mwp-Analysis Improvement and Implementation
Realizing Implicit Computational Complexity

C. Aubert[1] T. Rubiano[2] N. Rusch[1] T. Seiller[3]

[1] Augusta University
[2] Paris 13 University, LIPN – UMR 7030
[3] CNRS, LIPN – UMR 7030

FSCD, Haïfa, Israël, August 4th 2022

Summary

- Our research focuses on static program analysis of imperative programs
- Using a technique inspired by implicit computational complexity
- This talk will demonstrate how to use this technique to analyze variable value growth
- We have modified, extended and made this technique practical with a working protype

Implicit Computational Complexity (ICC) theory

Definition by Romain Péchoux[‡]:

Let L be a programming language, C a complexity class, and $[\![p]\!]$ the function computed by program p.

Find a restriction $R \subseteq L$, such that the following equality holds:

$$\{[\![p]\!] \mid p \in R\} = C$$

The variables L, C and R are the parameters that vary greatly between different ICC systems.

[‡] Péchoux, Romain. 2020. "Complexité implicite: bilan et perspectives." Habilitation à Diriger des Recherches (HDR). Université de Lorraine.

“A Flow Calculus of MWP-Bounds for Complexity Analysis"

Neil D. Jones and Lars Kristiansen (2009)

“Program complexity analysis seems naturally to decompose into two parts a termination analysis and a data size analysis. [. . .] This paper considers data size analysis. [. . .] The analysis aims, given a program, to find out whether its variables have acceptable growth rates."

Theoretical foundation: mwp analysis

- 2008 paper by Neil Jones and Lars Kristiansen:
  "A Flow Calculus of mwp-Bounds for Complexity Analysis"
- This technique is related in spirit to abstract interpretation but differs in that it bounds transitions between states (commands), instead of states
- Also related to size-change principle, quasi-interpretations.
- "Careful and detailed analysis of the relationship between resource requirements of computation and the way data might flow during computation"

mwp-Analysis: Program Syntax

Variable	$\mathtt{X}_1$ | $\mathtt{X}_2$ | $\mathtt{X}_3$ | ...
Expression	`X` | `e + e` | `e * e`
Boolean Exp.	`e = e`, `e < e`, etc.
Commands	`skip` | `X := e` | `C;C` | `loop X {C}` | `if b then C else C` | `while b do {C}`

mwp Calculus

Analyze variable value growth by:

1. Assigning a vector to each variable
2. Collecting vectors into a matrix
3. Applying derivation rules to evaluate program complexity

Flows represent quantitative information of variables on each other:

0	no dependency
$\underline{m}$	maximal
$\underline{w}$	weak polynomial
$\underline{p}$	polynomial

Definition

An mwp-bound is a number-theoretic expression of form $\max(\vec{(x)}, \text{poly1}(\vec{y})) + \text{poly2}(\vec{z})$.

An mwp-bound W over the variables $x_1, \dots, x_n$ is represented as a column vector

$$\begin{pmatrix} W(x1) \\ W(x_2) \\ \vdots \\ W(x_n) \end{pmatrix}$$

E.g., if $n = 5$, an mwp-bound of the form $\max(x_5, \text{poly1}(x_2, x_4)) + \text{poly2}(x_1)$ is represented by the vector

$$\begin{pmatrix} p \\ w \\ 0 \\ w \\ m \end{pmatrix}.$$

An $n \times n$ matrix consists of n column vectors $(V_1, \dots, V_n)$, and thus an $n \times n$ matrix over $\{0, m, w, p\}$ will represent a collection of n mwp-bounds.

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

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

(a) Rules for assigning vectors to expressions

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

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

$$\forall i, M^*_{ii} = m\ \frac{\vdash_{\mathrm{JK}} \mathtt{C} : M}{\vdash_{\mathrm{JK}} \mathtt{loop\ X1\ \{C\}} : M^* \oplus \{{}^{p}_{\mathtt{l}} \rightarrow j \mid \exists i, M^*_{ij} = p\}}\ \mathrm{L}$$

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

(b) Rules for assigning matrices to commands

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

mwp-Analysis: Derivation Example

Let's analyze this program: `loop X3 {X2 = X1 + X2}`

Example

```
loop X3 {X2 = X1 + X2}
```

$$\texttt{X1} : \begin{pmatrix} m \\ 0 \\ 0 \end{pmatrix} \qquad \texttt{X2} : \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix} \tag{E1}$$

