Fusion for Metaprogramming: Eliminating Intermediate Programs in Code Generation and Analysis

Zhangfan Li　Yukiyoshi Kameyama

In metaprogramming, developers frequently encounter a dilemma: either creating a monolithic metaprogram that suffers low maintainability, or decoupling functions into separate code generators and analyzers, inefficiently glued by generated programs. In this research, we present a fusion framework for metaprogramming that enables developers to construct intricate metaprograms by composing smaller ones, and automatically eliminates intermediate programs for efficiency. We use Parametric Higher-Order Abstract Syntax (PHOAS) as the representation of programs, which is hygiene yet suitable for fusion. By leveraging the covariant property of PHOAS, we show that existing studies on recursion schemes (Catamorphism, Anamorphism, and Hylomorphism) and generalized fusion (Acid Rain Theorem) can be naturally extended to the new formulation. Finally, we implement the fusion framework in TemplateHaskell, which partially evaluates the result of fusion and generates code with all abstraction overhead eliminated.

1 Introduction

Metaprogramming, and especially multi-stage programming (MSP) [14], has been proved to be a powerful approach for producing software that is maintainable, portable, and performant. By MSP, instead of writing software directly, one writes a program generator in high-level languages that promise maintainability and abstraction, and uses the generator to optimize and output the target program. Since the abstraction and optimization mainly take place in code generators, one can expect both the generated program to be highly specialized, and the source code to be sustainable at the same time. This approach has been widely adopted in various fields, including high-performance computing, cryptography, and image processing [7] [10] [16].

Nevertheless, analysis and verification for the generated program present a significant challenge. A common problem in metaprogramming is analyzing the properties (such as code size or locality in HPC [6]) and guaranteeing the safety (such as the absence of integer overflow in cryptography [9] [10] [16]) of the code produced by code generators. Traditionally, a naive approach to such analysis involves two distinct steps: generating the complete object program, and then performing analyses on the generated code. Although this method works for simple, one-pass analyses, it performs poorly when the generate-and-analyze loop repeats mutiple times. For example, a recent study on cryptography can be interpreted as a generate-and-analysis loop performing an exhaustive search for the optimal program, in which the analysis result was leveraged to guide code generation [9] [10].

In this research, we present a fusion framework for metaprogramming that eliminates all intermediate programs in the generate-and-analyze loop. Firstly, our framework promotes modular metaprogramming, where complex metaprograms are formed from smaller units called Hylomorphisms [11]. Since metaprograms created by our framework are either in a fixed, small enough pattern or composed from those small ones, they are highly reusable and easy to reason about. More-

---

* メタプログラミングのためのフュージョン: コード生成と解析における中間プログラムの除去

李 張帆, 筑波大学, University of Tsukuba.

亀山 幸義, 筑波大学, University of Tsukuba.

over, we adopt parametric higher-order abstract syntax (PHOAS) [1] as the representation of object programs, thereby preventing unintended name capturing and ensuring correctness. Finally, building upon existing studies, we introduce three refined fusion rules to eliminate intermediate programs that arise from the composition of metaprograms. Since those generated programs, which serve to bridge two metaprograms, are not part of the final result, fusion improves the efficiency and yields complex metaprograms of hand-written quality. Our contributions are as follows.

- We presente a hygienic representation of object programs used in metaprogramming, which guarantees the well-scopedness during fusion.
- We show that existing studies on Hylomorphism and fusion can be naturally extended to the new formulation. Furthermore, we refine the fusion rules from previous studies to simplify their implementation in practice.
- We implement the fusion framework in TemplateHaskell [13], which partially evaluates the result of fusion and generates code with all abstraction overhead eliminated. We demonstrate how our framework facilitates metaprogramming using an example of the optimization for the Number Theoretic Transform.

