The Design and Implementation of a High-Performance Polynomial System Solver

Alexander Brandt

PhD Research Topics Survey/Proposal

Department of Computer Science
University of Western Ontario

December 22, 2021

Solving Systems of Equations

Find values of x, y, z which satisfy $$F = \begin{cases} a(x,y,z) = 0 \\ b(x,y,z) = 0 \\ c(x,y,z) = 0 \end{cases}$$

- Solving systems of equations is a fundamental problem in scientific computing
- Numerical methods are very efficient and useful in practice, but only find approximate solutions as floating point numbers
  - ↳ Newton's method, Homotopy methods, Gradient descent
- **Symbolic methods** to find exact solutions are required in robotics, celestial mechanics, cryptography, signal processing [18]
  - ↳ Particularly used to find a complete description of all solutions

Solving a Linear System of Equations

$$\begin{cases} x + 3y - 2z = 6 \\ 3x + 5y + 6z = 7 \\ 2x + 4y + 3z = 8 \end{cases}$$

Step 1: triangularization

(a) by *elimination of variables*:

$$\begin{cases} x + 3y - 2z = 6 \\ 3x + 5y + 6z = 7 \\ 2x + 4y + 3z = 8 \end{cases} \xrightarrow[\text{substitute } x]{\text{solve for } x} \begin{cases} x = 5 - 3y + 2z \\ -4y + 12z = -8 \\ -2y + 7z = -2 \end{cases} \xrightarrow[\text{substitute } y]{\text{solve for } y} \begin{cases} x = 5 + 2z - 3y \\ y = 2 + 3z \\ z = 2 \end{cases}$$

(b) by *Gaussian elimination*:

$$\left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 3 & 5 & 6 & 7 \\ 2 & 4 & 3 & 8 \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 0 & 1 & -3 & 2 \\ 0 & -2 & 7 & -2 \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 0 & 1 & -3 & 2 \\ 0 & 0 & 1 & 2 \end{array}\right]$$

Step 2: back-substitution to find particular values for x, y, z

Solving a Non-Linear System of Equations

Via **Gröbner Basis** we can "solve" a non-linear system

$$\begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \\ x + y + z^2 = 1 \end{cases} \implies \begin{cases} x + y + z^2 = 1 \\ (y + z - 1)(y - z) = 0 \\ z^2 \left(z^2 + 2y - 1\right) = 0 \\ z^2 \left(z^2 + 2z - 1\right)(z - 1)^2 = 0 \end{cases}$$

"Solving" a system is not just about finding particular values, rather:

> *"find a description of the solutions from which we can easily extract relevant data"*

Why?

- A **positive-dimensional system** has *infinitely many solutions*
- *Underdetermined* linear systems, and most non-linear systems

Decomposing a Non-Linear System

Many ways to "solve" a system

$$\begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \\ x + y + z^2 = 1 \end{cases} \xrightarrow{\text{Gröbner Basis}} \begin{cases} x + y + z^2 = 1 \\ (y + z - 1)(y - z) = 0 \\ z^2\left(z^2 + 2y - 1\right) = 0 \\ z^2\left(z^2 + 2z - 1\right)(z - 1)^2 = 0 \end{cases}$$

$\Downarrow$ Triangular Decomposition

$$\begin{cases} x - z = 0 \\ y - z = 0 \\ z^2 + 2z - 1 = 0 \end{cases}, \quad \begin{cases} x = 0 \\ y = 0 \\ z - 1 = 0 \end{cases}, \quad \begin{cases} x = 0 \\ y - 1 = 0 \\ z = 0 \end{cases}, \quad \begin{cases} x - 1 = 0 \\ y = 0 \\ z = 0 \end{cases}$$

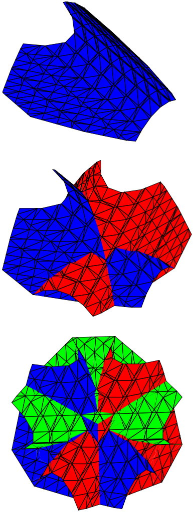
Both solutions are equivalent (via a union).

- by using triangular decomposition, **multiple components** are found, suggesting possible **component-level parallelism**

Outline

Incremental Decomposition of a Non-Linear System

Intersect one equation at a time with the current solution set



$$F = \begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \\ x + y + z^2 = 1 \end{cases}$$

$$\varnothing$$

$$F[1] \quad \downarrow$$

$$\left\{ x^2 + y + z = 1 \right\}$$

$$F[2] \quad \downarrow$$

$$\left\{ \begin{array}{r} x + y^2 + z = 1 \\ y^4 + (2z-2)y^2 + y + (z^2 - z) = 0 \end{array} \right\}$$

$$F[3] \qquad \swarrow \qquad \swarrow \qquad \searrow \qquad \searrow$$

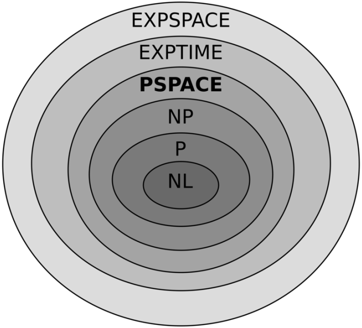
$$\begin{cases} x - z = 0 \\ y - z = 0 \\ z^2 + 2z - 1 = 0 \end{cases}, \quad \begin{cases} x = 0 \\ y = 0 \\ z - 1 = 0 \end{cases}, \quad \begin{cases} x = 0 \\ y - 1 = 0 \\ z = 0 \end{cases}, \quad \begin{cases} x - 1 = 0 \\ y = 0 \\ z = 0 \end{cases}$$

Solving polynomial systems symbolically is *hard*

- Algorithms are at least singly exponential $\mathcal{O}(d^n)$
  ↳ At least in $\mathcal{PSPACE}$ but up to $\mathcal{EXPSPACE}$-complete [17, Ch. 21]

- Algorithms require complex code and vast dependencies: arbitrary-precision integers, GCDs, factorization, linear algebra

- **Intermediate expression swell**



Motivations and Challenges

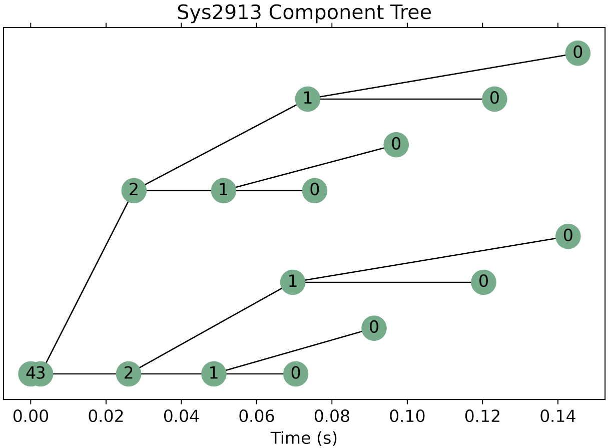
Motivations:

