Fusion for Metaprogramming

Eliminating Intermediate Programs in Code Generation and Analysis

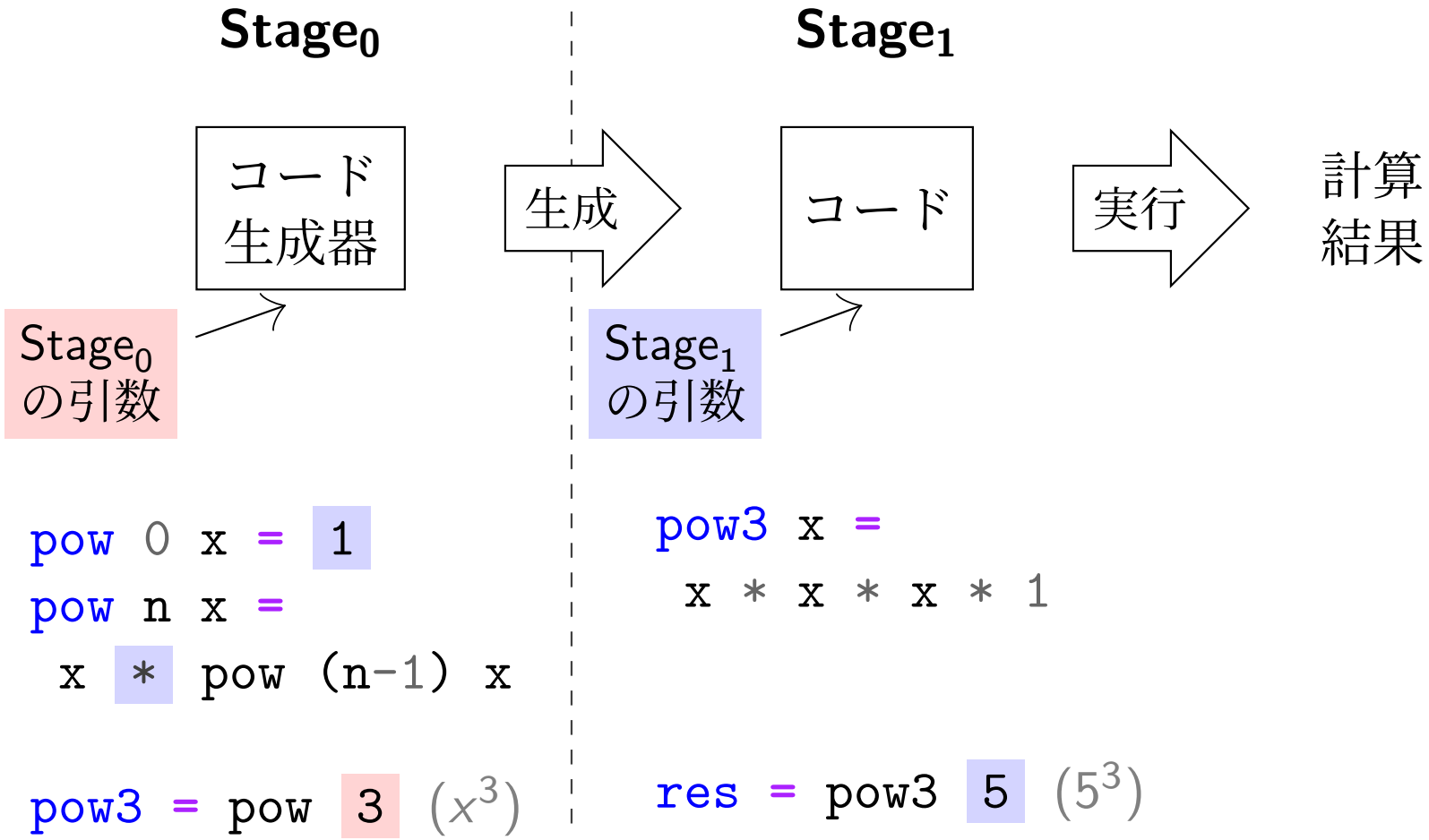
李 張帆　　亀山 幸義

筑波大学

2025 年 9 月 5 日
日本ソフトウェア科学会第 42 回大会

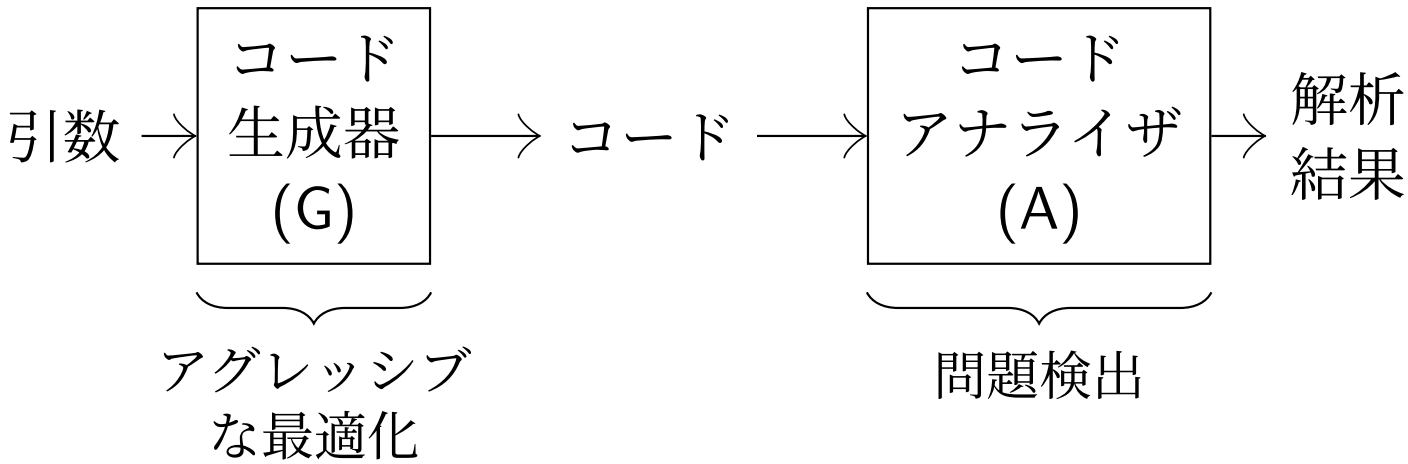
研究背景: 多段階計算 (Multi-Stage Programming)

多段階計算とは，計算を複数の段階に分けて処理する手法である．



研究課題 (1/2)