2 Related work

2.1 Recursion schemes

Recursion scheme is a concept that captures the pattern of recursion in recursive functions [19]. Meijer et al. represents one of the earliest systematic investigations into recursion schemes, in which they presented an idea that one should use "structured recursive function" instead of arbitrary recursive function for readability and the opportunity of optimization [11], and the concept of structured recursion eventually evolved into recursion schemes. Hylomorphism, which is known as one of the most expressive recursion schemes [19], was first introduced in that paper [11]. Hu et al. presented an approach to transform almost all recursive functions in practice into Hylomorphism, which highlighted the expressiveness of Hylomorphism [5]. In addition to Hylomorphism, plenty of recursion schemes are proposed and used in practice [19]. This research enables developers to use Hylomorphism as the unit of metaprogram, and construct complex processes from smaller ones without abstraction overheads.

2.2 Fusion

Fusion has been extensively studied for decades. One of the most attractive topics is *stream fusion*, which transforms a chain of processes for linear data (i.e., stream) into a single process for efficiency [8]. Wadler generalized such optimization to arbitrary tree structures (i.e., algebraic data types), which is known as *deforestation* [18], and Gill et al. proposed a practical implementation for deforestation in GHC, a Haskell compiler [4]. Later, Takano and Meijer formalized Gill's implementation using Hylomorphism and proposed a series of rules for fusion, Acid Rain Theorem, which turns out to be a powerful foundation of fusion [15]. Meijer and Hutton attempted to extend the theory on fusion beyond algebraic data types to include functions [12], and Fegaras and Sheard mitigated the limitations of their work by a "trick" [3], but it turns out that they all failed to develop a comprehensive theory for data types embedding arbitrary functions. This research, by contrast, exclusively uses a subset of functions in which the functor's fixed points only occur in positive locations. By restricting the form of functions, we successfully represented lexical scope and enabled fusion.

3 Background

3.1 F-algebra and Algebraic data types

Studies on recursion schemes and fusion are based on the theory of F-algebras in Category Theory. We follow the standard notions and terminology (see, e.g., [11] [15]) and assume that F is an endofunctor on $\mathcal{C}$, where $\mathcal{C}$ is a category which is a complete partial order. Given such a functor F built using `id` (identity), $\underline{A}$ (constant), $(\times)$ (product) and $(+)$ (direct sum), there exists a type T and two strict functions $\mathbf{in}_{\mathrm{F}} : \mathrm{F}\,T \to T$ and $\mathbf{out}_{\mathrm{F}} : T \to \mathrm{F}\,T$ which are each others inverse. The T above, denoted as $\mu\mathrm{F}$, is called the fixed point of F or the algebraic data type defined by F [15]. For example, lists in functional programming can be defined as follows.

$$\mathrm{L}_A = \underline{\mathbf{1}} + \underline{A} \times \mathtt{id}, \quad \mathrm{List}_A = \mu\mathrm{L}_A$$

where **1** denotes the unit type and A is a type constant. The same example can be written in Haskell as follows.

```
data L a x = Nil | Cons a x
newtype Fix f = In { out :: f (Fix f) }
type List a = Fix (L a)
```

where `Fix f` denotes the fixed point of functor `f`. Then a list `[1,2]` can be written as `In (Cons 1 (In (Cons 2 (In Nil))))` in this data type. Although the formulation seems cumbersome at first glance, it proved effective for subsequent discussion on recursion schemes and fusion.

3.2 Recursion schemes and fusion

Catamorphism and *Anamorphism* are the canonical examples of recursion schemes, representing the dual processes of folding (consumption) and unfolding (generation) of recursive data types. Given a functor `F` (e.g. `(L a)` in Section 3.1), they can be defined as follows in Haskell.

```
cata :: (F a -> a) -> Fix F -> a
ana  :: (a -> F a) -> a -> Fix F
cata φ = φ  . fmap (cata φ) . out
ana  ψ = In . fmap (ana  ψ) . ψ
```