- Solving symbolically is difficult but still desirable in many fields
- Algorithmic development [7] has come a long way; must now focus on implementation techniques, making the most of modern hardware
  - ↳ Multicore processors, cache hierarchy
  - ↳ Must apply **parallel computing** and **data locality**

Challenges:

- Study application of high-performance techniques to high-level geometric algorithms
- Potential **parallelism is problem-dependent** and *not* algorithmic
  - ↳ Geometry may or may not split into different components
  - ↳ Finding splittings is as difficulty as solving the problem
- Study how software design can manage maintainability and usability of highly complex mathematical code

Unbalanced and Irregular Parallelism



- More parallelism exposed as more components found,
- Work unbalanced between branches; this is **irregular parallelism**
- Mechanism needed for **adaptive, dynamic parallelism**

Previous Works

- Long history of theoretical and algorithmic development in triangular decomposition [3, 5, 7–9, 22, 26, 27]

- Parallelization of high-level algebraic and geometric algorithms was more common roughly 30 years ago
  - ↳ Such as in Gröbner Bases [2, 6, 15] and CAD [24]

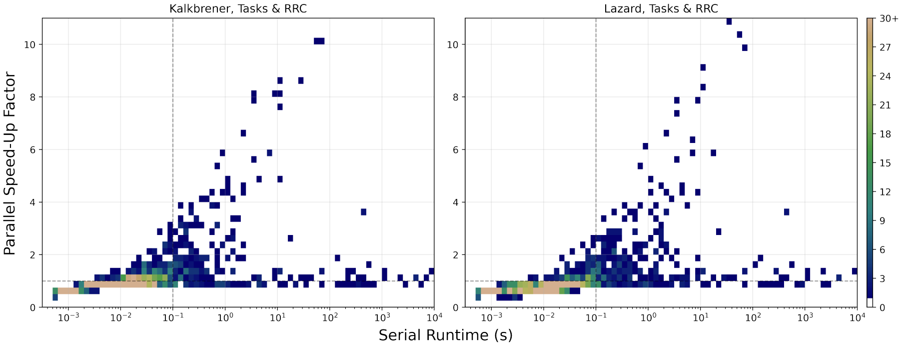
- Recent parallelism of *low-level* routines with *regular parallelism*:
  - ↳ Polynomial arithmetic [16, 20]
  - ↳ Modular methods for GCDs and Factorization [19, 21]

- Recently, high-level algorithms, often with *irregular parallelism*, have seen little progress in research or implementation
  - ↳ The normalization algorithm of [4] finds components serially, then processes each component with a simple parallel map
  - ↳ Early work on parallel triangular decomposition was limited by symmetric multi-processing and inter-process communication [23]

Objectives

1. Investigate and evaluate **component-level parallelism** and other high-performance techniques for triangular decompositions
2. Examine the composition of parallelism between high-level and low-level algorithms in symbolic computation
3. Re-imagine *dynamic evaluation* in the context of triangular decomposition
4. Study how software design can be used to improve the maintainability and usability of the resulting highly optimized and complex code

Preliminary Results



1. High-performance, parallel triangular decomposition in C/C++ with multiple simultaneous levels of parallelism
2. A library for composable and cooperative parallel programming with support for **parallel patterns**
3. An object-oriented class hierarchy encoding the algebraic hierarchy provides compile-time type safety and mathematical correctness

Outline

Polynomial Notations

- Let $\mathbb{K}$ be a perfect field (e.g. $\mathbb{Q}$ or $\mathbb{C}$) and $\overline{\mathbb{K}}$ its algebraic closure
- Let $\mathbb{K}[\underline{X}]$ be the set of multivariate polynomials (a *polynomial ring*) with n ordered variables, $\underline{X} = X_1 < \cdots < X_n$.
- For $p \in \mathbb{K}[\underline{X}]$:
  - ↳ the main variable of p is the maximum variable with positive degree
  - ↳ the initial of p is the leading coeff. of p with respect to its main variable
  - ↳ the tail of p is the terms leftover after setting its initial to 0

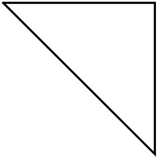
$$(2y + ba)x^2 + (by)x + a^2 \quad \in \ \mathbb{Q}[b < a < y < x]$$

- Any set of polynomials $F \subset \mathbb{K}[\underline{X}]$ can form a **system of equations** by setting $f = 0$ for each $f \in F$.
- The **zero set** of F is an **algebraic variety**—the geometric representation of its solutions
  - ↳ $V(F) = \left\{(a_1, \ldots, a_n) \in \overline{\mathbb{K}}^n \mid f(a_1, \ldots, a_n) = 0, \ \forall f \in F\right\}$

Triangular Sets and Regular Chains

A **triangular set** $T \subset \mathbb{K}[\underline{X}]$ is a collection of polynomials with pairwise different main variables

$$T = \left\{ \begin{array}{l} T_v = h\,v^d + \mathrm{tail}(T_v) \\ T_v^- = \{ \quad \} \end{array} \right\}$$



$$\subset \mathbb{K}[\underline{X}]$$

Example:

$$T = \left\{ \begin{array}{r} (2y + ba)x - by + a^2 \\ 2y^2 - by - a^2 \\ a + b \end{array} \right\} \subset \mathbb{Q}[b < a < y < x]$$

A **regular chain** is a triangular set if:

(i) T_v^- is a regular chain, and

(ii) initial of T_v (h) is **regular** with respect to T_v^-

In triangular decomposition, **every component is a regular chain**

Regularity: Not all triangular sets are regular chains

$$T_1 = \begin{cases} yx - 1 = 0 \\ y = 0 \\ z - 1 = 0 \end{cases}$$

- This set is inconsistent; there are no solutions
- Back-substituting $y = 0$ into $yx - 1 = 0$ yields $-1 = 0$

$$T_2 = \begin{cases} (y+1)x^2 - x = 0 \\ y^2 - 1 = 0 \\ z - 1 = 0 \end{cases}$$

- y has two solutions: $y^2 - 1 = (y+1)(y-1)$
- For $y = -1$, x has 1 solution
- For $y = 1$, x has 2 solutions

A polynomial is **regular** (*modulo* a regular chain) if it is neither:

(i) zero (e.g. y in T_1), nor

(ii) a *zero-divisor* (e.g. $(y+1)$ in T_2)

The foundation of splitting: regularity testing

To intersect a polynomial with an existing regular chain, it must have a regular initial, regularizing finds splittings via a **case discussion**