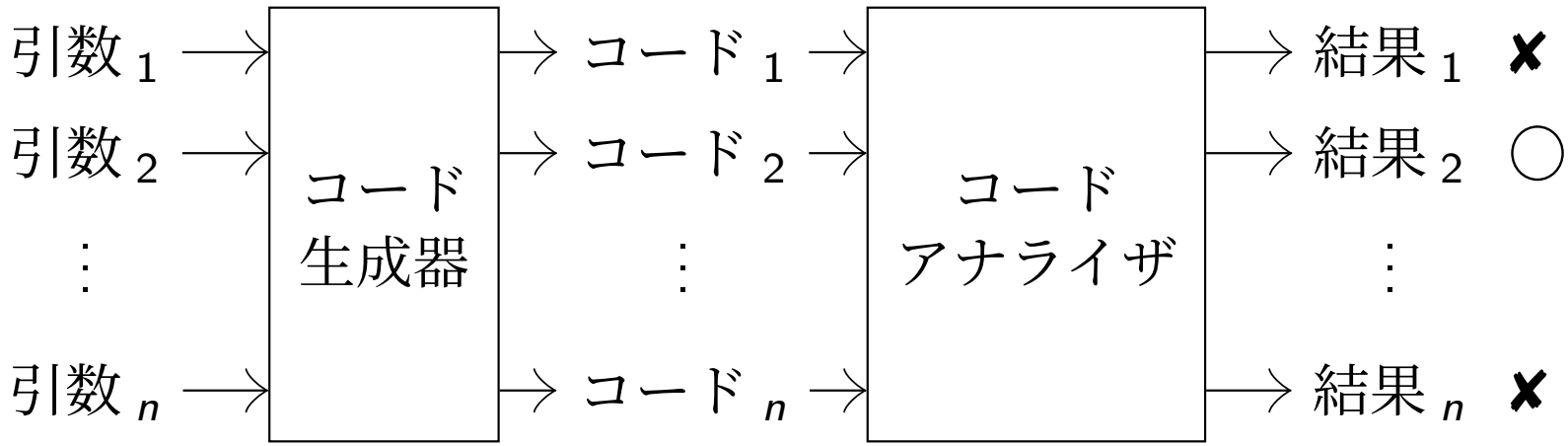
生成されたコードに対する解析



- コード生成器 G は，問題を含むコードを生成する可能性がある．
  - 整数オーバーフロー
  - 浮動小数点演算誤差の増大

研究課題 (2/2)

多数の引数に対する探索



- (コード$_i$ s.t. 結果$_i$ = ◯) を出力する
- 複数の引数を組み合わせると，探索空間が膨大になる

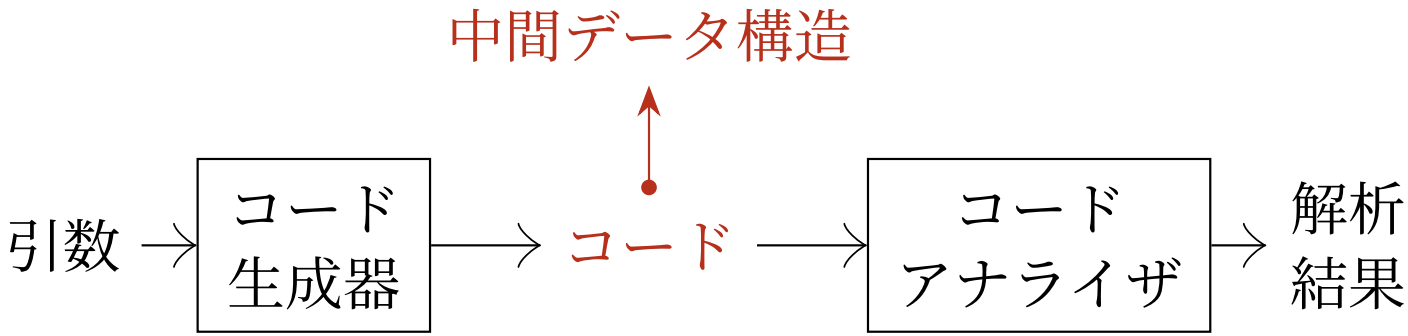
問題点

☞ コードが大量に生成されるため，探索が非効率的になる

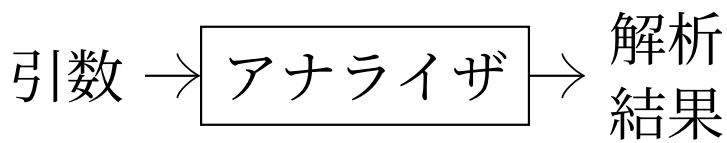
本研究の概要 (1/2)

研究目標: 生成されたコードに対する解析の高速化

提案手法



⇓ (Fusion)



研究概要 (2/2)

- Hylomorphism となるコード生成器・コードアナライザのためのフュージョンフレームワークを提案した.
  - Parametric higher-order abstract syntax (PHOAS) で生成されたコードを表現
  - 実装のために既存のフュージョン規則を改良
- Haskell でフュージョンフレームワークを実装して，Number Theoretic Transform (NTT) の最適化に対して実験を行った.

前提知識 (1/2)

フュージョンに関する先行研究では、中間データ構造として**代数的データ型**が用いられることが多い.