Hylomorphism is defined as an Anamorphism followed by a Catamorphism, i.e. $\texttt{cata}\,\varphi \circ \texttt{ana}\,\psi$. An equivalent definition [11] is also frequently used in practice.

```
hylo :: (F b -> b, a -> F a) -> a -> b
hylo (φ, ψ) = φ . fmap (hylo (φ, ψ)) . ψ
```

We also adopt a conventional denotation $[\![\varphi, \psi]\!]_{\mathrm{F}}$ for Hylomorphism, where φ and ψ are the parameter of Catamorphism and Anamorphism, respectively. Notice that $[\![\varphi, \mathbf{out}_{\mathrm{F}}]\!]_{\mathrm{F}}$ is a Catamorphism and $[\![\mathbf{in}_{\mathrm{F}}, \psi]\!]_{\mathrm{F}}$ is an Anamorphism.

One can derive fusion rules for Hylomorphism using the parametricity theorem [17]. One of the pioneering works on this topic is by Takano and Meijer, and their result is known as *Acid Rain Theorem* [15]. Given two functors F_1 and F_2, we show one of the simplified fusion rules as follows.

$$\frac{\tau : \forall A.(\mathrm{F}_2\, A \to A) \to \mathrm{F}_1\, A \to A}{[\![\varphi, \mathbf{out}_{\mathrm{F}_2}]\!]_{\mathrm{F}_2} \circ [\![\tau\ \mathbf{in}_{\mathrm{F}_2}, \psi]\!]_{\mathrm{F}_1} = [\![\tau\ \varphi, \psi]\!]_{\mathrm{F}_1}}$$

That is, given an Hylomorphism h satisfying some conditions, one can fuse a left composition of an arbitrary Catamorphism and h into a single Hylomorphism. A fusion rule for the right composition of an Anamorphism is also presented [15].

4 Fusion for metaprogramming

In this section, we present an approach that uses Hylomorphisms to represent metaprogramming processes and eliminates the compositional overhead through several fusion rules. Given a composition of two metaprograms (e.g., a code generator followed by a analyzer), our method is capable of eliminating the generated program and transforming those processes into a single process. Our method especially benefits the model of generate-and-analyze loop introduced in Section 1, where the pair of generator and analyzer within the loop body is fused into a one-pass process without generating intermediate programs, thereby improving efficiency.

To implement the framework, we first propose a hygienic representation of generated programs used to bridge two metaprograms in Section 4.1. Refining existing studies, we then present three fusion rules to eliminate those object programs that serve as intermediate data structures in Section 4.2. Section 5 demonstrates a concrete example showing how our framework helps the optimization of a cryptography algorithm.

4.1 Hygienic representation of programs

Algebraic data types, the focus of previous studies, are inadequate for representing programs in the metaprogramming context. As explained in Section 3.1, algebraic data types are constructed from product and direct sum and essentially correspond to general tree structures. A naive representation of programs by such tree structures is abstract syntax tree (AST). For example, the term of untyped lambda calculus can be represented by the following data type, where variables are represented by the text of their name (`String`).

```
data ULC x = Var String | Lam String x
           | App x x
```

This representation is awkward in that it is unaware of the lexical scope of variables, thereby causing unintended name capturing. For example, even η-expansion, a simple yet ubiquitous operation in lambda calculus, cannot be soundly implemented as follows.

```
η :: Fix ULC -> Fix ULC
η term =
```

```
In (Lam "x"
  (In (App term (In (Var "x")))))
```

If `term` contains a free variable called the same name `"x"`, the abstraction introduced by η will unintentionally capture it, leading to an incorrect result.

In contrast, this research leverages higher-order functions of the metalanguage to encode the lexical scope of variables. In metaprogramming, metalanguage refers to the programming language that is used to analyze or generate object programs. In the example above, the object language is untyped lambda calculus and the metalanguage is Haskell. By this approach, the term of untyped lambda calculus can be redefined as follows.

