# Reconciling Efficiency and Modularity in Metaprogramming via Shortcut Fusion

Zhangfan Li[0009−0007−5474−6261] ✉ and Yukiyoshi Kameyama[0000−0002−2693−5133]

University of Tsukuba, Japan
{zhangfan.li, kameyama}@acm.org

**Abstract.** Code generation is a leading approach to exploit domain- or hardware-specific knowledge. However, it faces an efficiency-modularity dilemma when generated programs must be filtered by a program analyzer. Simply composing a generator with an analyzer leads to an inefficient meta-program, while hand-weaving an analyzer into a generator suffers from poor modularity. In this research, we propose a recipe for modular implementation of the generate-analyze process without heavy performance penalty. Our key idea is to apply *Shortcut Fusion* to meta-programs. Shortcut fusion transforms a composed program of a producer and a consumer of data structures such as lists into a one-pass program. We apply it to a composed meta-program of a code generator and analyzer, to obtain a one-pass meta-program that does not generate code as intermediate data. We introduce four novel fusion rules, relaxing the premise of fusion transformation to widen their applicability in the context of metaprogramming. To validate our proposal, we conducted an experiment on the Number-Theoretic Transform, which has been heavily used in cryptography. Our recipe allows a fully modular implementation of a generator and an analyzer that efficiently tests various optimizations, by which, as a by-product, we discovered a highly efficient code with 50% fewer low-level reductions than the code in previous studies.

**Keywords:** Metaprogramming · Shortcut Fusion · Number-Theoretic Transform · Program Analysis.

## 1 Introduction

Multi-stage programming is a technique that generates highly efficient code by performing optimizations for individual problems based on domain-specific knowledge [11], and has been adopted in various fields including Parsing [30], Stream Processing [12], Automatic Differentiation [29] and Cryptography [15,27].

As multi-stage programming has expanded its application domains, post-analyzing generated code becomes increasingly important. To rule out deficient code, program analyzers and verification tools are frequently post-applied, leading to a performance problem when the size of generated code is huge, or the generate-analyze process needs to be performed repeatedly. For instance,

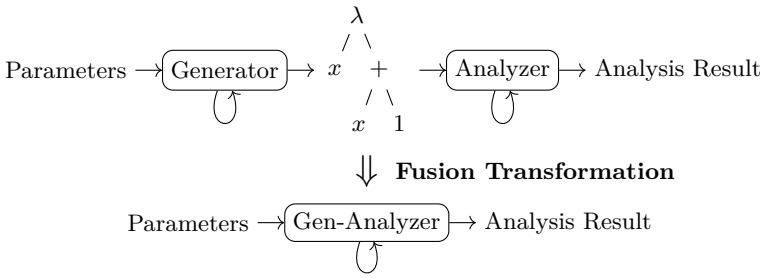


**Fig. 1.** Overview of fusion in the context of metaprogramming

in the Number-Theoretic Transform (NTT), maximizing performance critically depends on aggressively eliminating costly modular reductions[1]. However, eliminating too many reductions incurs runtime errors such as integer overflows. To safely balance this trade-off, interval analysis is used to filter out incorrect programs. Crucially, because the potential combinations for eliminating the modular reductions suffer from a combinatorial explosion, finding the optimal implementation requires executing the generate-analyze process repeatedly over unrolled code spanning tens of thousands of lines — which makes it highly inefficient to post-apply analyses to the fully generated code.

An alternative approach is to hand-weave the program analyzers into the code generators to avoid generating intermediate programs, which, however, suffers from poor modularity. In the case of NTT, some studies manually embed interval analysis into code generators [15,27], resulting in multiple analyses specialized to individual optimizations. This approach presents obvious shortcomings: not only is the work tedious and error-prone, but also the number of manual implementations scales according to the rule of product. Eventually, the poor modularity makes it laborious to verify domain-specific optimizations.