代数的データ型

- 直積 ($\times$) と直和 ($+$) で定義された自己関手でモデル化される

```
-- L(X) = 1 + Int × X
data L x = Nil | Cons Int x
```

- 自己関手の不動点で再帰的データ型を表現する

```
newtype Fix f = In { out :: f (Fix f) }
type List = Fix L
```

$$\mathbf{L}(\mathtt{List}) \underset{\mathtt{out}}{\overset{\mathtt{In}}{\rightleftarrows}} \mathtt{List}$$

前提知識 (2/2)

Recursion Schemes: 構造化した再帰関数

- **Catamorphism:** 畳み込み操作を抽象化した概念

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata φ = φ . fmap (cata φ) . out
```

- **Anamorphism:** 展開操作を抽象化した概念

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana ψ = In . fmap (ana ψ) . ψ
```

Hylomorphism

Catamorphism と **Anamorphism** を一般化した概念

```
⟦•, •⟧ :: Functor f => (f b -> b, a -> f a) -> a -> b
⟦φ, ψ⟧ = φ . fmap ⟦φ, ψ⟧ . ψ
```

- (cata φ . ana ψ) と等価である
- cata $\varphi = [\![\varphi, \mathsf{out}]\!]$, ana $\psi = [\![\mathsf{In}, \psi]\!]$

本研究: メタプログラミングへの適用 (1/3)

提案手法: Hylomorphism となるコード生成器・アナライザの合成を変換 (Fusion) して，生成されたコードを消去する

生成されたコードの表現

本研究は，Parametric high-order abstract syntax (PHOAS) で中間プログラムを表現する．

```
data ULC v x = Var v | Lam (v -> x) | App x x
```

- メタ言語の関数を用いて，識別子の衝突を回避する
- 関手の引数 (`x`) を関数の戻り値に制限することにより，Hylomorphism は自然に定義できる．

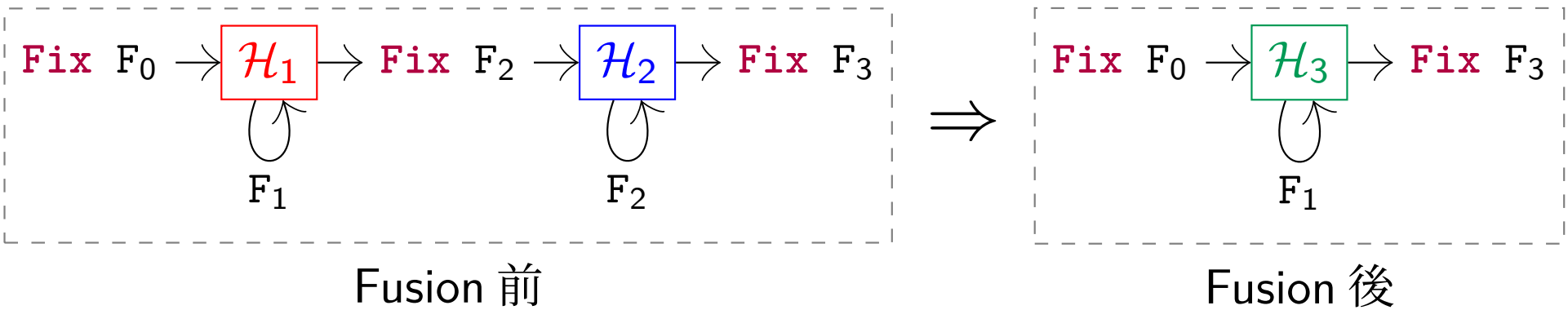
本研究: メタプログラミングへの適用 (2/3)

[Takano+1995] のフュージョン規則を，実装のために改良した.

☞ 適用範囲は拡張していない

フュージョン規則の例: AbsHylo-Left

$$\frac{\begin{array}{rcl}\tau_1 & : & \forall A.(\mathrm{F}_2\,A \to A) \to \mathrm{F}_1\,A \to A \\ \tau_2 & : & \forall A.(\mathrm{F}_3\,A \to A) \to \mathrm{F}_2\,A \to A \\ \sigma & : & \forall A.(A \to \mathrm{F}_0\,A) \to A \to \mathrm{F}_1\,A\end{array}}{\underbrace{[\![\tau_2\ \mathbf{in}_{\mathrm{F}_3}, \mathbf{out}_{\mathrm{F}_2}]\!]_{\mathrm{F}_2}}_{\mathcal{H}_2} \circ \underbrace{[\![\tau_1\ \mathbf{in}_{\mathrm{F}_2}, \sigma\ \mathbf{out}_{\mathrm{F}_0}]\!]_{\mathrm{F}_1}}_{\mathcal{H}_1} = \underbrace{[\![(\tau_1 \circ \tau_2)\ \mathbf{in}_{\mathrm{F}_3}, \sigma\ \mathbf{out}_{\mathrm{F}_0}]\!]_{\mathrm{F}_1}}_{\mathcal{H}_3}}$$