- either the initial is regular, or it is not regular

$$f = (y+1)x^2 - x$$

$$T = \begin{cases} y^2 - 1 = 0 \\ z - 1 = 0 \end{cases}$$

$$\xrightarrow{y+1=0} \quad T_1 = \begin{cases} y + 1 = 0 \\ z - 1 = 0 \end{cases} \quad \xrightarrow{f = x} \quad T_3 = \begin{cases} x = 0 \\ y + 1 = 0 \\ z - 1 = 0 \end{cases}$$

$$\xrightarrow{y+1 \neq 0} \quad T_2 = \begin{cases} y - 1 = 0 \\ z - 1 = 0 \end{cases} \quad \xrightarrow{f = 2x^2 - x} \quad T_4 = \begin{cases} 2x^2 - x = 0 \\ y - 1 = 0 \\ z - 1 = 0 \end{cases}$$

This actually forms a **direct product** isomorphism:

$$\mathbb{K}[x,y,z]/\mathrm{sat}(T) \;\cong\; \mathbb{K}[x,y,z]/\mathrm{sat}(T_1) \otimes \mathbb{K}[x,y,z]/\mathrm{sat}(T_2)$$

Ideal-Variety Correspondence

$\mathcal{I} \subseteq \mathbb{K}[\underline{X}]$ is an **ideal** if:
(i) $0 \in \mathcal{I}$,
(ii) for $f, g \in \mathcal{I},\ f + g \in \mathcal{I}$, and
(iii) for $f \in \mathcal{I}, r \in \mathbb{K}[\underline{X}],\ rf \in \mathcal{I}$

For $f, g \in \mathbb{K}[\underline{X}],\ \langle f, g \rangle = \langle f \rangle + \langle g \rangle = \{r_1 f + r_2 g \mid r_1, r_2 \in \mathbb{K}[\underline{X}]\}$

$\{f_1, f_2, \dots, f_k\} = F \subset \mathbb{K}[\underline{X}]$, $\langle F \rangle$ is all ***polynomial consequences*** of F:

↳ that is, all results which follow from $f_1 = f_2 = \cdots = f_k = 0$.

↳ $V(F) = V(\langle F \rangle)$

Sum: $V(\mathcal{I} + \mathcal{J}) = V(\mathcal{I}) \cap V(\mathcal{J})$

Product: $V(\mathcal{I}\mathcal{J}) = V(\mathcal{I} \cap \mathcal{J}) = V(\mathcal{I}) \cup V(\mathcal{J})$

Saturation: $V(\mathcal{I} : \mathcal{J}^\infty) = \overline{V(\mathcal{I}) \setminus V(\mathcal{J})}$

Note: for $S \subset \overline{\mathbb{K}}^n$, $\overline{S}$ is its *closure*: the smallest variety V such that $S \subseteq V$

Regular Chains and Triangular Decomposition

Let T be a regular chain and $h_T = \prod_{p \in T} \mathrm{initial}(p)$

Saturated ideal of a regular chain:

- $\mathrm{sat}(T) = \langle T \rangle : h_T^{\infty}$
- $\mathrm{sat}(\varnothing) = \langle 0 \rangle$

Quasi-component of a regular chain:

- $W(T) := V(T) \setminus V(h_T)$
- $\overline{W(T)} = V(\mathrm{sat}(T))$

A **triangular decomposition** of an input system $F \subseteq \mathbb{K}[\underline{X}]$ is a set of regular chains $T_1, \ldots, T_e$ such that:

(*Kalkbrener decomposition*) $V(F) = \bigcup_{i=1}^{e} \overline{W(T_i)}$, or

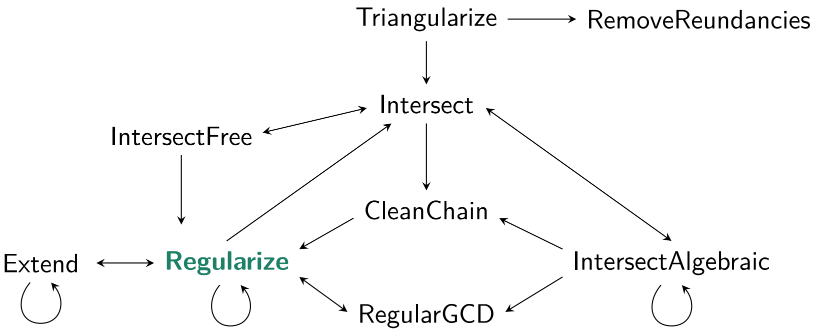
(*Lazard-Wu decomposition*) $V(F) = \bigcup_{i=1}^{e} W(T_i)$

Note: Some T_i may be *redundant*; $\exists j \; W(T_i) \subseteq W(T_j)$

All roads lead to Regularize

The Triangularize algorithm iteratively calls intersect, then a network of mutually recursive functions do the heavy-lifting.

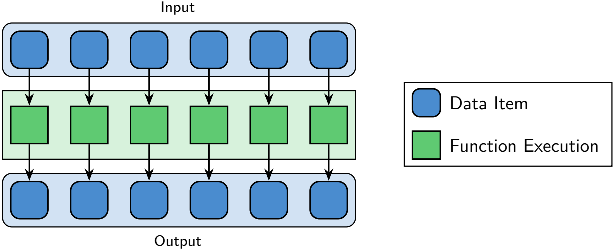
↳ In all cases, polynomials are forced to be regular and splittings are (possibly) found via **Regularize**



Outline

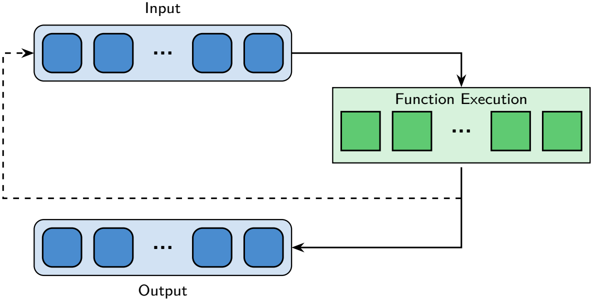
Map

- Simultaneously execute a function on each data item in a collection
- If more data items than threads, apply the pattern block-wise: partition the collection and apply one thread to each partition
- Often simplified as just a **parallel_for** loop
- Where multiple map steps are performed in a row, they must operate in **lockstep**



Workpile

- Workpile generalizes map pattern to a *queue* of tasks
- Tasks in-flight can add new tasks to input queue
- Threads take tasks from queue until it is empty
- Very similar in structure to a thread pool
- Can be seen as a **parallel_while** loop



Triangularize: Incremental Triangular Decomposition

Algorithm 1 Triangularize(F)

