assignment `assign_expression` example.

[7] This is a variant of Lemma 5. While the latter lemma is stated for an algebra 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.

Our derivation system replaces the E3 and E4 rules with a single rule E^{A} (for "additive"), and imposes an additional restriction on E2, thus giving E^{M} (for "multiplicative"), so that it is used *only* when E1 followed by E2 or E3 cannot be applied.

The implementation of binary additive operators ($-$ and $+$) with E^{A} captures all possible choices for distinct operands and merges `i` and `j` into a single coefficient when `i` = `j`. Binary multiplication is handled by applying the E^{M} rule – note that the application of E2 to additive operators in the original system is still handled by the last choice present in E^{A}. Given this need to treat binary operations differently, based on operators and combinations of operands, more work is needed to handle statements of greater arity. As the implementation already processes abstract syntax trees of `C` commands recursively, handling operations of greater arity will require implementing additional recursive steps, but we do not expect that to be problematic conceptually or at the level of implementation. At the light of this reflection, and knowing that there is no benefit in applying E2 to a single variable, as it result in a w coefficient being applied in lieu of a lesser m coefficient, it is easy to observe that E^{A} and E^{M} are as expressive as E1, E2, E3 and E4 taken together – something we will be using when proving the equi-expressiveness of our system (Lemma 1).

3.2 Representing failure with an "infinity" coefficient

The original analysis would stop whenever a non-polynomial flow was detected, putting an end to the chosen strategy (i.e. set of choices) and restarting from scratch with another one. We will now discuss the fact that every derivation can be completed even in the presence of non-polynomial flows, which constitutes our second improvement. This is done by first extending the mwp semi-ring with a new element. While this approach results in derivations for program where some variables *are not* polynomially related to their inputs, we argue that pinpointing which variables are "faulty" from within the analysis can have benefits.

The first step is to incorporate a top element ∞ into our semi-rings to represent undefined elements. The semi-ring $\textsc{mwp}^{\infty}$ we will be using is hence $(\textsc{mwp} \cup \{\infty\}, 0, m, +^{\infty}, \times^{\infty})$, 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 $\times^{\infty}$ insures that once non-polynomial flows have been detected, they cannot be erased (as $\infty \times^{\infty} 0 = \infty$), but comes at the price of the strength of the semi-ring (the details are discussed in Sect. A.4).

Below, we will work with $\mathbb{M}(\textsc{mwp}^{\infty})$, write $\times$ for $\times^{\infty}$ and similarly for $+$, and remind the reader that we write $\{{}^{\alpha}_{i}\!\!\rightarrow j\}$ for the matrix M with $M_{ij} = \alpha$ and 0 everywhere else. The only cases where the original analysis may fail is if the side condition of L or W (Figure 1) are not met; we now replace those by the rules L^{∞} and W^{∞} of Figure 2, with no side condition.

Those rules, which can always be applied, simply replace the problematic coefficients with ∞. Note that in the cases for which the original rule is applicable, the results coincide. This will be essential to prove that our modified analysis is coherent with Jones and Kristiansen's original approach (Lemma 1).

3.3 Merging the two improvements: illustration with operations

We introduced and discussed the deviations from the original system for the "axiomatic" / "expression" ($\mathrm{E^A}$, $\mathrm{E^M}$) and "loop" rules ($\mathrm{L^\infty}$ and $\mathrm{W^\infty}$), but remains to briefly discuss the rules for assignment (A), `if` (I) and the composition (C), that remained unchanged. Those rules are the place where both improvements meet. Mathematically speaking, adopting the semi-ring defined over matrices using coefficients in $\{0,1,2\}^m \to \text{MWP} \cup \{\infty\}$ is fairly simple, but computationally speaking, simple operations like multiplication and addition of matrices become very costly and memory-demanding. This became particularly problematic when keeping a usable implementation in mind, and is illustrated below.

Example 3. In our new system, consider the following derivation:

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

with

$$V = 0 \mapsto \{{}^{m}_{1}\, {}^{p}_{2}\} + 1 \mapsto \{{}^{p}_{1}\, {}^{m}_{2}\} + 2 \mapsto \{{}^{w}_{1}\, {}^{w}_{2}\}$$
$$V' = 0 \mapsto \{{}^{m}_{1}\, {}^{p}_{3}\} + 1 \mapsto \{{}^{p}_{1}\, {}^{m}_{3}\} + 2 \mapsto \{{}^{w}_{1}\, {}^{w}_{3}\}$$
$$1 \overset{1}{\leftarrow} V = \begin{pmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix} \overset{1}{\leftarrow} V \cong \begin{pmatrix} (0\mapsto m)+(1\mapsto p)+(2\mapsto w) & 0 & 0 \\ (0\mapsto p)+(1\mapsto m)+(2\mapsto w) & m & 0 \\ 0 & 0 & m \end{pmatrix}$$
$$1 \overset{1}{\leftarrow} V = \begin{pmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix} \overset{1}{\leftarrow} V' \cong \begin{pmatrix} (0\mapsto m)+(1\mapsto p)+(2\mapsto w) & 0 & 0 \\ 0 & m & 0 \\ (0\mapsto p)+(1\mapsto m)+(2\mapsto w) & 0 & m \end{pmatrix}$$

Now, to perform the addition required by the I rule, some care is needed: indeed, the choices in the left branch of the derivation are independent from the choices in the right branch, and we must use coefficients in $\{0,1,2\}^2 \to \text{MWP}$ to represent the 2^3 choices. Assuming the choice in the left branch is first, we obtain e.g. for the beginning of the top-left coefficient (the complete coefficient will be given below, once we introduced a more compact notation):

$$(0 \mapsto (0 \mapsto (m{+}m = m))){+}(0 \mapsto (1 \mapsto (m{+}p = p))){+}(0 \mapsto (2 \mapsto (m{+}w = w)))$$

Writing $ab \mapsto$ for $a \mapsto b \mapsto$, with $a, b \in \{0,1,2\}$, and $a\Box \mapsto$ (resp. $\Box a \mapsto$) if the second (resp. first) choice has no impact on the resulting coefficient, we can let:

$$A = 00 \mapsto m + 01 \mapsto p + 02 \mapsto w + 1 \mapsto p + 20 \mapsto w + 21 \mapsto p + 22 \ \mapsto w$$

to obtain

$$(1 \overset{1}{\leftarrow} V) + (1 \overset{1}{\leftarrow} V') = \begin{pmatrix} A & 0 & 0 \\ (0\Box\mapsto p)+(1\Box\mapsto m)+(2\Box\mapsto w) & m & 0 \\ (\Box 0\mapsto p)+(\Box 1\mapsto m)+(\Box 2\mapsto w) & 0 & m \end{pmatrix}$$

Although the presentation and numbering diverge a bit, the example at `https://seiller.github.io/pymwp/demo/#improvement_paper_example3.c` can help the curious reader to check that the implementation reflects this derivation correctly.