Fusion 前　　Fusion 後

本研究: メタプログラミングへの適用 (3/3)

Abstract Hylomorphism (よい性質を持つ Hylomorphism)

τ と σ が次の型を持つとき，$[\![\tau\ \mathbf{in}, \sigma\ \mathbf{out}]\!]_{\mathrm{F}_1}$ を *Abstract Hylomorphism* と定義する.

$$\begin{aligned} \tau &: \forall A.\, (\mathrm{F}_2\, A \to A) \to (\mathrm{F}_1\, A \to A) \\ \sigma &: \forall A.\, (A \to \mathrm{F}_0\, A) \to (A \to \mathrm{F}_1\, A) \end{aligned}$$

- Abstract Hylomorphism 同士のフュージョンは，Abstract Hylomorphism を出力する (さらなるフュージョン可能)
- フュージョン可能な処理と不可能な処理を明示的に区別することにより，実装を簡素化する.

本研究のフュージョンフレームワーク

1. **Code generator (G):** Anamorphism
2. **Code transaformer (T):** Abstract Hylomorphism
3. **Code analyzer (A):** Catamorphism

$$A \circ \underbrace{T_n \circ \cdots \circ T_2 \circ T_1}_{\text{複数のプログラム変換}} \circ G \quad \Rightarrow \quad \underbrace{\mathcal{H}}_{\text{一つの Hylomorphism に変換する}}$$

実験概要: Number Theoretic Transform

数論変換 (Number Theoretic Transform, NTT)

NTT とは，有限体上の高速フーリエ変換である．

- 多項式乗算を高速化するアルゴリズム
- 耐量子暗号の基盤

本研究の実験

本実験は，フュージョンによる NTT に対する最適化の高速化を実証する．

- **最適化 1．** モジュロ演算の低レベル実装
- **最適化 2．** Lazy Reduction
- **プログラム解析．** 区間解析

Number Theoretic Transform

入力: $a = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{Z}_p^n$

e.g. n = 1024, p = 12289 (NewHope)

n = 256 , p = 3329 (Kyber)

出力: 離散フーリエ変換した a

1: bit_reverse(a)
2: **for** $s = 1$ **to** $\log_2 n$ **do**
3: $m \leftarrow 2^s$
4: $o \leftarrow 2^{s-1} - 1$
5: **for** $k = 0$ **to** $m - 1$ **by** m **do**
6: **for** $j = 0$ **to** $m/2 - 1$ **do**
7: $u \leftarrow a[k + j]$
8: $t \leftarrow (a[k+j+m/2] \times \Omega[o+j]) \mod p$
9: $a[k + j] \leftarrow (u + t) \mod p$
10: $a[k + j + m/2] \leftarrow (u - t) \mod p$
11: **end for**
12: **end for**
13: **end for**

a_i に関するすべての演算は**剰余演算**で行われる.