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

The new definition differs from the previous one in two key aspects: (1) variables are represented by a parameterized type `v` rather than `String`, and (2) abstraction is encoded using functions in Haskell. The point here is that we take advantage of the lexical scopes of variables in the metalanguage to encode those in the object language. One can simply verify that the η-expansion can be implemented directly and soundly with this data type.

Generally, in addition to product and direct sum used to construct algebraic data types, we also allow function types with a fixed domain to be used in the definition of data types. Just like product and direct sum, it is also a fundamental functor in Haskell, written as `((->) v)`, where `v` is a fixed type. Although arbitrary functions are known to be intractable in the context of fusion [12] [3], which hinders the use of higher-order abstract syntax as a hygienic representation, the restricted form of functions is effective for both scope representation and fusion. Specifically, thanks to the property that fixed points only occur in positive locations, the Hylomorphism and theory for fusion can be directly extended to the new formulation [12]. This approach, also known as parametric higher-order abstract syntax (PHOAS) [1], is fundamentally a variation of HOAS distinguished by its use of a parameterized type for variables.

4.2 Refined fusion rules

Although the fusion rules in previous studies [15] established a foundation for fusion, their implementation remains a significant challenge, often requiring non-trivial compiler modification. Refining the Acid Rain Theorem, this research puts forward a cheap yet effective implementation of the fusion, which enables an automatic fusion.

Figure 1 shows the refined fusion rules used in this research. The central idea is to separate Hylomorphisms suitable for further fusion from those that are not. We call the former *Abstract Hylomorphism* and define it as follows.

Definition 1 (Abstract Hylomorphism) *Let* F_0, F_1 *and* F_2 *be functors,* τ *and* σ *be functions of the following types.*

$$\tau \quad : \quad \forall A.\ (\mathrm{F}_2\ A \to A) \to (\mathrm{F}_1\ A \to A)$$
$$\sigma \quad : \quad \forall A.\ (A \to \mathrm{F}_0\ A) \to (A \to \mathrm{F}_1\ A)$$

Then an Abstract Hylomorphism is a Hylomorphism of the following form.

$$[\![\tau\ \mathbf{in}_{\mathrm{F}_2}, \sigma\ \mathbf{out}_{\mathrm{F}_0}]\!]_{\mathrm{F}_1}$$

We also call it Abstract Catamorphism when $\sigma = \mathrm{id}$*, or Abstract Anamorphism when* $\tau = \mathrm{id}$.

Then, we rephrase the Acid Rain Theorem [15] to distinguish the fusion that keeps the form of Abstract Hylomorphism (AbsHylo-Left and AbsHylo-Right) and that does not (Hylo-Closing). In summary, one can fuse

- A left composition of an Abstract Catamorphism with an Abstract Hylomorphism to get an Abstract Hylomorphism (AbsHylo-Left)
- A right composition of an Abstract Anamorphism with an Abstract Hylomorphism to get an Abstract Hylomorphism (AbsHylo-Right)
- A left composition of an arbitrary Catamorphism as well as a right composition of an arbitrary Anamorphism with an Abstract Hylomorphism to get a Hylomorphism incapable of further fusion (Hylo-Closing)

The proof of these fusion rules is straightforward by Acid Rain Theorem [15]. Moreover, one can simply implement these fusion rules in a practical programming language without modifying compilers or complicated reduction rules [15]. For example, in Haskell, if one defines those abstract recursion schemes as follows,

```
data AbsHylo f2 f1 f0 = AbsHylo
  { τ :: ∀a. (f2 a -> a) -> f1 a -> a
  , σ :: ∀a. (a -> f0 a) -> a -> f1 a }
data AbsCata f2 f1 =
  AbsCata (∀a. (f2 a -> a) -> f1 a -> a)
data AbsAna  f1 f0 =
```