Input: a finite set $F \subseteq \mathbb{K}[\underline{X}]$

Output: regular chains $T_1, \ldots, T_e \subseteq \mathbb{K}[\underline{X}]$ such that $V(F) = W(T_1) \cup \cdots \cup W(T_e)$

$\mathcal{T} := \{\varnothing\}$
for $p \in F$ **do**
  $\mathcal{T}' := \varnothing$
  parallel_for $T \in \mathcal{T}$ ▷ map Intersect over the current components
    $\mathcal{T}' := \mathcal{T}' \cup$ **Intersect**(p, T)
  end for
  $\mathcal{T}$:= RemoveRedundantComponents($\mathcal{T}'$) ▷ prune redundancies each step
return $\mathcal{T}$

- **Coarse-grained parallelism**: each Intersect represents substantial work
- At each "level" there $|\mathcal{T}|$ components with which to intersect, yielding $|\mathcal{T}| - 1$ additional threads
- Performs a *breadth-first search*, with synchronization at each level

Triangularize: a task-based approach

Algorithm 2 TriangularizeByTasks(F)

Input: a finite set $F \subseteq \mathbb{K}[\underline{X}]$

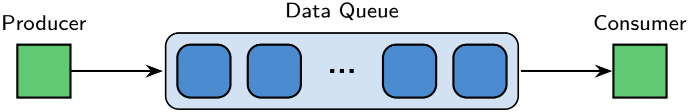
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbb{K}[\underline{X}]$ such that $V(F) = W(T_1) \cup \cdots \cup W(T_e)$

```
1: Tasks := { (F, ∅) };  𝒯 := ∅
2: while |Tasks| > 0 do
3:     (P, T) := pop a task from Tasks
4:     Choose a polynomial p ∈ P; P′ := P ∖ {p}
5:     for T′ in Intersect(p, T) do
6:         if |P′| = 0 then 𝒯 := 𝒯 ∪ {T′}
7:         else Tasks := Tasks ∪ {(P′, T′)}
8: return RemoveRedundantComponents(𝒯)
```

- Performs a *depth-first search*
- *Tasks* is essentially a data structure for a **task scheduler**
- Tasks create more tasks, workers pop Tasks until none remain.
- Adaptive to load-balancing, no inter-task synchronization

Producer-Consumer, Asynchronous Generators

- Two functions connected by a queue, executing concurrently
- The producer produces data items, pushing them to the queue
- The consumer processes data items, pulling them from the queue



- Producer may be considered as an **iterator** or **generator**
  ↳ special kinds of coroutines which **yield** data items one at a time, rather than many as a collection
- If generation of data is expensive, generator may execute **asynchronously**, fulfilling the role of producer

Intersect as a Generator

Algorithm 3 **Intersect**(p, T)

Input: $p \in \mathbb{K}[\underline{X}] \setminus \mathbb{K}$, $v :=$ mvar(p), a regular chain T s.t. $T = T_v^- \cup T_v$

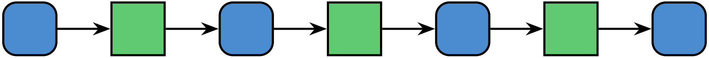
Output: regular chains $T_1, \ldots, T_e$ satisfying specs.

for $(g_i, T_i) \in$ **RegularGCD**(p, T_v, v, T_v^-) **do**
    if $\dim(T_i) \neq \dim(T_v^-)$ **then**
        for $T_{i,j} \in$ **Intersect**(p, T_i) **do**
            yield $T_{i,j}$
    else
        if $g_i \notin \mathbb{K}$ and $\deg(g_i, v) > 0$ **then**
            yield $T_i \cup \{g_i\}$
        for $T_{i,j} \in$ **Intersect**$(\mathrm{lc}(g_i, v), T_i)$ **do**
            for $T' \in$ **Intersect**$(p, T_{i,j})$ **do**
                yield T'

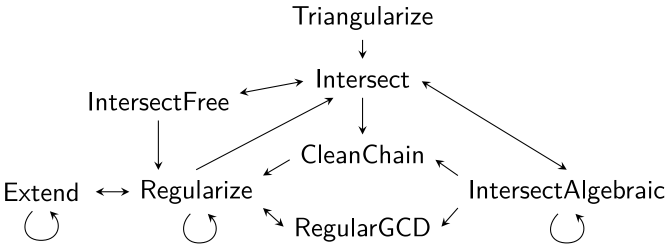
- **yield** "produces" a single data item, and then continues computation
- each **for** loop iteration consumes one data item from a generator

Pipeline

- A sequence of stages where the output of one stage is used as the input to another
- Two consecutive stages form a producer-consumer pair
- Internal stages are both producer and consumer
- Typically, a pipeline is constructed statically through code organization
- Pipelines can be created dynamically and implicitly with **asynchronous generators** and the call stack



Triangularize Subroutine Pipeline



- all subroutines as generators allows pipeline to evolve dynamically with the call stack.
- data **streams** between subroutines; all soubroutines are effectively *non-blocking*
- call stack forms a **tree** as several generators invoked by one consumer
- pipeline creates **fine-grained parallelism** since work diminishes with each recursive call

Outline

Thread-level parallelism

Multithreading: using (software) threads—multiple independent control flows in one process—for concurrency

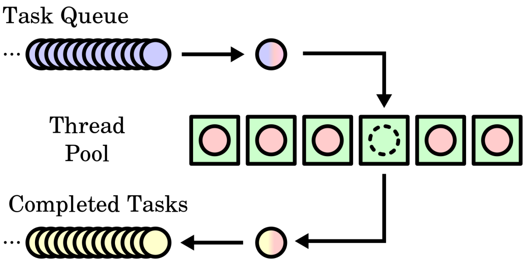
Hardware enables parallelism by executing multiple threads simultaneously on independent processors (i.e. **hardware threads**)

Parallel overheads:

- ↳ software threads > hardware threads $\implies$ **over-subscription**
- ↳ **spawning** and **joining** threads
- ↳ **load imbalance**: unevenly distributed work between threads; some are left idle while others are still executing
- ↳ inter-thread communication and **synchronization**

C++11 *Thread Support Library* supports object-oriented multithreading

Thread Pools



- A fixed number of threads are spawned, only once, at the beginning of the program
- Threads remain active for the program lifetime
- Threads receive *tasks*, code blocks or functions, to execute as needed
- Threads return to the pool upon completing their task
- Services requests from multiple client codes **enabling cooperation**

`ExecutorThreadPool`

- A pool of `ExecutorThread`s able to execute any *function object*
- `AsyncObjectStream` implements producer-consumer pattern to stream objects between threads
  - ↳ includes function objects, and later, objects for generators
- Allows for thread cooperation: (normal) tasks vs. **priority tasks**
  - ↳ If all normal threads busy, new "priority thread" spawned to immediately launch a priority task
  - ↳ A returning thread is *retired* to avoid over-subscription
  - ↳ Limits total number priority threads; after limit, priority tasks pushed to the **front of queue**
- Enables **optional parallelism**: user specifies areas for concurrency in code, runtime dynamically chooses which to execute in parallel