Example 4. Re-using Example 1, we now obtain in our new system a derivation that assign to `loop X3{X2 = X1 + X2}` the unique matrix

$$\begin{pmatrix} m & (0\mapsto p)+(1\ \mapsto m)+(2\mapsto w) & 0 \\ 0 & (0\mapsto m)+(1\ \mapsto \infty)+(2\mapsto \infty) & 0 \\ 0 & (0\mapsto p)+(1\ \mapsto 0)+(2\mapsto 0) & m \end{pmatrix}$$

where we observe that 1. only one choice (0) – one assignment – 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 impact the matrix only *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. This example was not implemented, as `loop` is not a standard `C` operator, but is currently being implemented as a restricted form of `for` loop (cf. `https://github.com/seiller/pymwp/issues/5`).

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

Lemma 1. *Given a program P, there is a single matrix $M \in \mathbb{M}(\{0,1,2\}^p \to \text{MWP}^\infty)$ such that $P \vdash M$, i.e. the system is deterministic. Moreover, for any assignment $\alpha = (a_1, \ldots, a_p) \in A^p$, we have that*

$$P \vdash_{\text{JK}} M[\alpha] \text{ if and only if } M[\alpha] \in \mathbb{M}(\text{MWP}).$$

This shows that the performed analyses coincide, as $M[\alpha] \in \mathbb{M}(\text{MWP})$ implies that no ∞ coefficient occurs in it. However, our alternative definition should be understood as an important improvement, as it allows for a more efficient implementation (Sect. 5). But before discussing the efficiency of the implementation, we will now explain the natural but important extension to function calls enabled by our alternative formalism.

4 Extending the analysis with function calls

We begin by extending the syntax presented in Sect. 2.1 by adding *function declarations* `F ≔ f(X1, ..., XN){C; return R}` and a command that performs a function call and assign its return value to a variable `Xi = F(X1, ..., XN)`[8]. A *program* is now a series of function declarations, with one of them called `main` with $N = 0$, and such that all the commands of the form `Xi = F(X1,..., XN)` refers to a function previously declared. A *chunk* is simply a series of commands inside a function declaration[9].

One of the key points of our contribution is the extension of the analysis to function calls, in a way that can be used in practice, as we handle a function f

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

[9] Note that this implies that if a loop belongs to a chunk, then the entire loop body belongs to the chunk.

with a single analysis that stores a minimal amount of data for latter calls. The principle is the following: given the matrix $M(f)$ obtained from the analysis of the program computing f, we store only the k choices for which no ∞ coefficients appear, and then project them to only keep track of the different input/output behaviors, merging choices leading to the same result. After this operation, we are left with a family $M(f)_0, M(f)_1, \ldots, M(f)_k$ of matrices[10] that should be understood as providing quantitative (i.e. polynomial, weak polynomial, maximum, or zero) information about the dependency of output values w.r.t. input values. Now, the analysis of the command calling the function f is dealt with by the call rule of Figure 2.

Formally, we show that our definition of composition is coherent with the initial analysis as follows. We consider two *chunks*: the first chuck P contains a call to a function f, the second is obtained by replacing within P the call to the function f by inserting in its place the sequence of commands F computing f. This second chunk is called $P[F]$. We then prove that the matrix associated to P is "the same"[11] Intuitively, this mechanism provides the expected result because the choices made in the chunk F do *not* affect the context $P[\cdot]$, and the variables used in the chunk F are *not* used in the context $P[\cdot]$ except for the return variables.

More formally, let P be a chunk of program, containing a call to the function f, and let F be the chunk computing the function f. We define from P the *context* $P[\cdot]$, a chunk containing a hole $[\cdot]$ to be filled with the chunk F, obtained as follows (supposing f has a single output variable).

- We remove the line with the function call, say `Xi=f(X1, ..., XN);`.
- We add in place the following lines, where `R`, `Y1`, ..., `Yn` are fresh variables:

```
Y1 = X1;
...
YN = XN;
[·]
Xi = R;
```

The code $P[F]$ is then obtained by defining a chunk $\tilde{F}$, and inserting it in place of the symbol $\cdot$ in $P[\cdot]$. The chunk $\tilde{F}$ is obtained as follows from F:

- the header is removed,
- the input variables of F are renamed to `Y1`, `Y2`, ..., `YN`,
- the variable returned by f is renamed to `R`, the `return` statement is removed,
- all other variables are renamed if needed to avoid using the same names as the variables in $P[\cdot]$. We write the set of these variables V_F.

[10] To ease the presentation, the syntax considered here is restricted to functions with a single output value, so we actually have vectors in place of matrices here. But it is more natural to think in terms of matrices here, as the overall approach is valid in the more general setting in which functions may have several output values, and then the obtained objects are indeed matrices.

[11] Here one has to consider equality up to some projections as the chuck F inserted in P may introduce new choices and use additional variables.

Example 5. Refer to Figure 3 for a simple example of the code transformation for in-lining a function call.

$P =$
```
int main(){
        X3 = X1 + X2;
        X2 = X3 + X1;
        X1 = f(X2);
}
```

$P[\cdot] =$
```
int main(){
        X3 = X1 + X2;
        X2 = X3 + X1;
        Y1 = X2;
        [·]
        X1 = R;
}
```

$Q =$
```
int f(int X1){
        loop X1{X2 = X2 + X3};
        return X2;
}
```

$\tilde{Q} =$
```
loop Y1{R = R + X4};
```

Fig. 3: A simple example of "inlining" a function call

Now, we can compute both matrices:
- $M(P)$ where the line `Xi=f(X1, ..., XN);` is analysed using the call rule, and
- $M(P[F])$.