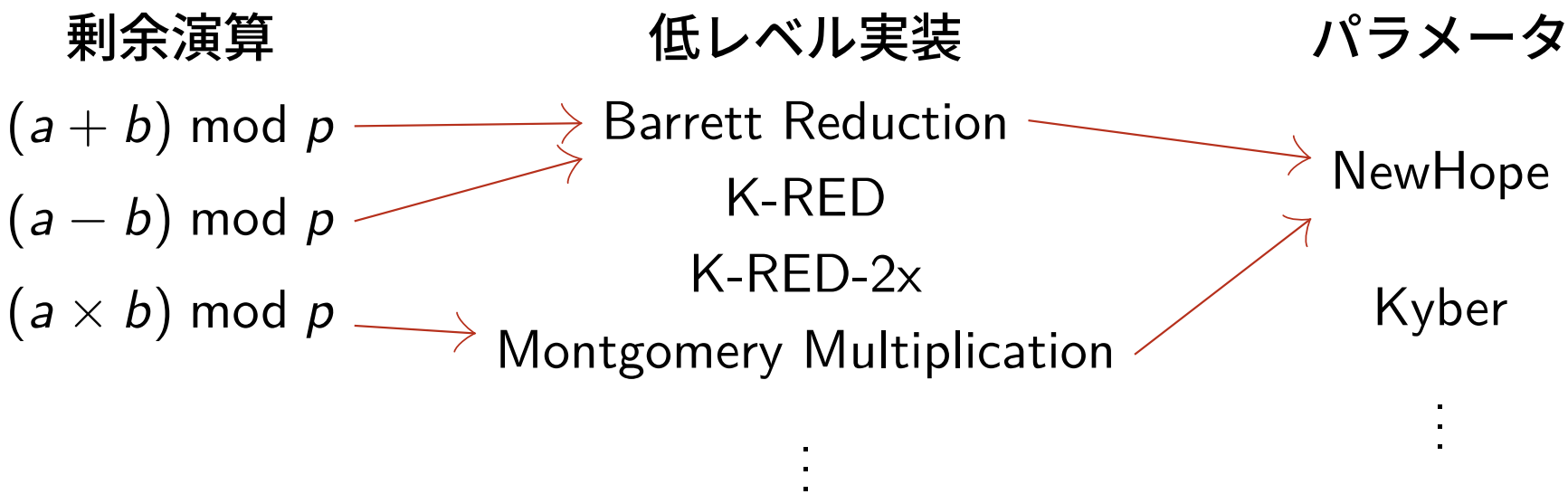
最適化1. モジュロ演算の低レベル実装

- 足し算や掛け算と比べて，割り算は処理に時間がかかる．
- 割り算に依存しないモジュロ演算の実装が望ましい．

Barrett Reduction (NewHope, i.e., $n = 1024, p = 12289$)

16 ビットの符号なし整数 a_i, a_j に対して，

$$a_j \leftarrow a_i \bmod p \;\Leftrightarrow\; \left(\begin{array}{c} u \leftarrow (\texttt{uint32})\; a * 5 >> 16 \\ a_j \leftarrow a_i - (\texttt{uint16})(u * p) \end{array}\right)$$



最適化 2. Lazy Reduction

3-stage reduction [Masuda+2024]

16 ビットの符号なし整数に対して,

```
1:  ...
2:  for s = 1 to 10 do
3:      ...
4:      for k = 0 to m − 1 by m do
5:          for j = 0 to m/2 − 1 do
6:              ...
7:              a[k+j] ← { (u + t) mod p   if s ∈ {3,6,9}
                         { u + t           otherwise
8:              ...
9:          end for
10:     end for
11: end for
```