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

$$\frac{\begin{array}{rcl}\sigma_1 & : & \forall A.(A \to \mathrm{F}_0\ A) \to A \to \mathrm{F}_1\ A \\ \sigma_2 & : & \forall A.(A \to \mathrm{F}_1\ A) \to A \to \mathrm{F}_2\ A \\ \tau & : & \forall A.(\mathrm{F}_3\ A \to A) \to \mathrm{F}_2\ A \to A\end{array}}{[\![\tau\ \mathbf{in}_{\mathrm{F}_3}, \sigma_2\ \mathbf{out}_{\mathrm{F}_1}]\!]_{\mathrm{F}_2} \circ [\![\mathbf{in}_{\mathrm{F}_1}, \sigma_1\ \mathbf{out}_{\mathrm{F}_0}]\!]_{\mathrm{F}_1} = [\![\tau\ \mathbf{in}_{\mathrm{F}_3}, (\sigma_2 \circ \sigma_1)\ \mathbf{out}_{\mathrm{F}_0}]\!]_{\mathrm{F}_2}}\ \textsc{AbsHylo-Right}$$

$$\frac{\begin{array}{rcl}\sigma & : & \forall A.(A \to \mathrm{F}_0\ A) \to A \to \mathrm{F}_1\ A \\ \tau & : & \forall A.(\mathrm{F}_2\ A \to A) \to \mathrm{F}_1\ A \to A\end{array}}{[\![\varphi, \mathbf{out}_{\mathrm{F}_2}]\!]_{\mathrm{F}_2} \circ [\![\tau\ \mathbf{in}_{\mathrm{F}_2}, \sigma\ \mathbf{out}_{\mathrm{F}_0}]\!]_{\mathrm{F}_1} \circ [\![\mathbf{in}_{\mathrm{F}_0}, \psi]\!]_{\mathrm{F}_0} = [\![\tau\ \varphi, \sigma\ \psi]\!]_{\mathrm{F}_1}}\ \textsc{Hylo-Closing}$$

Fig. 1 Refined fusion rules

`AbsAna (∀a. (a -> f0 a) -> a -> f1 a)` then fusion rules AbsHylo-Left, AbsHylo-Right and Hylo-Closing can be implemented as follows, respectively.

```
AbsCata τ2 ▷ AbsHylo { τ = τ1, σ } =
  AbsHylo { τ = τ1 . τ2, σ }
AbsHylo { τ, σ = σ2 } ◁ AbsAna σ1 =
  AbsHylo { τ, σ = σ2 . σ1 }
close (φ, AbsHylo { τ, σ }, ψ) =
  [| let h x =
       $(τ φ
         . fmap (\x -> [| (h $x) |])
         . σ ψ $ [|x|])
     in h $a |]
```

Note that `close`, which corresponds to the Hylo-Closing rule, is a staged implementation of Hylomorphism using TemplateHaskell [13] to eliminate abstraction overheads. By combining the hygienic representation introduced in Section 4. 1 with these fusion rules, developers can create complex metaprograms from small, fixed patterns, all without compositional overhead.

5 Example: Lazy Reduction in NTT

In this section, we demonstrate how our framework facilitates the optimization of Number Theoretic Transform (NTT). NTT is an algorithm computing discrete Fourier transform over finite fields. As it is a fundamental algorithm for post-quantum cryptography, researchers continuously work to perform aggressive, low-level optimization on it, where a small performance gain is highly valued. Figure 2 shows the pseudocode for NTT, using

Require: $a = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$,
precomputed constants $\Omega \in \mathbb{Z}_q^n$
Ensure: $a = \mathtt{DFT}(a)$ in the standard order
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: $t \leftarrow (a[k + j + m/2] \times \Omega[o + j]) \mod q$
9: $a[k + j] \leftarrow (u + t) \mod q$
10: $a[k + j + m/2] \leftarrow (u - t) \mod q$
11: **end for**
12: **end for**
13: **end for**

Fig. 2 Cooley-Tukey algorithm for NTT

the standard Cooley-Tukey algorithm [2].