We write $\Pi_P(M(P[F]))$ the projection of $M(P[F])$ onto the variables in P and $(1 - \Pi_P)(M(P[F]))$ the projection of $M(P[F])$ onto the variables *not in* P.

Some non-deterministic choices may appear within the (modified) chunk $\tilde{F}$ inside $P[F]$, i.e.
- the coefficients of the matrix $M(P)$ are elements of the semi-ring $\prod_{i=1}^{p+1} A_i \to \mathbb{M}(\text{MWP})$, with one particular choice corresponding to the call rule – we write the corresponding index i_0;
- the coefficients of $P[F]$ are elements of the semi-ring $\prod_{i=1}^{p+k} B_i \to \mathbb{M}(\text{MWP})$, where k choices are made within the chunk $\tilde{F}$ – we write the corresponding indexes $j_1, j_2, \ldots, j_k$ (note these are in fact consecutive indexes).

We note $\pi : \{1, \ldots, p+k\} \to \{1, \ldots, p+1\}$ the projection of the choices in $P[F]$ onto the corresponding choices in P, i.e.

$$\pi(j) = \begin{cases} j & \text{if } j < j_0 \\ i_0 & \text{if } j_0 \leqslant j < j_k \\ j - k + 1 & \text{if } j_k < j \end{cases}$$

We note that each matrix used as axiom in the function call corresponds to a specific assignment on indexes $j_1, \ldots, j_k$. We write $\Psi : A_{i_0} \to \prod_{i=j_1}^{j_k} B_i$ the corresponding injection. This is extended to $\bar{\Psi} : \prod_{i=1}^{p+1} A_i \to \prod_{i=0}^{p+k} B_i$ in a straightforward way.

We can now state the main theorem showing that the call rule adequately analyses function calls.

Theorem 1. *For all assignment α of $\prod_{i=1}^{p+1} A_i$,*

$$M(P)[\alpha] = (1 - \Pi_P)(M(P[F]))[\bar{\Psi}(\alpha)]$$

Moreover, for all assignment β of $\prod_{i=0}^{p+k} B_i$ not in $\mathrm{Im}(\bar{\Psi})$, the matrix $(1-\Pi_P)(M(P[F])[\beta])$ contains an infinite value.

Proof. To prove this, we first notice that it is sufficient to prove it for the simplest chuck P containing only one command: `Xi = f(X1, ..., XN)`. This is explained by the compositional nature of the analysis (a sequence of commands is simply assigned the product of the matrices of each individual command). Then, checking that the theorem holds in this case is a straightforward, though tedious (due to keeping track of all indices), computation. □

5 Implementation of the analysis

The formulation of the extended mwp analysis exposed in the previous sections was also intended for implementation. As such, the choice of the representation of non-determinism – for instance – was also guided by our wish for a faster analysis, something not discussed in depth in our tool paper [4], or our documentation. In this section, we expose some of the specific choices made in the implementation.

5.1 Non-determinism, and the challenges to efficient calculations

As explained in the above sections, the result of the analysis is a matrix with coefficients in a semi-ring of the form $\prod_{i=1}^{p} A_i \to \mathbb{M}(\text{MWP})$– setting aside ∞ coefficients for a moment. To implement this correctly, we represent elements of this semi-ring as polynomials w.r.t. the generating set given by the functions $\delta(i,j) : \prod_{i=1}^{p} A_i \to \text{MWP}$ defined by $\delta(i,j)(a_1, a_2, \ldots, a_p) = m$ if $a_j = i$ and $\delta(i,j)(a_1, a_2, \ldots, a_p) = 0$ otherwise. i.e. an element of $\prod_{i=1}^{p} A_i \to \text{MWP}$ is represented as a polynomial $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ with $\alpha_i \in \text{MWP}$.

This basis have 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 in such a way that the product with another monomial is ordered. i.e. if $m \leqslant m'$ and both $m \times n$ and $m' \times n$ are non-zero, then $m \times n \leqslant m' \times n$. This order is leveraged to obtain efficient algorithms for computing operations on the representation of coefficients, similar to what is done using Gröbner bases for computation of standard polynomials. For instance, the algorithm for multiplication of polynomials makes use of the property above and proceeds as follows to compute the product of a polynomial P with $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ (supposing the representation of P is ordered):

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.

This clever method has some very concrete consequences. As an example, our `explosion.c` program calls the multiplication 11,907 times and could not be completed with a naive multiplication implementation. More precise profiling further exposes the need for this optimization.

5.2 Infinite values cluttering the analysis, and difficulties to evaluate

One very costly aspect of the analysis is the *evaluation* step which takes a matrix with coefficients in $\prod_{i=1}^{p} A_i \to \text{MWP}$ and checks all possible assignments $(a_1, \ldots, a_p) \in \prod_{i=1}^{p} A_i$ to look for infinite coefficients. While this step is necessary (in one form or another) if one wishes to produce the actual mwp matrices certifying polynomial bounds (something needed at least once to allow for function calls), we implemented a specific data structure allowing to keep track of infinite assignments on the fly, thus allowing the analysis to provide a qualitative answer quickly. I.e. the analysis can ensure the existence or not of mwp-bounds *without computing the corresponding matrix*.

This is implemented by a structure we called `delta_graphs`. This 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 it contains (the number of deltas $\delta(i, j)$ is 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. (Note here that our intuitions 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.) 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 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 $\sum_{k=0}^{\text{Card}(A_i)-1} m\delta(k, j) = m$.

Remember the delta_graph represents the subspace of assignments for which an infinite coefficient appeared. So 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, it means at least one assignment exists for which no infinite coefficients appear, and therefore at least one mwp-bound exists.

6 Future work

We here provide some details on extensions of this work that we are currently working on, or that will be tackled in the near future.

The first natural line of work is the extension of the language analysed, in particular to accommodate other data structures. While structures such as lists should not be problematic, dealing with pointer will certainly require more involved work, in particular to ensure the theoretical results obtained by Jones and Kristiansen hold, i.e. that the obtained mwp-bounds are indeed correct. These extensions, together with the extension to function calls discussed in this paper, will then be added within our implementation of the analysis.