❶ s はステージと呼ばれる

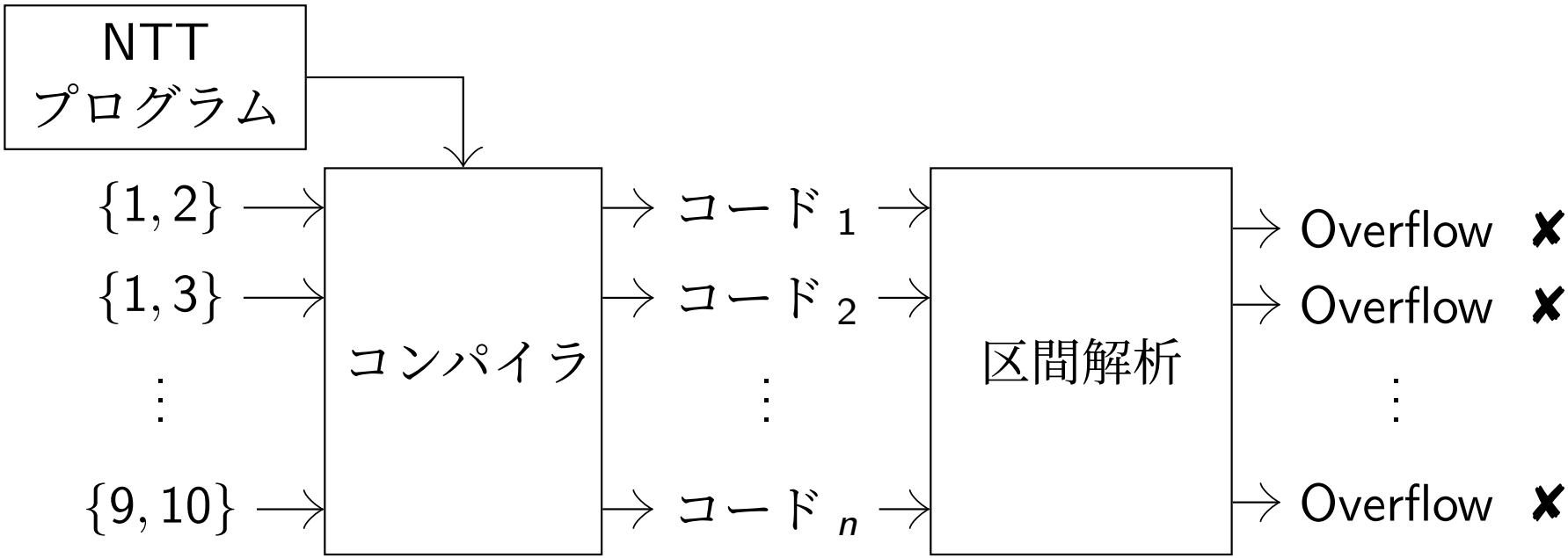
❷ 3 つのステージのみで (低レベルの) 剰余演算を施す

❸ 途中の結果は $[0, p)$ の範囲を超えることを許容するが, $[0, 2^{16})$ の範囲内に収まる必要がある