AsyncGenerator and AsyncObjectStream

We want an *object-oriented* approach to create and use generators

↳ `AsyncGenerator` acts as interface between producer and consumer

↳ Use `AsyncObjectStream` as producer-consumer queue

- The consumer constructs the `AsyncGenerator`, passing the constructor the producer's function and arguments
- The `AsyncGenerator` inserts itself into the producer's list of arguments so that it has reference to the generator object
- The producer's signature should be:

```
void producerFunction(..., AsyncGenerator<Object>&);
```

- **If `ExecutorThreadPool` not empty** producer executes asynchronously, otherwise execute serially on consumer's thread

AsyncGenerator Example

```
void FibonacciGen(int n, AsyncGenerator<int>& gen) {
    int Fn_1 = 0;
    int Fn = 1;
    for (int i = 0; i < n; ++i) {
        gen.generateObject(Fn_1); //yield Fn_1 and continue
        Fn = Fn + Fn_1;
        Fn_1 = Fn - Fn_1;
    }
    gen.setComplete();
}

void Fib() {
    int n, fib;
    std::cin >> n;
    AsyncGenerator<int> gen(FibonacciGen, n);

    //get one integer at a time until generator is finished
    while (gen.getNextObject(fib)) {
        std::cerr << fib << std::endl;
    }
}
```

Outline

Improved Parallel Performance, Avoiding Redundancies

- **TriangularizeByTasks** improved parallelism but could not intermittently remove redundancies
  ↳ we will investigate a hybrid approach: depth-first search with task cancellation to prune redundant branches

- Parallelize low-level routines to add parallelism and load-balance when there is little to no component-level parallelism to exploit

- **Memoization** of subroutines
  ↳ Typical of (mutually-)recursive algorithms
  ↳ Different branches of computation deriving from the same regular chain are very likely to share geometric and algebraic features
  ↳ *Caching* the results of operations in, e.g., a hash table will avoid redundant re-computation

Dynamic Evaluation and Avoiding Redundant Computation

Dynamic Evaluation: an automatic case discussion based on choices of particular values on parameters [13, 14]

Regularity testing:

$$T = \begin{cases} y^2 - 1 = 0 \\ z - 1 = 0 \end{cases} \quad \xrightarrow{y+1=0} \quad T_1 = \begin{cases} y + 1 = 0 \\ z - 1 = 0 \end{cases}$$

$$T = \begin{cases} y^2 - 1 = 0 \\ z - 1 = 0 \end{cases} \quad \xrightarrow{y+1\neq 0} \quad T_2 = \begin{cases} y - 1 = 0 \\ z - 1 = 0 \end{cases}$$

Two branches are likely to share geometric and algebraic features

$$T_5 = \begin{cases} a(y,z) \\ b(z)c(z) \end{cases} \qquad T_6 = \begin{cases} d(y,z) \\ b(z)c(z) \end{cases}$$

- T_5 splitting into $\{a(y,z),\ b(z)\}$ and $\{a(y,z),\ c(z)\}$ should automatically split T_6 into $\{d(y,z),\ b(z)\}$ and $\{d(y,z),\ c(z)\}$
- Requires a **universal view** and shared data structure [10]

Polymorphic Regular Chains

- Triangular decomposition, in theory, works over any perfect field
- Current implementation limited to the field of rationals $\mathbb{Q}$
- Working over a *finite field* enables additional component-level parallelism as components more easily split [23]
- Solving over finite fields is itself useful in practice and is required as a *modular method* to solve very hard problems [11]
- Our regular chains code requires refactoring to properly use a *generic multivariate polynomial interface*, and thus rely on polymorphism

References

[1] J. Abbott and A. M. Bigatti. *CoCoALib: a C++ library for doing Computations in Commutative Algebra*. Available at `http://cocoa.dima.unige.it/cocoalib`.

[2] G. Attardi and C. Traverso. “Strategy-Accurate Parallel Buchberger Algorithms”. In: *Journal of Symbolic Computation* 22 (1996), pp. 1–15.

[3] P. Aubry, D. Lazard, and M. Moreno Maza. “On the Theories of Triangular Sets”. In: *Journal of Symbolic Computation* 28.1-2 (1999), pp. 105–124.

[4] J. Böhm, W. Decker, S. Laplagne, G. Pfister, A. Steenpaß, and S. Steidel. “Parallel algorithms for normalization”. In: *J. Symb. Comput.* 51 (2013), pp. 99–114.

[5] F. Boulier, F. Lemaire, and M. Moreno Maza. “Well known theorems on triangular systems and the D5 principle”. In: *Transgressive Computing 2006, Proceedings*. Granada, Spain, 2006.

[6] B. Buchberger. “The parallelization of critical-pair/completion procedures on the L-Machine”. In: *Japanese Symposium on Functional Pogramming, Proceedings*. 1987, pp. 54–61.

[7] C. Chen and M. Moreno Maza. “Algorithms for computing triangular decomposition of polynomial systems”. In: *Journal of Symbolic Computation* 47.6 (2012), pp. 610–642.

[8] C. Chen, J. H. Davenport, J. P. May, M. Moreno Maza, B. Xia, and R. Xiao. “Triangular decomposition of semi-algebraic systems”. In: *Journal of Symbolic Computation* 49 (2013), pp. 3–26.

[9] C. Chen, O. Golubitsky, F. Lemaire, M. Moreno Maza, and W. Pan. “Comprehensive Triangular Decomposition”. In: *Computer Algebra in Scientific Computing, CASC 2007, Proceedings*. 2007, pp. 73–101.

[10] C. Chen and M. Moreno Maza. “An Incremental Algorithm for Computing Cylindrical Algebraic Decompositions”. In: *Asian Symposium on Computer Mathematics, ASCM 2012, Proceedings*. Springer, 2012, pp. 199–221.

[11] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. "Lifting techniques for triangular decompositions". In: *International Symposium on Symbolic and Algebraic Computation, ISSAC 2005, Proceedings*. 2005, pp. 108–115.

[12] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR *4-1-1 — A computer algebra system for polynomial computations*. http://www.singular.uni-kl.de. 2018.

[13] J. D. Dora, C. Dicrescenzo, and D. Duval. "About a New Method for Computing in Algebraic Number Fields". In: *European Conference on Computer Algebra, EUROCAL 1985, Proceedings Volume 2: Research Contributions*. Vol. 204. Lecture Notes in Computer Science. Springer, 1985, pp. 289–290.