A second line of work that was already started is to implement the analysis in the Compcert compiler [19], which would allow for a formal certification of the polynomial bounds computed by the analysis using the Coq proof assistant [1]. Some preliminary work in this direction was already done. In particular, it seems natural to use COMPCERT-SSA [6] to be later used as stepping stone towards an implementation within `llvm` – and if possible `certified-llvm` [25] – which would enable the analysis to programs written in other languages than `C`.

References

1. Coq documentation, `https://coq.github.io/doc/`
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Costa: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. pp. 113–132. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
3. Amadio, R.M., Ayache, N., Bobot, F., Boender, J.P., Campbell, B., Garnier, I., Madet, A., McKinna, J., Mulligan, D.P., Piccolo, M., Pollack, R., Régis-Gianas, Y., Sacerdoti Coen, C., Stark, I., Tranquilli, P.: Certified complexity (cerco). In: Dal Lago, U., Peña, R. (eds.) Foundational and Practical Aspects of Resource Analysis. pp. 1–18. Springer International Publishing, Cham (2014)
4. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: An implementation of flow calculus for complexity analysis (tool paper). In: Submitted to APLAS 2021 (2021)
5. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda-calculus. In: LICS. pp. 266–275. IEEE Computer Society (2004). `https://doi.org/10.1109/LICS.2004.1319621`
6. Barthe, G., Demange, D., Pichardie, D.: Formal verification of an ssa-based middle-end for compcert. ACM Trans. Program. Lang. Syst. **36**(1), 4:1–4:35 (2014). `https://doi.org/10.1145/2579080`
7. Bellantoni, S.J., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions (extended abstract). In: Kosaraju, S.R., Fellows, M., Wigderson, A., Ellis, J.A. (eds.) STOC. pp. 283–93. ACM (1992). `https://doi.org/10.1145/129712.129740`
8. Ben-Amram, A.M., Jones, N.D., Kristiansen, L.: Linear, polynomial or exponential? complexity inference in polynomial time. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) Logic and Theory of Algorithms, 4th

Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings. LNCS, vol. 5028, pp. 67–76. Springer (2008). https://doi.org/10.1007/978-3-540-69407-6_7
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973, http://dl.acm.org/citation.cfm?id=512950
10. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: Neuhold, E.J. (ed.) 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. pp. 237–278. North-Holland (1977)
11. Dal Lago, U.: A short introduction to implicit computational complexity. In: Bezhanishvili, N., Goranko, V. (eds.) ESSLLI. LNCS, vol. 7388, pp. 89–109. Springer (2011). https://doi.org/10.1007/978-3-642-31485-8_3
12. Gulwani, S., Mehra, K.K., Chilimbi, T.: 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. p. 127–139. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480898
13. Hofmann, M., Moser, G.: Multivariate amortised resource analysis for term rewrite systems. In: Altenkirch, T. (ed.) 13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland. LIPIcs, vol. 38, pp. 241–256. Schloss Dagstuhl (2015). https://doi.org/10.4230/LIPIcs.TLCA.2015.241, http://www.dagstuhl.de/dagpub/978-3-939897-87-3
14. Jones, N.D., Kristiansen, L.: A flow calculus of *mwp*-bounds for complexity analysis. ACM Trans. Comput. Log. **10**(4), 28:1–28:41 (2009). https://doi.org/10.1145/1555746.1555752
15. Jones, N.D., Nielson, F.: Abstract Interpretation: A Semantics-Based Tool for Program Analysis, Handbook of Logic in Computer Science, vol. 4, pp. 527 – 636. Oxford University Press (1995)
16. Kristiansen, L., Niggl, K.H.: On the computational complexity of imperative programming languages. Theor. Comput. Sci. **318**(1–2), 139–161 (Jun 2004). https://doi.org/10.1016/j.tcs.2003.10.016
17. Lafont, Y.: Soft linear logic and polynomial time. Theor. Comput. Sci. **318**(1), 163–180 (2004). https://doi.org/10.1016/j.tcs.2003.10.018
18. Leivant, D.: Stratified functional programs and computational complexity. In: Van Deusen, M.S., Lang, B. (eds.) POPL. pp. 325–333. ACM Press (1993). https://doi.org/10.1145/158511.158659
19. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
20. Lichtman, B., Hoffmann, J.: Arrays and references in resource aware ML. In: Miller, D. (ed.) 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK. LIPIcs, vol. 84, pp. 26:1–26:20. Schloss Dagstuhl (2017). https://doi.org/10.4230/LIPIcs.FSCD.2017.26, http://www.dagstuhl.de/dagpub/978-3-95977-047-7

21. Moyen, J.Y.: Resource control graphs. ACM Trans. Comput. Logic **10**(4) (Aug 2009). https://doi.org/10.1145/1555746.1555753
22. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk detection. In: D'Souza, D., Kumar, K.N. (eds.) ATVA. LNCS, vol. 10482. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_7
23. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk motion by peeling with statement composition. In: Bonfante, G., Moser, G. (eds.) 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. EPTCS, vol. 248, pp. 47–59 (2017). https://doi.org/10.4204/EPTCS.248.9, http://arxiv.org/abs/1704.05169
24. Niggl, K., Wunderlich, H.: Certifying polynomial time and linear/polynomial space for imperative programs. SIAM J. Comput. **35**(5), 1122–1147 (2006). https://doi.org/10.1137/S0097539704445597
25. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formal verification of ssa-based optimizations for LLVM. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 175–186. ACM (2013). https://doi.org/10.1145/2491956.2462164

A Technical Appendix on Semi-rings

A.1 The mwp semi-ring

This subsection briefly recall semi-ring definition and proves that the mwp semi-ring is indeed a semi-ring.

Definition 2 (Semi-ring). *A semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$ *is specified by a set* S *and two binary operations* $+$ *(addition) and* $\times$ *(multiplication) such that* $\{0, 1\} \in S$ *and*

1. $(S, 0, +)$ *is a commutative monoid: the operation* $+$ *is associative, commutative, and has* 0 *as the identity element,*
2. $(S, 1, \times)$ *is a monoid: the operation* $\times$ *is associative and has* 1 *as the identity element,*
3. *the operation* $\times$ *distributes with respect to* $+$*: for all* $a, b, c \in S$*,* $a \times (b + c) = a \times b + a \times c$ *and* $(b + c) \times a = b \times a + c \times a$

   We call $\mathbb{S}$ *a* strong *semi-ring if, additionally,* 0 annihilates S*, i.e.*

4. $0 \times a = a \times 0 = 0$ *for all* $a \in S$*.*

Lemma 2 (mwp-semi-ring). *The tuple* $(\{0, m, w, p\}, 0, m, +, \times)$*, with*

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

$$- \ \alpha \times \beta = \begin{cases} \alpha + \beta & \textit{if } \alpha \neq 0 \textit{ and } \beta \neq 0 \\ 0 & \textit{otherwise} \end{cases}$$

is a strong *semi-ring.*

Proof. We prove that $(\{0, m, w, p\}, 0, m, +, \times)$ as defined respects the conditions of Definition 2. The proof is straightforward but detailed nevertheless.