Example

```
loop X3 {X2 = X1 + X2}
```

$$\texttt{X1 + X2} : \begin{pmatrix} p \\ m \\ 0 \end{pmatrix} \tag{E3}$$

Example

```
loop X3 {X2 = X1 + X2}
```

$$\texttt{X2 = X1 + X2}: \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix} \tag{A}$$

Example

```
loop X3 {X2 = X1 + X2}
```

$$\texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & p & m \end{pmatrix} \tag{L}$$

Nondeterminism

The body X2 = X1 + X2 of the loop command admits in fact three different derivations, obtained by applying A to one of the following derivations π_0, π_1, π_2:

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

This is because different mwp-bounds may be numerically equal to the same polynomial. e.g.,max$(0, x_1 + x_2)$, max$(x_1, 0) + x_2$, and max$(x_2, 0) + x_1$ (represented by the above bounds) are all numerically equal to $x_1 + x_2$.

Nondeterminism

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:

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

Similarly, the L rule cannot be applied to extend π_2 because of a diagonal w coefficient.

Open questions

The original mwp-analysis was theoretical

There were open questions:

1. Can it be applied to richer languages?
2. How powerful and convenient is this technique? [Can it be implemented?]

Implementing mwp analysis

Two modifications were needed to enable implementation:

1. Changing handing of failure: introduced a new flow ∞ to represent failure locally

$$0, m, w, p, \infty$$

- Enables completing every derivation
- Provides fine-grained information on source of failure on programs that do not have polynomially bounded growth

Implementing mwp analysis

Two significant modifications were needed to enable implementation:

2. Non-determinism of original analysis was impractical:
replaced by deterministic derivation rules

$$\texttt{X2 = X1 + X1} : \begin{pmatrix} m & w\delta(0,0)+p\delta(1,0)+w\delta(2,0) \\ 0 & 0 \end{pmatrix}$$

- All derivations are represented in the same matrix

Nondeterminism: impractibility

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.

- Computing all the matrices one after the other leads to time explosion.
- 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 would result in 9 matrices of size 6×6, i.e., 324 "scalars".
- Using the isomorphism $A \to \mathbb{M}(\Omega) \cong \mathbb{M}(A \to \Omega)$ allows for a compact representation avoiding redundancies (if a coefficient depends on only one choice, represented as 3 elements of Ω; if independent, represented as a single element): the above program involving 6 variables with 3 choices would now be assigned a unique 6×6 matrix that requires 66 "scalars" instead.

Further optimisations: polynomials

We represent functions $A \to \{0, m, w, p, \infty\}$ (think $A = \{0,1,2\}^n$) as "polynomials":

- we define basic functions $\delta(i,j)$ by $\delta(i,j)(a_n, a_{n-1}, \ldots, a_1) = m$ if $a_j = i$, and $\delta(i,j)(a_n, a_{n-1}, \ldots, a_1) = 0$ otherwise.
- any function $A \to \{0, m, w, p, \infty\}$ is represented as a linear combination of "monomials" (products of basic functions): $\sum_k \alpha_k \left(\prod \delta(i^\ell, j^\ell)\right)$.

Using techniques akin to Gröbner bases, we can implement efficient computation of algebraic operations. Multiplying by a monomial preserves the (well-chosen) order (of non-zero elements), which can be used to implement multiplication efficiently: $P_1 P_2$ is computed by producing the collection of $m_i P_2$ for m_i monomials in P_1, then fusion the ordered list thus obtained.

Deterministic system

We thus replace the original mwp rules by the following deterministic system.

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

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

(a) Rules for assigning vectors to expressions

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

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

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

(b) Rules for assigning matrices to commands

Figure 2: Deterministic improved flow analysis rules

The new system now assigns to **loop** X3 {X2 = X1 + X2} the unique matrix

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

where we observe that

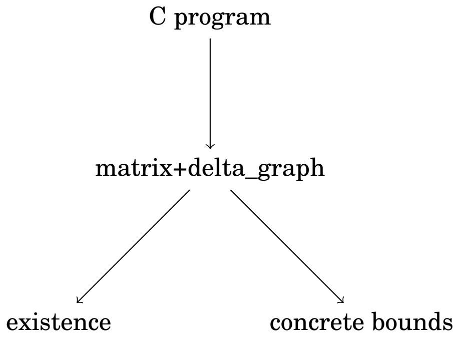
1. only one choice, one assignment, 0, gives a matrix without ∞ coefficient, corresponding to the fact that, in the original system, only π_0 could be used to complete the proof,
2. the choice impacts the matrix locally, the coefficients being mostly the same, independently from the choice,
3. the influence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

Separating the problem

This representation allows us to separate the computation of mwp-bounds in two distinct problems:

- Decide the existence of a bound;
- Compute concrete bounds.