[14] D. Duval. "Algebraic Numbers: An Example of Dynamic Evaluation". In: *Journal of Symbolic Computation* 18.5 (1994), pp. 429–445.

[15] J. C. Faugere. "Parallelization of Gröbner Basis". In: *Parallel Symbolic Computation, PASCO 1994, Proceedings*. Vol. 5. World Scientific. 1994, p. 124.

[16] M. Gastineau and J. Laskar. "Parallel sparse multivariate polynomial division". In: *Parallel Symbolic Computation, PASCO 2015, Proceedings*. 2015, pp. 25–33.

[17] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. 3rd ed. NY, USA: Cambridge University Press, 2013.

[18] J. Grabmeier, E. Kaltofen, and V. Weispfenning, eds. *Computer algebra handbook*. Springer-Verlag, 2003.

[19] J. Hu and M. B. Monagan. "A Fast Parallel Sparse Polynomial GCD Algorithm". In: *International Symposium on Symbolic and Algebraic Computation, ISSAC 2016, Proceedings*. 2016, pp. 271–278.

[20] M. B. Monagan and R. Pearce. "Parallel sparse polynomial multiplication using heaps". In: *International Symposium on Symbolic and Algebraic Computation, ISSAC 2009, Proceedings*. ACM, 2009, pp. 263–270.

[21] M. B. Monagan and B. Tuncer. "Sparse Multivariate Hensel Lifting: A High-Performance Design and Implementation". In: *Mathematical Software - ICMS 2018, Proceedings*. 2018, pp. 359–368.

[22] M. Moreno Maza. *On Triangular Decompositions of Algebraic Varieties*. Tech. rep. TR 4/99. Presented at the MEGA-2000 Conference, Bath, England. Oxford, UK: NAG Ltd, 1999.

[23] M. Moreno Maza and Y. Xie. "Component-level parallelization of triangular decompositions". In: *Parallel Symbolic Computation, PASCO 2007, Proceedings*. ACM. 2007, pp. 69–77.

[24] B. D. Saunders, H. R. Lee, and S. K. Abdali. "A parallel implementation of the cylindrical algebraic decomposition algorithm". In: *International Symposium on Symbolic and Algebraic Computation, ISSAC 1989, Proceedings*. Vol. 89. 1989, pp. 298–307.

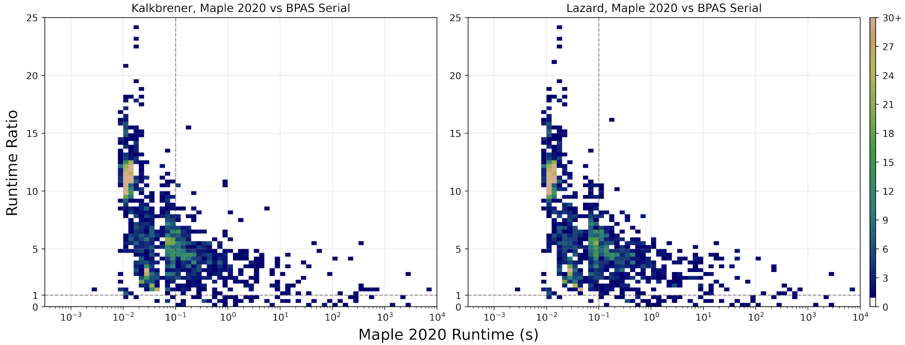
[25] The LinBox group. *LinBox*. v1.6.3. 2019. URL: `http://github.com/linbox-team/linbox`.

[26] W. Wu. "A zero structure theorem for polynomial equations solving". In: *MM Research Preprints* 1 (1987), pp. 2–12.

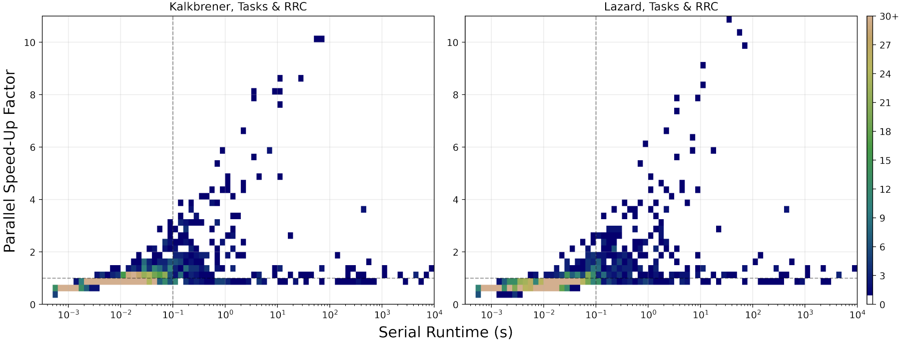
[27] W. Wu. "On zeros of algebra equations—an application of Ritt principle". In: *Kexeu Tongbao* 31.1 (1986), pp. 1–5.

Outline

BPAS vs *RegularChains* in *Maple*

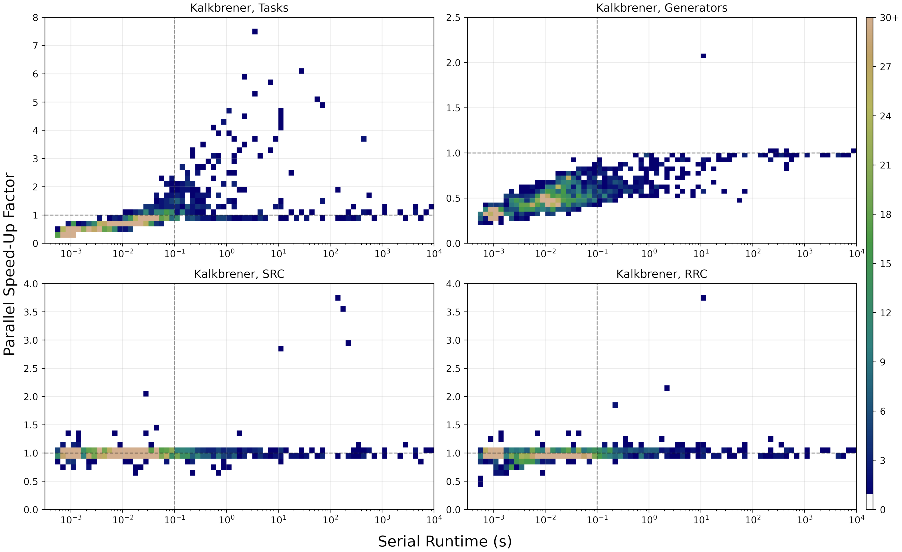


Parallel Speedup

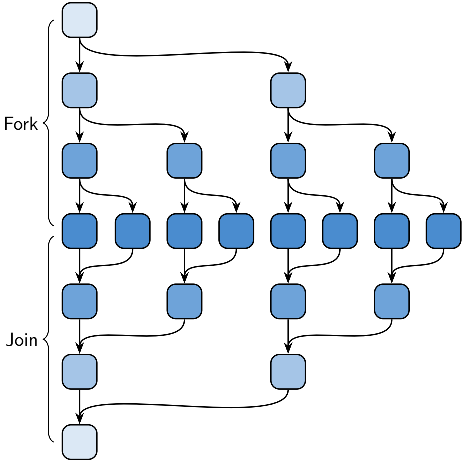


SRC: Subresultant chain computations, RRC: removal of redundant components

Speedup for each parallel scheme individually



Fork-Join



- **Fork**: divide problem and execute separate calls in parallel
- **Join**: merge parallel execution back into serial
- Recursively applying fork-join can easily parallelize a divide-and-conquer algorithm