$(\{0, m, w, p\}, 0, +)$ **is a commutative monoid** We prove that $(\{0, m, w, p\}, +)$ is a commutative monoid by showing that it is associative, commutative, and has 0 as identity.

Associativity $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$

Case 1: $\alpha \geqslant \beta \geqslant \gamma$

$$\begin{aligned} & \alpha = \alpha \\ \implies \quad & \alpha + \gamma = \alpha + \beta \\ \implies \quad & (\alpha + \beta) + \gamma = \alpha + (\beta + \gamma) \end{aligned}$$

Case 2: $\alpha \geqslant \gamma \geqslant \beta$

$$\begin{aligned} & \alpha = \alpha \\ \implies \quad & \alpha + \gamma = \alpha + \gamma \\ \implies \quad & (\alpha + \beta) + \gamma = \alpha + (\beta + \gamma) \end{aligned}$$

Case 3: $\beta \geqslant \alpha \geqslant \gamma$

$$\begin{aligned} & \beta = \beta \\ \implies \quad & \beta + \gamma = \alpha + \beta \\ \implies \quad & (\alpha + \beta) + \gamma = \alpha + (\beta + \gamma) \end{aligned}$$

Case 4: $\beta \geqslant \gamma \geqslant \alpha$

$$\begin{aligned} & \beta = \beta \\ \implies \quad & \beta + \gamma = \alpha + \beta \\ \implies \quad & (\alpha + \beta) + \gamma = \alpha + (\beta + \gamma) \end{aligned}$$

Case 5: $\gamma \geqslant \alpha \geqslant \beta$

$$\begin{aligned} & \gamma = \gamma \\ \implies \quad & \alpha + \gamma = \alpha + \gamma \\ \implies \quad & (\alpha + \beta) + \gamma = \alpha + (\beta + \gamma) \end{aligned}$$

Case 6: $\gamma \geqslant \beta \geqslant \alpha$

$$\begin{aligned} & \gamma = \gamma \\ \implies \quad & \beta + \gamma = \alpha + \gamma \\ \implies \quad & (\alpha + \beta) + \gamma = \alpha + (\beta + \gamma) \end{aligned}$$

Commutative Property Both cases are immediate:

Case 1: $\alpha \geqslant \beta \implies \alpha + \beta = \alpha = \beta + \alpha$

Case 2: $\beta \geqslant \alpha \implies \alpha + \beta = \beta = \beta + \alpha$

Identity element is 0

$$0 + 0 = 0 \qquad 0 + m = m \qquad 0 + w = w \qquad 0 + p = p$$

$(\{0, m, w, p\}, m, \times)$ **is a monoid** We now prove that $(\{0, m, w, p\}, m, \times)$ is a monoid by showing that it is associative, has m as identity, and has 0 as the annihilator.

Associativity $(\alpha \times \beta) \times \gamma = \alpha \times (\beta \times \gamma)$

Case 1: $\alpha, \beta, \gamma \in \{m, w, p\}$

$\alpha \times \beta = \alpha + \beta$ Associativity of operation $+$ is shown in the proof of the commutative monoid, $(\{0, m, w, p\}, +)$.

Case 2: α, β, or γ equals 0

By definition of multiplication, the product is 0.

Identity element is m

$$\begin{aligned} 0 \times m &= 0 = m \times 0 \\ m \times m &= m = m \times m \\ w \times m &= w = m \times w \\ p \times m &= p = m \times p \end{aligned}$$

0 annihilates $\{0, m, w, p\}$

$$\begin{aligned} 0 \times 0 &= 0 = 0 \times 0 \\ m \times 0 &= 0 = 0 \times m \\ w \times 0 &= 0 = 0 \times w \\ p \times 0 &= 0 = 0 \times p \end{aligned}$$

Distribution of multiplication over addition We conclude by proving that $\times$ distributes over $+$.

Right Distribution $\alpha \times (\beta + \gamma) = (\alpha \times \beta) + (\alpha \times \gamma)$

Case 1: $\beta \geqslant \gamma$

$$\begin{aligned} &\implies && \alpha \times \beta = \alpha \times \beta \\ &\implies && \alpha \times (\beta + \gamma) = (\alpha \times \beta) + (\alpha \times \gamma) \end{aligned}$$

Case 2: $\gamma \geqslant \beta$

$$\begin{aligned} &\implies && \alpha \times \gamma = \alpha \times \gamma \\ &\implies && \alpha \times (\beta + \gamma) = (\alpha \times \beta) + (\alpha \times \gamma) \end{aligned}$$

Left Distribution $(\alpha + \beta) \times \gamma = (\alpha \times \gamma) + (\beta \times \gamma)$

Case 1: $\alpha \geqslant \beta$

$$\begin{aligned} &\Longrightarrow && \alpha \times \gamma = \alpha \times \gamma \\ &\Longrightarrow && (\alpha + \beta) \times \gamma = (\alpha \times \gamma) + (\beta \times \gamma) \end{aligned}$$

Case 3: $\beta \geqslant \alpha$

$$\begin{aligned} &\Longrightarrow && \beta \times \gamma = \beta \times \gamma \\ &\Longrightarrow && (\alpha + \beta) \times \gamma = (\alpha \times \gamma) + (\beta \times \gamma) \end{aligned}$$

□

A.2 Matrix Semi-ring

This subsection explains and details how matrices with coefficients in a semi-ring can be used to construct semi-rings.

Lemma 3. *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$*, we define the tuple* $\mathbb{M} = (M, 0, 1, +, \times)$*, 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,*
- $+$ *defined by* $C = A + B$ *iff* $C_{ij} = A_{ij} + B_{ij}$*,*
- $\times$ *defined by* $C = A \times B$ *iff* $C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$*,*

is a strong semi-ring.