The workflow is the following:



Key ingredient: Compositionality

We integrate function calls as follows. Let f be a function defined independently (assuming it has only one output value). Analysing the code defining f produces a matrix M which we use to produce mwp-certificates as follows: we find the assignments (choices) for which no ∞ coefficients appear, and project the resulting matrices to only keep the vector representing the corresponding mwp bound of the output value w.r.t. the input values of f. We thus obtain k possible certificates $M_f^1, M_f^2, \ldots, M_f^k$.

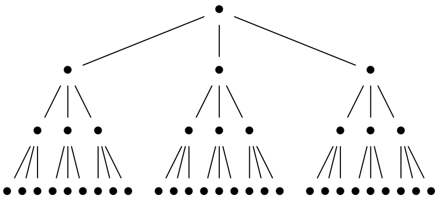
We then add the following rule to assign a mwp flow to functions calls to f.

$$\frac{}{\vdash \mathrm{Xi} = \mathrm{F}(\mathrm{X1}, \ldots, \mathrm{Xn}) : 1 \xleftarrow{\mathrm{i}} ((M_f^1)\delta(0,c) \oplus \cdots \oplus (M_f^k)\delta(0,c)\delta(k,c))}\ \mathbf{F}$$

We also explain how this can be used to analyse recursive calls.

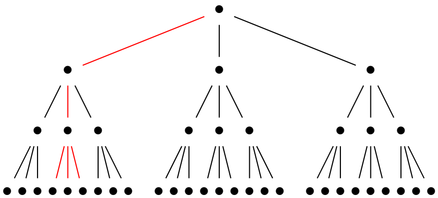
Further optimisations: delta_graphs

We collect during the analysis of the monomials with infinite coefficients. Note that these coefficients can be thought of as basic open (cylindrical) sets: e.g.,$\delta(0,1)\delta(2,2)$. We use a specific data structure called delta_graphs that manages this collection of polynomials and simplifies it.



Further optimisations: delta_graphs

We collect during the analysis of the monomials with infinite coefficients. Note that these coefficients can be thought of as basic open (cylindrical) sets: e.g.,$\delta(0,1)\delta(2,2)$. We use a specific data structure called delta_graphs that manages this collection of polynomials and simplifies it.



Further optimisations: delta_graphs

We collect during the analysis of the monomials with infinite coefficients. Note that these coefficients can be thought of as basic open (cylindrical) sets: e.g.,. We use a specific data structure called delta_graphs that manages this collection of polynomials and simplifies it.

Example

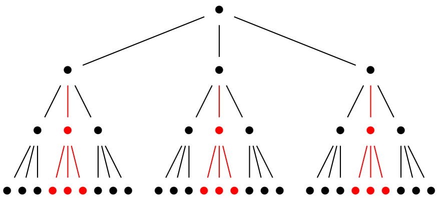
If one has infinite coeficients for the monomials $\delta(0,1)\delta(2,2)$, $\delta(2,2)\delta(0,3)$, $\delta(2,2)\delta(1,3)$, and $\delta(2,2)\delta(2,3)$, then it is equivalent to having infinite coefficients for $\delta(0,1)\delta(2,2)$ and $\delta(2,2)$, which in turn is equivalent to having an infinite coefficient for the monomial $\delta(2,2)$.

The existence of an mwp-bound then becomes equivalent to the question: is the delta_graph different from the graph containing only the monomial 1?

Further optimisations: delta_iterator

When computing the concrete bounds, we use a specific iterator using the delta_graph that produces only values not covered by the monomials for which an infinite coefficient appears.