Divide-and-Conquer and Fork-Join

Remove redundancies from a list of regular chains with DnC:

- Recursively and concurrently obtain two irredundant lists, then merge.
- Merge can be done as a **map**

Algorithm 4 RemoveRedundantComponents($\mathcal{T}$)

Input: a finite set $\mathcal{T} = \{T_1, \ldots, T_e\}$ of regular chains
Output: an irredundant set $\mathcal{T}'$ with the same algebraic set as $\mathcal{T}$

```
if e = 1 then return 𝒯
ℓ := ⌈e/2⌉; 𝒯_{≤ℓ} := {T_1, ..., T_ℓ}; 𝒯_{>ℓ} := {T_{ℓ+1}, ..., T_e}
𝒯_1 := spawn RemoveRedundantComponents(𝒯_{≤ℓ})
𝒯_2 := RemoveRedundantComponents(𝒯_{>ℓ})
sync
𝒯_1' := ∅;  𝒯_2' := ∅
parallel_for T_1 ∈ 𝒯_1
  | if ∀T_2 in 𝒯_2 IsNotIncluded (T_1, T_2) then 𝒯_1' := 𝒯_1' ∪ {T_1}
parallel_for T_2 ∈ 𝒯_2
  | if ∀T_1 in 𝒯_1' IsNotIncluded (T_2, T_1) then 𝒯_2' := 𝒯_2' ∪ {T_2}
return 𝒯_1' ∪ 𝒯_2'
```

Threading Primitives

C++11 introduced the *Thread Support Library*

- **`std::thread`**
  - ↳ C++ class encapsulating a thread (often a `pthread`) and its low-level spawn and join
- **`std::mutex`**
  - ↳ shared object between threads to indicate *mutual exclusion* to a **critical region**.
  - ↳ `mutex` is *locked* or *owned* by at most one thread at a time.
- **`std::lock_guard`**, **`std::unique_lock`**
  - ↳ temporary object wrapping a `mutex` whose object lifetime automatically locks and unlocks the `mutex`.
  - ↳ the constructor **blocks** and only returns once the shared `mutex` is successfully owned by the calling thread.
- **`std::condition_variable`**
  - ↳ blocks the current thread and temporarily releases a lock
  - ↳ receives notification from another thread to awaken the blocked thread

std::function

Functors, function objects, callable objects

- First-class objects which are callable using normal function syntax
- Are often constructed by passing function names, function pointers
- std::bind binds arguments to a function or function object, returning a function object which requires fewer arguments

```
void printInteger(int a) {
    std::cout << a << std::endl;
}

//Function object from function name
std::function<void(int)> f_printInt(printInteger);
f_printInt(12);

//Function object binding arguments to function name
std::function<void()> f_print42( std::bind(printInteger,42) );
f_print42();
```

Function Executor Thread: Implementation

```
class FunctionExecutorThread {
    AsyncObjectStream<std::function<void()>> requestQueue;
    std::thread m_worker;

    std::mutex m_mutex;
    std::condition_variable m_cv;

    FunctionExecutorThread() {
        //member functions require pointer to member
        m_worker = std::thread(
            &FunctionExecutorThread::eventLoop, this);
    }

    //NOTE: copy constructor and copy operator are deleted

    void eventLoop();

    void sendRequest(std::function<void()>);

    void waitForThread();
}
```

AsyncObjectStream

1. a **synchronized** producer-consumer queue of objects, and
2. a *blocking* mechanism to keep the ExecutorThread alive and idle when waiting for tasks

```
template <class Object>
class AsyncObjectStream {
  //Producer: add an object to the queue
  void addResult(Object& res);

  //Producer: close the producer end of stream,
  //          no more objects to produce
  void resultsFinished();

  //Consumer: wait for an object from the queue, return true
  //          iff stream is open and objects available
  bool getNextObject(Object& res);

  //Consumer: determine if queue is currently empty
  void streamEmpty();
};
```

AsyncObjectStream: getNextObject

```
bool getNextObject(Object& res) {
    std::unique_lock<std::mutex> lk(m_mutex);
    if (finished && retObjs.empty()) {
        lk.unlock();
        return false;
    }

    //Wait in a loop in case of spurious wake ups
    while (!finished && retObjs.empty() {
        m_cv.wait(lk);
    }

    if (finished && retObjs.empty()) {
        lk.unlock();
        return false;
    } else {
        res = retObjs.front();
        retObjs.pop();
        lk.unlock();
        return true;
    }
}
```

ExecutorThreadPool

- A thread pool built using FunctionExecutorThreads
- An internal queue of tasks and queue of threads
- When threads are busy, they are temporarily removed from the pool
- When all threads busy, tasks are added to task queue

```
class ExecutorThreadPool {

private:
    std::deque<FunctionExecutorThread*> threadPool;
    std::deque<std::function<void()>> taskPool;
    std::mutex m_mutex;
    std::condition_variable m_cv; //used in waitForThreads

    void tryPullTask();
    void putBackThread(FunctionExecutorThread* t);

public:
    void addTask(std::function<void()> f);
    void waitForThreads();
}
```

ExecutorThreadPool: Flexible Usage (1/2)

- In support of certain **parallel patterns**, clients can (temporarily) obtain ownership of threads from the pool, rather than using addTask
- Abstract away actual threads through **thread IDs**
- Once thread obtained, repeat Steps 2–3 as often as necessary

```
class ExecutorThreadPool {
    //Storage for threads removed from pool by obtainThread
    std::vector<FunctionExecutorThread*> occupiedThreads;

    //Step 1: obtain a thread's ID, removing it from the pool
    void obtainThread(threadID& id);

    //Step 2: execute a task on a particular thread
    void executeTask(threadID id, std::function<void()>& f);

    //Step 3 (optional): wait for thread to become idle
    void waitForThread(threadID id);

    //Step 4: return thread to pool (waits before returning)
    void returnThread(threadID id);
}
```

ExecutorThreadPool: Flexible Usage (2/2)

- In support of certain **parallel patterns**, clients can (temporarily) obtain ownership of threads from the pool, rather than using `addTask`
- Can obtain one thread at a time (previous slide), or multiple threads at a time

```
class ExecutorThreadPool {

    //Step 1: obtain threadIDs, removing them from the pool
    void obtainThreads(std::vector<threadID>& ids);

    //Step 2: execute a task on a particular thread
    void executeTask(threadID id, std::function<void()>& f);

    //Step 3 (optional): wait for threads to become idle
    void waitForThreads(std::vector<threadID>& ids);

    //Step 4: return threads to pool (waits before returning)
    void returnThreads(std::vector<threadID>& ids);
}
```

Motivation: Usability

BPAS is concerned with **accessibility**, **interoperability**, and **usability**.