Proof. We prove that $\mathbb{M} = (M, 0, 1, +, \times)$ as defined respects the conditions of Definition 2. Let A,B,C be $n \times n$ matrices over S where $n \in \mathbb{N}$.

$(M, 0, 1, +)$ **is a commutative monoid** We first prove that $(M, +)$ is a commutative monoid by showing that it is associative, commutative, and has 0 as identity.

Associativity $(A+B)+C = A+(B+C)$ iff $((A+B)+C)_{ij} = (A+(B+C))_{ij}$ for all i, j.

$$\begin{aligned} ((A + B) + C)_{ij} &= (A + B)_{ij} + C_{ij} \\ &= (A_{ij} + B_{ij}) + C_{ij} \\ &= A_{ij} + (B_{ij} + C_{ij}) && \text{(by associativity of } +) \\ &= A_{ij} + (B + C)_{ij} \\ &= (A + (B + C))_{ij} \end{aligned}$$

Commutative Property $A + B = B + A$ iff $(A + B)_{ij} = (B + A)_{ij}$ for all i, j.

$$\begin{aligned} (A + B)_{ij} &= A_{ij} + B_{ij} \\ &= B_{ij} + A_{ij} && \text{(by commutativity of } +) \\ &= (B + A)_{ij} \end{aligned}$$

Identity element is 0 Let $A = 0$, then $A_{ij} = 0$ for all i, j, and 0 is the identity element iff $A_{ij} + B_{ij} = B_{ij}$ for all i, j

$$\begin{aligned}(A+B)_{ij} &= A_{ij} + B_{ij} \\ &= 0 + B_{ij} && \text{(by identity of } +) \\ &= B_{ij}\end{aligned}$$

$(M, 1, \times)$ **is a monoid** We now prove that $(M, \times)$ is a monoid by showing that it is associative and has 1 as identity.

Associativity $(A \times B) \times C = A \times (B \times C)$ iff $((A \times B) \times C)_{ij} = (A \times (B \times C))_{ij}$ for all i, j.

$$\begin{aligned}((A \times B) \times C)_{ij} &= (\sum_{k=1}^{n} A_{ik} \times B_{kj}) \times C \\ &= \sum_{l=1}^{n} (\sum_{k=1}^{n} A_{ik} \times B_{kj})_{il} \times C_{lj} \\ &= \sum_{l=1}^{n} \sum_{k=1}^{n} (A_{ik} \times B_{kl}) \times C_{lj} \\ &= \sum_{k=1}^{n} \sum_{l=1}^{n} A_{ik} \times (B_{kl} \times C_{lj}) && \text{(by assoc. of } \times) \\ &= \sum_{k=1}^{n} A_{ik} \times (\sum_{l=1}^{n} B_{il} \times C_{lj})_{kj} \\ &= A \times (\sum_{l=1}^{n} B_{il} \times C_{lj}) \\ &= (A \times (B \times C))_{ij}\end{aligned}$$

Identity element is 1 $A \times B = B$ and $B \times A = B$ where $A = 1$ iff $A_{ij} = 1$ for $i = j$ and $A_{ij} = 0$ otherwise.

$$\begin{aligned}(A \times B)_{ij} &= \sum_{k=1}^{n} A_{ik} \times B_{kj} \\ &= (A_{ii} \times B_{ij}) + \sum_{k=1, k \neq i}^{n} A_{ik} \times B_{kj} \\ &= (1 \times B_{ij}) + \sum_{k=1, k \neq i}^{n} 0 \times B_{kj} && \text{(by def. of 1)} \\ &= (1 \times B_{ij}) + \sum_{k=1, k \neq i}^{n} 0 && \text{(by annihilation prop. of 0)} \\ &= (1 \times B_{ij}) && \text{(by identity of } +) \\ &= B_{ij} && \text{(by identity of } \times)\end{aligned}$$

$$\begin{aligned}
(B \times A)_{ij} &= \sum_{k=1}^{n} B_{ik} \times A_{kj} \\
&= (B_{ij} \times A_{jj}) + \sum_{k=1,k\neq j}^{n} B_{ik} \times A_{kj} \\
&= (B_{ij} \times 1) + \sum_{k=1,k\neq j}^{n} B_{ik} \times 0 && \text{(by def. of 1)} \\
&= (B_{ij} \times 1) + \sum_{k=1,k\neq j}^{n} 0 && \text{(by annihilation prop. of 0)} \\
&= (B_{ij} \times 1) && \text{(by identity of +)} \\
&= B_{ij} && \text{(by identity of $\times$)}
\end{aligned}$$

0 **annihilates** M $A \times B = 0$ and $B \times A = 0$ where $A = 0$ iff $A_{ij} = 0$ for all i, j.

$$\begin{aligned}
(A \times B)_{ij} &= \sum_{k=1}^{n} A_{ik} \times B_{kj} \\
&= \sum_{k=1}^{n} 0 \times B_{kj} && \text{(by def. of 0)} \\
&= \sum_{k=1}^{n} 0 && \text{(by annihilation prop. of 0)} \\
&= 0
\end{aligned}$$

$$\begin{aligned}
(B \times A)_{ij} &= \sum_{k=1}^{n} B_{ik} \times A_{kj} \\
&= \sum_{k=1}^{n} B_{kj} \times 0 && \text{(by def. of 0)} \\
&= \sum_{k=1}^{n} 0 && \text{(by annihilation prop. of 0)} \\
&= 0
\end{aligned}$$

Distribution of multiplication over addition

Right Distribution $A \times (B + C) = (A \times B) + (A \times C)$ iff $(A \times (B + C))_{ij} = ((A \times B) + (A \times C))_{ij}$ for all i, j.

$$
\begin{aligned}
A \times (B + C))_{ij} &= \sum_{k=1}^{n} \big(A_{ik} \times (B_{kj} + C_{kj})\big) \\
&= \sum_{k=1}^{n} \big((A_{ik} \times B_{kj}) + (A_{ik} \times C_{kj})\big) \\
&\qquad\qquad \text{(by right distribution of } \times) \\
&= \sum_{k=1}^{n} (A_{ik} \times B_{kj}) + \sum_{k=1}^{n} (A_{ik} \times C_{kj}) \\
&= (A \times B)_{ij} + (A \times C)_{ij} \\
&= ((A \times B) + (A \times C))_{ij}
\end{aligned}
$$