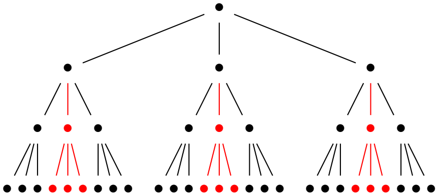
e.g.,if the delta_graph contains the single monomial $\delta(2,2)$, the delta_iterator for size 3 lists will produce $(0,0,0)$ initially, then the following values: $(0,0,1),(0,0,2),(0,1,0),(0,1,1),(0,1,2),(1,0,0),\ldots$.



Further optimisations: delta_iterator

When computing the concrete bounds, we use a specific iterator using the delta_graph that produces only values not covered by the monomials for which an infinite coefficient appears.

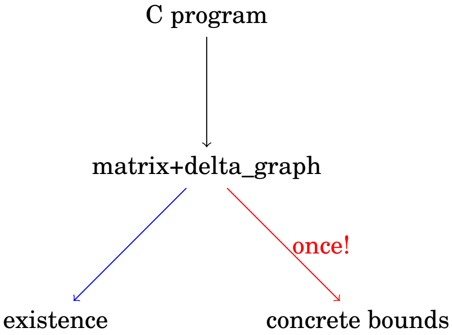
e.g.,if the delta_graph contains the single monomial $\delta(2,2)$, the delta_iterator for size 3 lists will produce $(0,0,0)$ initially, then the following values: $(0,0,1),(0,0,2),(0,1,0),(0,1,1),(0,1,2),(1,0,0),\dots$.



Resolving practical inefficiencies

Computing all mwp certificates is still costly. This issue is however resolved by the following strategies:

1. decoupling computation by using delta graph
2. compositionality enables reusing results



Resolving practical inefficiencies

Compositionality of analysis enables computing result once then reusing the result it in the future

- Analysis can be performed on parts of source code
- It is possible to analyze a function, then save the result
- Previously analyzed result can be reused at next execution
- Expensive computation needs to be carried out once

Prototype: pymwp

- Implementation of mwp-analysis on a subset of C99, in Python
- Open source: `github.com/statycc/pymwp`
- If analysis succeeds:
  - program uses at most a polynomial amount of space
  - if it terminates, it will do so in polynomial time
- If variable grows too much, polynomial bound cannot be guaranteed
- Still work to be done.

Not the beginning...

This work follows a previous implementation of similar techniques by Moyen, Rubiano, and Seiller.

1. Loop optimization: using dependency analysis borrowed from ICC to detect inefficiencies in loops and to automatically unroll them to optimize the code. This was implemented on C (`https://github.com/statycc/LQICM_On_C_Toy_Parser`), as well as on (an old version of) LLVM Intermediate Representation (`https://github.com/ThomasRuby/LQICM_pass`).

Not the beginning...

This work follows a previous implementation of similar techniques by Moyen, Rubiano, and Seiller.

1. Loop optimization: using dependency analysis borrowed from ICC to detect inefficiencies in loops and to automatically unroll them to optimize the code. This was implemented on *C* (`https://github.com/statycc/LQICM_On_C_Toy_Parser`), as well as on (an old version of) LLVM Intermediate Representation (`https://github.com/ThomasRuby/LQICM_pass`).

Coming back to the original questions.

1. Can it be applied to richer languages?
2. Can it be implemented? Yes!

Not the beginning...

This work follows a previous implementation of similar techniques by Moyen, Rubiano, and Seiller.

1. Loop optimization: using dependency analysis borrowed from ICC to detect inefficiencies in loops and to automatically unroll them to optimize the code. This was implemented on *C* (`https://github.com/statycc/LQICM_On_C_Toy_Parser`), as well as on (an old version of) LLVM Intermediate Representation (`https://github.com/ThomasRuby/LQICM_pass`).

Coming back to the original questions.

1. Can it be applied to richer languages? Sure, but more interestingly: ICC techniques should be used on Intermediate Representation.

2. Can it be implemented? Yes!

... nor the end.

Future directions for complexity analysis include compiler integration:

1. Leverage intermediate representation
2. Static single assignment (SSA) form for efficiency and fine-grained information
3. Certified complexity analysis to be able to integrate with CompCert

More generally, flow-analyses open a rich new territory to be explored:

1. Automatic loop optimisation (previous work)
2. Complexity analysis (this work, and extensions)
3. Automatic loop parallelisation (available draft)
4. Floating-point analysis to track growth of error in precision (project)
5. ...