全探索に基づいた最適化

k-stage の Lazy Reduction で NTT を計算できるか?

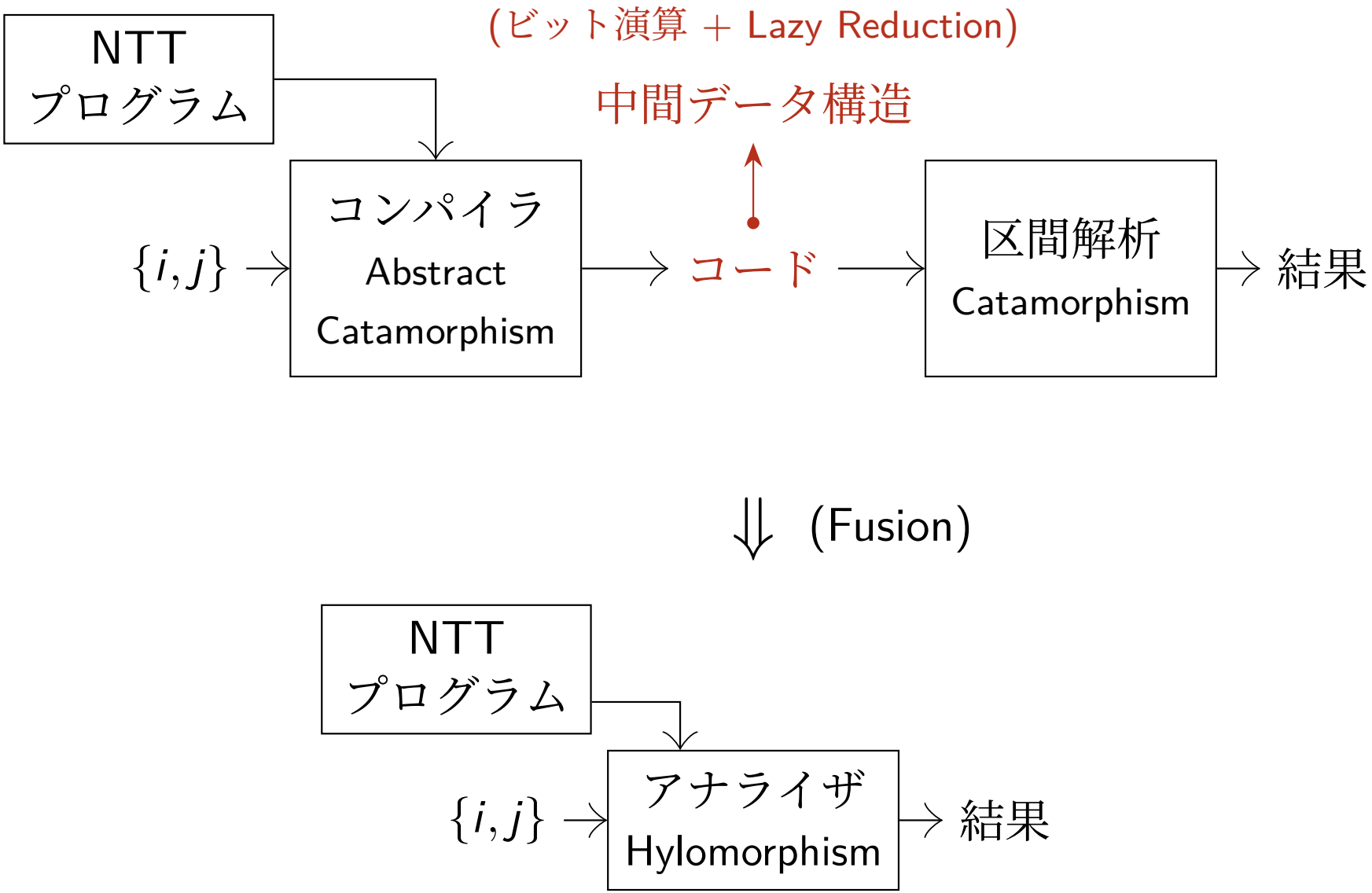
実験内容: k-stage reduction 探索 (例: $k = 2$)