Left Distribution $(A+B) \times C = (A \times C) + (B \times C)$ iff $((A+B) \times C)_{ij} = ((A \times C) + (B \times C))_{ij}$ for all i, j.

$$
\begin{aligned}
((A + B) \times C)_{ij} &= \sum_{k=1}^{n} \big((A_{ik} + B_{ik}) \times C_{kj}\big) \\
&= \sum_{k=1}^{n} \big((A_{ik} \times C_{kj}) + (B_{ik} \times C_{kj})\big) \\
&\qquad\qquad \text{(by left distribution of } \times) \\
&= \sum_{k=1}^{n} (A_{ik} \times C_{kj}) + \sum_{k=1}^{n} (B_{ik} \times C_{kj}) \\
&= (A \times C)_{ij} + (B \times C)_{ij} \\
&= ((A \times C) + (B \times C))_{ij}
\end{aligned}
$$

□

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

A.3 Choices Semi-ring

This subsection explains and details how functions into semi-ring coefficients can be used to construct semi-rings, and the interplay between this construction and the matrix semi-ring from the previous subsection.

Lemma 4. *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.

Proof. $(F, 0, \boxplus)$ **is a commutative monoid** We first prove that $(F, 0, \boxplus)$ is a commutative monoid by showing that it is associative, commutative, and has 0 as identity.
Associativity

$$\begin{aligned}((f \boxplus g) \boxplus h)(a) &= (f(a) + g(a)) + h(a) \\ &= f(a) + (g(a) + h(a)) && \text{(by assoc. of } +) \\ &= (f \boxplus (g \boxplus h))(a) && \text{(by def. of } \boxplus)\end{aligned}$$

Commutativity

$$\begin{aligned}(f \boxplus g)(a) &= f(a) + g(a) \\ &= g(a) + f(a) && \text{(by commutativity of } +) \\ &= (g \boxplus f)(a) && \text{(by def. of } \boxplus)\end{aligned}$$

Identity element is 0

$$\begin{aligned}(0 \boxplus f)(a) &= 0(a) + f(a) \\ &= 0 + f(a) && \text{(by def. of 0)} \\ &= f(a) && \text{(by identity prop of } +)\end{aligned}$$

$(F, 1, \boxtimes)$ **is a monoid** We now prove that $(F, 1, \boxtimes)$ is a monoid by showing that it is associative and has 1 as identity.
Associativity

$$\begin{aligned}((f \boxtimes g) \boxtimes h)(a) &= (f(a) \times g(a)) \times h(a) \\ &= f(a) \times (g(a) \times h(a)) && \text{(by assoc. of } \times) \\ &= (f \boxtimes (g \boxtimes h))(a) && \text{(by def. of } \boxtimes)\end{aligned}$$

Identity element is 1

$$\begin{aligned}(1 \boxtimes f)(a) &= 1(a) \times f(a) \\ &= 1 \times f(a) && \text{(by def. of 1)} \\ &= f(a) && \text{(by identity prop of } \times)\end{aligned}$$

Distribution of multiplication over addition We conclude by proving that $\boxtimes$ distributes over $\boxplus$.
Right Distribution

$$\begin{aligned}(f \boxtimes (g \boxplus h))(a) &= f(a) \times (g(a) + h(a)) \\ &= (f(a) \times g(a)) + (f(a) \times h(a)) \\ &\qquad \text{(by right distribution of } \times) \\ &= ((f \boxtimes g) \boxplus (f \boxtimes h))(a)\end{aligned}$$

Left Distribution

$$\begin{aligned}((f \boxplus g) \boxtimes h)(a) &= (f(a) + g(a)) \times h(a) \\ &= (f(a) \times h(a)) + (g(a) \times h(a)) \\ &\qquad \text{(by left distribution of } \times) \\ &= ((f \boxtimes h) \boxplus (g \boxtimes h))(a)\end{aligned}$$

0 **annihilates** F

$$\begin{aligned}(0 \boxtimes f)(a) &= 0(a) \times f(a) \\ &= 0 \times f(a) && \text{(by def. of 0)} \\ &= 0 && \text{(by annihilation prop of 0)}\end{aligned}$$

$$\begin{aligned}(f \boxtimes 0)(a) &= f(a) \times 0(a) \\ &= f(a) \times 0 && \text{(by def. of 0)} \\ &= 0 && \text{(by annihilation prop of 0)}\end{aligned}$$

□

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

Definition 3. *We say two semi-rings* $\mathbb{S} = (S, 0, 1, +, \times)$ *and* $\mathbb{T} = (T, 0, 1, \boxplus, \boxtimes)$ *are* isomorphic *and write* $\mathbb{S} \cong \mathbb{T}$ *if there exists* $g : S \to T$ *such that*

- g *is a bijection,*
- $g(0) = 0$*,*
- $g(1) = 1$*,*
- $g(s_1 + s_2) = g(s_1) \boxplus g(s_2)$ *for all* $s_1, s_2 \in S$
- $g(s_1 \times s_2) = g(s_1) \boxtimes g(s_2)$ *for all* $s_1, s_2 \in S$

For simplicity, we write $g : \mathbb{S} \to \mathbb{T}$ *for such morphisms.*

Lemma 5. *For all set* A *and strong semi-ring* $\mathbb{S}$*,* $\mathbb{M}(A \to \mathbb{S}) \cong A \to \mathbb{M}(\mathbb{S})$*.*

Proof. First, observe that by Lemmas 3 and 4, both $A \to \mathbb{M}(\mathbb{S})$ and $\mathbb{M}(A \to \mathbb{S})$ are strong semi-rings, and we write 0_f (resp. 0_M) and 1_f (resp. 1_M) for the 0 and 1 elements of $A \to \mathbb{M}(\mathbb{S})$ (resp. of $\mathbb{M}(A \to \mathbb{S})$). Now we have to prove that we can construct a bijection $g : M(A \to S) \to (A \to M(S))$ that respects the conditions of Definition 3.