- Open-source and written in C/C++ provides the former two.

To achieve usability, we consider best practices for its interface.

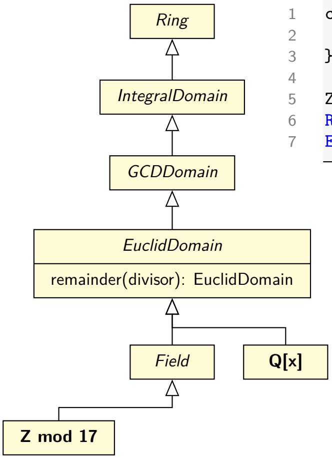
1. Natural: a symmetric encoding of the algebraic hierarchy

   $$\text{field} \subset \text{Euclidean domain} \subset \text{GCD domain} \subset \text{integral domain} \subset \text{ring}$$

2. Easy to use: an object-oriented design with well-defined interfaces. A so-called **algebraic class hierarchy**: rings are classes and elements of a ring are objects
3. Encapsulation: hide complexity of low-level code; class interfaces
4. Extensible: adaptable to new (user-created) types, type composition
5. Type safe: compile-time type safety and mathematical type safety

Motivation: Type Safety

A naive implementation of the algebraic hierarchy as a class hierarchy creates mathematically unsafe operations via polymorphism.




```
class EuclidDomain {
    EuclidDomain remainder(EuclidDomain& divisor);
}

Zmod17 a;
RationalPoly b;
EuclidDomain r = a.remainder(b);
```

- $\mathbb{Z}/17\mathbb{Z}$ and $\mathbb{Q}[x]$ are Euclidean domains
- the code is valid via polymorphism
- could compile, but then issues at runtime.

Existing Solutions

In other compiled libraries, mathematical type safety is only a runtime property maintained through runtime value checks.

- In Singular's libpolys [12], all algebraic types are a single class. Instance variables (Booleans, enums) store properties of rings
- In CoCoA [1] rings and elements of a ring are separate classes. Elements hold references to their "owning" ring which are compared at runtime and errors thrown if not identical.
- In LinBox [25] rings and elements are again distinct, with references to abstract ring elements being downcasted for operations.

Our Goal: provide both compile-time mathematical type safety and a natural, extensible object-oriented hierarchy for the algebraic hierarchy

Algebraic Class Hierarchy

The algebraic hierarchy as a class hierarchy with mathematical type safety

Solution: an **abstract class template hierarchy**.

- abstract classes: well-defined interfaces, default behaviour
- inheritance incrementally extends/builds interface
- template parameter modifies interface to restrict method parameters

```
template <class Derived>
class Ring {...};

template <class Derived>
class IntegralDomain : Ring<Derived> {...};

template <class Derived>
class GCDDomain : IntegralDomain<Derived> {...};

template <class Derived>
class EuclidDomain : GCDDomain<Derived> {
    Derived remainder(Derived& divisor);
}
```

Algebraic Class Hierarchy: Static Polymorphism

Static polymorphism via *Curiously Recurring Template Pattern*: concrete class is used as template parameter of super class.

- function resolution occurs at compile-time
- method declaration restricts params to be compile-time compatible

```
template <class Derived>
class EuclidDomain : GCDDomain<Derived> {
    Derived remainder(const Derived& divisor);
};

class Integer : EuclidDomain<Integer> {...}; //CRTP
//Integer remainder(const Integer& divisor);

class RationalPoly : EuclidDomain<RatonalPoly> {...}; //CRTP
//RationalPoly remainder(const RationalPoly& divisor);

Integer x;  RationalPoly p;

//compiler error: EuclidDomain<RationalPoly>::remainder
//                takes RationalPoly as parameter
RationalPoly r = p.remainder(x);
```

Algebraic Class Hierarchy with Polynomials

Extend abstract class template hierarchy to include polynomials

- parameterize polynomial abstract classes by coefficient ring

```
template <class Derived>
class Ring {...};

template <class CoefRing, class Derived>
class Poly : Ring<Derived> {...};

class RationalPoly : Poly<RationalNumber, RationalPoly> {...};
```

Problem: What if `CoefRing` is not actually a ring?

- e.g. `Poly<std::string>` or `Poly::<Apple>`

Problem: polynomial rings form different algebraic types depending on the ground ring

- e.g. $\mathbb{Q}[x]$ is a Euclidean domain, $\mathbb{Z}[x]$ is an integral domain

Constraining the Ground Ring

At compile-time ensure that a polynomial's coefficient ring is an actual ring with template metaprogramming.

`Derived_from<T, Base>`: statically determines if `T` is a subclass of `Base`, creating a compiler-error if not

- inheriting from `Derived_from` forces evaluation at compile-time during template instantiation
- Coefficient ring must be a subclass of `Ring`
- `Poly` can assume `CoefRing` has a certain interface at minimum

```
template <class T, class Base>
class Derived_from {...};

template <class CoefRing, class Derived>
class Poly : Ring<Derived>,
Derived_from<CoefRing, Ring<CoefRing>> {...};
```

Adapting to Different Coefficient Rings (1/2)

Determine type of coefficient ring using *compile-time introspection*

- **Conditional inheritance** then determines correct algebraic type and interface for polynomials over that ring
- "Dynamic" type creation via introspection, template instantiation

`is_base_of<T, Base>::value`

- compile-time Boolean value determines if `T` is a subclass of `Base`

`conditional<Bool, T1, T2>::value`

- A compile-time tertiary conditional operator for choosing types
- `Bool ? T1 : T2`

```
template <class CRing, class Derived>
class Poly : conditional< is_base_of<CRing, Field<CRing>>::value,
                          EuclidDomain<Derived>,
                          Ring<Derived>
                        >::value {...};
```

Adapting to Different Coefficient Rings (2/2)

A chain of `conditional`'s create a case-discussion at compile-time

- *Tester* hierarchy separates introspection from actual interface
- Concrete classes inherit from *Polynomial* to automatically determine their type and interface