- ステージ i, j $(1 \leq i < j \leq 10)$ で剰余演算を施すコードを生成
- 区間解析で整数オーバーフローを検出

☞ 整数オーバーフローを起こさない Lazy Reduction Stage $\{i, j\}$ を探索する

フュージョン


NTT
プログラム
(ビット演算 + Lazy Reduction)
中間データ構造
コンパイラ
Abstract
Catamorphism
{i, j}
コード
区間解析
Catamorphism
結果
(Fusion)
NTT
プログラム
アナライザ
Hylomorphism
{i, j}
結果

実験結果 (1/2)

フュージョンによる探索の効率向上

Approach	Size	Before Fusion	After Fusion	Speedup
[Masuda+2024]	175	257.16 (s)	18.44 (s)	13.9
[Tokuda+2023]	55	53.32 (s)	3.58 (s)	14.9

探索の結果 (実験の目的ではない)

- 2-stage の組み合わせで生成したすべてのコードにオーバーフローを検出した.
- 3-stage の組み合わせの中に，$\{3, 6, 9\}$ 以外に 9 つのオーバーフローを起こさない組み合わせを発見した.

実験結果 (2/2)

効率向上の考察

- フュージョン前，探索関数は一度低レベルコードの木構造を生成してから解析を行う．
- フュージョン後，探索関数は木構造を生成せずに，**整数演算のみ**で解析を行う．

☞ メモリの割り当てやパターンマッチのオーバーヘッドが効率差の原因となる．

実験の結論

フュージョンは全探索に基づいた最適化の効率を著しく改善できる．

まとめ

1. 本研究は，コード生成器とコードアナライザの合成のためにのフュージョンフレームワークを提案した．
   - PHOAS で中間プログラムを表現する
   - 実装のために Abstract Hylomorphism を提案して，既存のフュージョン規則を改良した．
2. フュージョンフレームワークを実装して，NTT の最適化に対して実験を行った．
   - フュージョンにより，全探索に基づいた最適化の効率が大幅に向上することを実証した．

今後の展望

- Hylomorphism で表現可能なメタプログラミング処理の範囲を解明
- より高度な Recursion Scheme (e.g. `histo` や `futu`) に適用できるフュージョン理論を構築

論文改訂版