We define g and g^{-1} at the same time, then show that they are indeed inverses:

$g : M(A \to S) \to (A \to M(S))$ Given $M \in M(A \to S)$ of size $n \times n$, we let $g(M) \in A \to M(S)$ be the function that maps $a \in A$ to M where the same argument a has been applied to the functions $f_{1,1}, \ldots, f_{n,n}$. Graphically:

$$g(M)a = g(\begin{pmatrix} M_{1,1} & \dots & M_{1,n} \\ \vdots & \ddots & \vdots \\ M_{n,1} & \dots & M_{n,n} \end{pmatrix})a = \begin{pmatrix} M_{1,1}a & \dots & M_{1,n}a \\ \vdots & \ddots & \vdots \\ M_{n,1}a & \dots & M_{n,n}a \end{pmatrix}$$

Below, we write f_M for $g(M)$.

$g^{-1} : (A \to M(S)) \to M(A \to S)$ Given $f \in A \to M(S)$, we define $g^{-1}(f) \in M(A \to S)$ to be the matrix of size $n \times n$, for $n \times n$ the size of the matrix returned by f, such that $(g^{-1}(f))_{i,j}$ is the function that maps $a \in A$ to $(f(a))_{i,j}$ for all i, j. Graphically:

$$g^{-1}(f)a = \begin{pmatrix} (fa)_{1,1} & \dots & (fa)_{1,n} \\ \vdots & \ddots & \vdots \\ (fa)_{n,1} & \dots & (fa)_{n,n} \end{pmatrix}$$

Below, we write M_f for $g^{-1}(f)$.

g is a bijection We first prove that $g \circ g^{-1} = g^{-1} \circ g = \mathrm{id}$.

$(g^{-1} \circ g)(M) = M$

$$\begin{aligned} (g^{-1} \circ g)(M) &= g^{-1}(g(M)) \\ &= g^{-1}(f_M) && \text{(where } (f_M(a))_{ij} = M_{ij}(a)\text{)} \\ &= M \end{aligned}$$

$(g \circ g^{-1})(f) = f$

$$\begin{aligned} (g \circ g^{-1})(f) &= g(g^{-1}(f)) \\ &= g(M_f) && \text{(where } (M_f)_{ij}a = (f(a))_{ij}\text{)} \\ &= f \end{aligned}$$

$g(0_M) = 0_f$ Let $f = g(0_M)$, then $f = 0_f$ iff $f(a)_{ij} = 0_{\mathbb{S}}$ for all i, j.

$$\begin{aligned} f(a)_{ij} &= (0_M)_{ij}(a) \\ &= 0_f(a) && \text{(by def. of } 0_M\text{)} \\ &= 0_{\mathbb{S}} && \text{(by def. of } 0_f\text{)} \end{aligned}$$

$g(1_M) = 1_f$ Let $f = g(1_M)$, then $f = 1_f$ iff $f(a)_{ij} = 1_{\mathbb{S}}$ for all $i = j$ and $f(a)_{ij} = 0_{\mathbb{S}}$ otherwise.

Case 1: $i = j$

$$\begin{aligned} f(a)_{ij} &= (1_M)_{ij}(a) \\ &= 1_f(a) && \text{(by def. of } 1_M\text{)} \\ &= 1_{\mathbb{S}} && \text{(by def. of } 1_f\text{)} \end{aligned}$$

Case 2: $i \neq j$

$$\begin{aligned} f(a)_{ij} &= (1_M)_{ij}(a) \\ &= 0_f(a) && \text{(by def. of } 1_M\text{)} \\ &= 0_{\mathbb{S}} && \text{(by def. of } 0_f\text{)} \end{aligned}$$

$g(M_1 + M_2) = g(M_1) + g(M_2)$

$$\begin{aligned}
& g(M_1 + M_2) = g(M_1) + g(M_2) \\
\iff & f_{M_1+M_2} = f_{M_1} + f_{M_2} \\
\iff & f_{M_1+M_2}(a) = (f_{M_1} + f_{M_2})(a) \\
\iff & f_{M_1+M_2}(a) = f_{M_1}(a) + f_{M_2}(a) \\
\iff & (f_{M_1+M_2}(a))_{ij} = (f_{M_1}(a) + f_{M_2}(a))_{ij} \\
\iff & (M_1 + M_2)_{ij}(a) = (M_1)_{ij}(a) + (M_2)_{ij}(a) \qquad \text{(by assoc. of } +\text{)}
\end{aligned}$$

$g(M_1 \times M_2) = g(M_1) \times g(M_2)$

$$\begin{aligned}
& g(M_1 \times M_2) = g(M_1) \times g(M_2) \\
\iff & f_{M_1\times M_2} = f_{M_1} \times f_{M_2} \\
\iff & f_{M_1\times M_2}(a) = (f_{M_1} \times f_{M_2})(a) \\
\iff & f_{M_1\times M_2}(a) = (f_{M_1})(a) \times (f_{M_2})(a) \\
\iff & (f_{M_1\times M_2}(a))_{ij} = ((f_{M_1})(a) \times (f_{M_2})(a))_{ij} \\
\iff & (\sum_{k=1}^{n} (M_1)_{ik} \times (M_2)_{kj})(a) = \sum_{k=1}^{n} (M_1)_{ik}(a) \times (M_2)_{kj}(a) \\
& \qquad\qquad \text{(by assoc. of } + \text{ and } \times\text{)}
\end{aligned}$$

□

A.4 Partiality

In our improvement of the analysis, we add an ∞ element to the mwp-semi-ring, but reason abstractly below with an arbitrary semi-ring and a $\bot$ element.

Lemma 6. *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$ *and an element* $\bot \notin S$, $\mathbb{S}^\bot = (S \cup \{\bot\}, 0, 1, +^\bot, \times^\bot)$ *with, for all* $a, b \in S \cup \{\bot\}$,

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

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

is a semi-ring.

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

A good intuition on this construction comes from partial functions. Indeed, we can define $A \rightharpoonup \mathbb{S}$ as the semi-ring of partial functions from A to $\mathbb{S}$, i.e. of functions from A to $\mathbb{S}^\bot$. Furthermore, if we identify a matrix in $\mathbb{M}(\mathbb{S}^\bot)$ where at least a coefficient is $\bot$ with the matrix $\bot$, then we get that $\mathbb{M}(A \rightharpoonup \mathbb{S}) \cong A \rightharpoonup \mathbb{M}(\mathbb{S})$. However, note that none of those semi-rings are strong.