As lines 8, 9, and 10 in Figure 2 show, operations like $a \times b$ are interpreted as modular arithmetic like $(a \times b \mod q)$, where q is the modulus and often a carefully chosen prime number. Instead of applying modular arithmetic to all operations, which leads to poor performance, lazy reduction allows intermediate results to reside outside the range $[0, q)$ and performs modular reduction only when necessary, such as to prevent integer overflow. For example, when $(n, q) = (1024, 12289)$ and the input a is an array of unsigned 16-bit integers, Masuda and Kameyama proposed a lazy-reduction scheme as follows [10].

- Interpret $(a \times b \mod q)$ as `csub(mred(a * b))`
- Interpret $(a + b \mod q)$ as
  - `bred(a + b)`, if $s \in \{3, 6, 9\}$
  - `a + b`, otherwise
- Interpret $(a - b \mod q)$ as `bred(a + 2q - b)`

`csub`, `mred` and `bred` correspond to conditional subtraction, Montgomery reduction, and Barrett reduction, respectively [10], and all of them are highly optimized operations over bits. For example, `bred` in [10] is as follows.

```
uint16_t bred(uint16_t a) {
  uint32_t u = ((uint32_t) a * 5) >> 16;
  return a - (uint16_t)(u * q)
}
```

which has the same effect as modular reduction. Therefore, Masuda's work allows the modular reduction to be delayed unless the index (s) of the outermost loop, called *stage*, is 3, 6, or 9, thereby improving efficiency.

5.1 Analysis for lazy reduction

Given such a scheme of lazy reduction, we need to ask 2 questions.

1. Does overflow occur?
2. Can we delay more `bred` for $a + b \mod q$?

We use a brute-force yet effective method to answer these questions: to answer the first question, we compile the NTT algorithm in Figure 2 to low-level codes optimized by Masuda's scheme, and perform an interval analysis to guarantee the absence of overflow; to answer the second question, we enumerate all possible combinations of 1 and 2 stages, and repeat the previous analysis to find a better optimization. As we compile the high-level code into low-level code, and the low-level code is immediately consumed by the interval analyzer, eliminating the low-level code by fusion would significantly improve the efficiency.

As shown in Figure 3, we use an Abstract Catamorphism followed by a Catamorphism to implement the analysis[†1]. The Abstract Catamorphism compiles high-level code (`HiTerm`), which is a DSL capable of experssing NTT shown in Figure 2, to low-level code (`LoTerm`), which corresponds C language with all loops unrolled and modular operations compiled. The data structures of both high-level code and low-level code are defined in the style explained in Section 4.1, in which the lexical scope of variable are guaranteed by functions in metalanguage, i.e., Haskell. Then we use a Catamorphism to perform the interval analysis over low-level code.

As expected, our framework is capable of fusing the compiler and the analyzer together, resulting in an analyzer that does not really generate code. By fusion rules shown in Figure 1, we can transform the composition of compiler and analyzer into a single Hylomorphism as follows, which is also a Catamorphism.

$$[\![\tau\ \varphi, \mathbf{out}]\!]_{\texttt{HiTerm}}$$

Moreover, we also use TemplateHaskell (i.e., `close` shown in Section 4.2) to partially evaluate the result of fusion, in which all statically known data structures (i.e., `LoTerm`) are eliminated.

The result of analysis shows that Masuda's lazy-reduction stages will not cause integer overflow, and no combinations of 1 or 2 stages can replace Masuda's lazy-reduction stages without overflow, indicating that Masuda's lazy-reduction stages (i.e., 3, 6, and 9) are optimal for this approach.

5.2 The modularity of metaprograms

Thanks to the compositional property, metaprograms in this framework present excellent modularity and can be easily replaced. Consider the following example. Tokuda and Kameyama found an opportunity to slightly improve Masuda's lazy-reduction scheme [16]. Specifically, they

- Interpret $(a - b \mod q)$ as `bred`$(a + q - b)$

while all other arithmetic have the same interpretation as Masuda's work. To redo the analysis in Section 5.1, it turns out that a simple overwriting of the compiler (τ, in Figure 3) is sufficient, where the interval analyzer (φ) are reused. Although such modularity is natural in functional programming, our framework eliminates the abstraction penalty, and thus makes such resource-intensive analysis feasible.

5.3 Evaluation

We evaluated our fusion framework by measuring the speedup of analysis. We use GHC's built-in profiling options to measure the execution time of the analyses described in Section 5.1 and 5.2.

†1 In fact, an additional Anamorphism is also needed to transform code into statical structures due to the usage of TemplateHaskell. We omit the detail in this paper.

$$\underbrace{[\![\varphi, \mathbf{out}]\!]_{\texttt{LoTerm}}}_{\text{Interval Analysis}} \circ \underbrace{[\![\tau\ \mathbf{in}, \mathbf{out}]\!]_{\texttt{HiTerm}}}_{\text{Compilation}}$$

Fig. 3 Overview of the lazy-reduction analyzer

	Before fusion	After fusion	Speedup
MK	257.16 (s)	18.44 (s)	13.9
TK	53.32 (s)	3.58 (s)	14.9

Table 1 Time of analyses

The first experiment involves the composition of generate-analyze for 1, 2 and 3-combinations of stages in totally 10 stages, while the second involves 1 and 2-combinations of stages in totally 10 stages. Therefore, the generate-analysis loop repeats 175 and 55 times, respectively.

Table 1 shows the result, where MK and TK denote two different approaches of lazy reduction described in previous sections [10] [16].Our result indicates that fusion can significantly accelerate the time of analysis. The main cause of the speedup is twofold. Firstly, the fusion is capable of turning the construction of tree structures and pattern matching over tree structures into straightforward integer operations. The absence of memory allocation and condition branch over tree structures (i.e., `LoTerm`) is the driving factor of performance gain. Secondly, both approaches in the experiment [10] [16] use a brote-force analysis in their interval analysis to avoid the precision loss of analysis for bitwise operations like `bred`. This makes the interval analysis time-intensive and amplifies the effect of fusion: compared to pure integer operations after fusion, the original analyzer has to perform pattern matching over tree structures repeatedly. In conclusion, the result shows that our fusion framework makes such analysis-guided optimization more doable at a larger scale.

6 Conclusion

We have presented a fusion framework for metaprogramming. Our framework allows developers to create metaprograms in a small, fixed pattern, called Hylomorphism, and construct complex metaprograms from those units. Extending algebraic data types with a restricted form of functions, we have proposed a representation of object programs that is capable of avoiding unintended name capturing, yet is suitable for fusion. We have refined the existing research on fusion and implemented the framework in TemplateHaskell, which automatically eliminates the abstraction penalty introduced by the compositional style. We have applied our approach to the optimization of NTT. The result indicated that metaprograms in our framework exhibit excellent modularity, and fusion yielded a significant speedup in the time of analysis.

For future work, we plan to extend our framework to object programs defined by Generalized Algebraic Data Types (GADTs). As shown in 4. 1, we assume that all generated programs in the framework are untyped and we defined recursion schemes and fusion based on them. By GADT, we can build a typed object programs and thus obtain stronger guarantee for sound metaprograms. Moreover, we are also interested in advanced recursion schemes in addition to Catamorphism, Anamorphism and Hylomorphism. Our experiment showed that Catamorphisms and Anamorphisms are inadequate for complicated metaprogramming processing, while Hylomorphisms appear cumbersome sometimes. Therefore, a fusion theory for advanced recursion schemes like `histo` and `futu` [19], which is considered too expressive to stream fusion but seems suitable for program manipulation, would further broaden the spectrum of metaprograms we can write in this framework.

Acknowledgements

The authors are supported in part by JSPS Grant-in-Aid for Scientific Research (B) 23K24819.

References

[1] Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics, *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP08, ACM, September 2008, pp. 143–156.

[2] Cormen, T. H.: *Introduction to algorithms*, MIT Press, Cambridge, Massachusetts, third edition edition, 2009. Description based upon print version of record. - IMD-Felder maschinell generiert.

[3] Fegaras, L. and Sheard, T.: Revisiting Cata-

morphisms over Datatypes with Embedded Functions (or, Programs from Outer Space), *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Boehm, H. and Jr., G. L. S.(eds.), ACM Press, 1996, pp. 284–294.

[4] Gill, A., Launchbury, J., and Peyton Jones, S. L.: A short cut to deforestation, *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA93, ACM, July 1993, pp. 223–232.

[5] Hu, Z., Iwasaki, H., and Takeichi, M.: Deriving structural hylomorphisms from recursive definitions, *ACM SIGPLAN Notices*, Vol. 31, No. 6(1996), pp. 73–82.

[6] Kamin, S., Garzarán, M. J., Aktemur, B., Xu, D., Yılmaz, B., and Chen, Z.: Optimization by runtime specialization for sparse matrix-vector multiplication, *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE '14, ACM, September 2014, pp. 93–102.

[7] Kiselyov, O.: *Reconciling Abstraction with High Performance: A MetaOCaml approach*, Foundations and Trends in Programming Languages Ser., No. v.12, Now Publishers, Norwell, MA, 2018. Description based on publisher supplied metadata and other sources.

[8] Kiselyov, O., Biboudis, A., Palladinos, N., and Smaragdakis, Y.: Stream fusion, to completeness, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, ACM, January 2017, pp. 285–299.

[9] Masuda, M. and Kameyama, Y.: *Unified Program Generation and Verification: A Case Study on Number-Theoretic Transform*, Springer International Publishing, 2022, pp. 133–151.

[10] Masuda, M. and Kameyama, Y.: Program generation meets program verification: A case study on number-theoretic transform, *Sci. Comput. Program.*, Vol. 232(2024), pp. 103035.

[11] Meijer, E., Fokkinga, M. M., and Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, Hughes, J.(ed.), Lecture Notes in Computer Science, Vol. 523, Springer, 1991, pp. 124–144.

[12] Meijer, E. and Hutton, G.: Bananas in space: extending fold and unfold to exponential types, *Proceedings of the seventh international conference on Functional programming languages and computer architecture - FPCA '95*, FPCA '95, ACM Press, 1995, pp. 324–333.

[13] Sheard, T. and Jones, S. P.: Template meta-programming for Haskell, *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, Pittsburgh, Pennsylvania, USA, October 3, 2002*, Chakravarty, M. M. T.(ed.), ACM, 2002, pp. 1–16.

[14] Taha, W.: A Gentle Introduction to Multi-stage Programming, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, Lengauer, C., Batory, D. S., Consel, C., and Odersky, M.(eds.), Lecture Notes in Computer Science, Vol. 3016, Springer, 2003, pp. 30–50.

[15] Takano, A. and Meijer, E.: Shortcut deforestation in calculational form, *Proceedings of the seventh international conference on Functional programming languages and computer architecture - FPCA '95*, FPCA '95, ACM Press, 1995, pp. 306–313.

[16] Tokuda, R. and Kameyama, Y.: Generating Programs for Polynomial Multiplication with Correctness Assurance, *Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, POPL '23, ACM, January 2023, pp. 27–40.

[17] Wadler, P.: Theorems for free!, *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, FPCA '89, ACM Press, 1989, pp. 347–359.

[18] Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees, *Theor. Comput. Sci.*, Vol. 73, No. 2(1990), pp. 231–248.

[19] Yang, Z. and Wu, N.: Fantastic Morphisms and Where to Find Them: A Guide to Recursion Schemes, (2022).