In this paper, we address this efficiency-modularity dilemma. Specifically, we propose a recipe for systematically implementing such generate-analyze processes, and provide an optimization technique to improve the performance of the whole process. We build our solution upon *Shortcut Fusion*[2], a technique that has long been used in data processing but, to the best of our knowledge, has never been systematically investigated in the context of metaprogramming. Specifically, given two independently defined meta-programs — a code producer $f$ and a consumer $g$ — we transform the composition $g \circ f$ into an equivalent meta-program $h$, eliminating the intermediate code produced by $f$ and resulting in a one-pass recursive process. In the special case where $f$ is a code generator and $g$ is a code analyzer, as shown in Figure 1, the transformation yields a process — called "Gen-Analyzer" throughout this paper — that takes the code genera-

[1] Modular reduction refers to the operation computing $a \bmod q$, which is unrelated to *module systems* or *modularity* in programming languages.

[2] Throughout this paper, the term *fusion* refers specifically to *shortcut fusion*.

tor's parameters and directly returns the analysis result of the object program to be generated, completely bypassing the code generation phase.

Applying the fusion theory to metaprogramming needs non-trivial extensions. Firstly, the fusion theory should handle data structures capable of preserving variable bindings correctly (known as *hygiene*), as managing binders manually is notoriously tricky and error-prone. Secondly, the new fusion theory must accommodate a wide range of metaprogramming operations, which are more complex than those typically found in traditional fusion theory, such as list summation. We summarize our contribution as follows.

- We propose a novel, modular approach to the generate-analyze process within the programs-as-data model.
- We provide a theoretical foundation for the fusion transformation to the programs-as-data model, thereby enabling its efficient optimization.
- As a case study of our methodology, we implement a generate-analyze process for Number-Theoretic Transform and show the performance improvements.

The rest of this paper is organized as follows. We review the traditional fusion theory by a simple example and explain the challenges that emerge when applying it to metaprogramming in Section 2. We give our solution for each challenge in Section 3. We validate our solution by a case study in Section 4. We discuss related work in Section 5 and conclude this paper in Section 6.

## 2 Background and Motivation

### 2.1 Traditional Fusion Theory

*Shortcut fusion* is a program transformation that has been extensively investigated for decades [7–9, 20, 26]. It converts the composition of two recursive functions, the first being a producer of a data structure and the second its consumer, into a single recursive function.

As an example, consider the task of summing up the odd elements in a given integer list. A functional programmer divides the task into the selection of odd elements (`filterOdd`) and the summation of the resulting elements (`sum`), then composes them:

$$\texttt{sumOdd} = \texttt{sum} \circ \texttt{filterOdd} \tag{1}$$

This program is easy to implement, yet has a performance problem in that it would create a possibly huge list, then immediately consumes it. Shortcut fusion can convert the composed function into a single recursive function that does not generate intermediate data.

We use Haskell-like syntax in concrete examples. We need three steps to apply the shortcut fusion transformation to the above equation (1). Firstly, we represent the inductive data type list as follows:

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

where the "shape" of the data structure is represented as a *functor*, and the recursive data structure is represented as its fixed point (i.e, `Fix (ListF a)`).

Secondly, we implement the operations to be fused in terms of *recursive schemes* [31]. In this example, we implement `sum` and `filterOdd` in terms of `fold` as follows.

```
sum :: Fix (ListF Int) -> Int
sum = fold(φ) where φ Nil = 0; φ (Cons a b) = a + b

filterOdd :: Fix (ListF Int) -> Fix (ListF Int)
filterOdd = fold(τ In) where
  τ :: ∀x. (ListF Int x -> x) -> ListF Int x -> x
  τ φ Nil = φ Nil
  τ φ (Cons a b) | odd a = φ (Cons a b) | otherwise = b
```

Finally, we apply *fusion rules* to transform the composed functions in (1) into an equivalent recursive function.

$$\begin{aligned} & \texttt{sum} \circ \texttt{filterOdd} \\ = \quad & \{\text{Unfolding the definition}\} \\ & \texttt{fold}(\varphi) \circ \texttt{fold}(\tau\ \texttt{In}) \\ = \quad & \{\text{Applying the fusion rule for folds [26]}\} \\ & \texttt{fold}(\tau\ \varphi) \end{aligned} \tag{2}$$

In the last equation above, two folds are converted into a single fold, where the recursive data structure (i.e., `Fix (ListF Int)`) created by `filterOdd` is eliminated. By inlining the result of fusion, we get a familiar, one-pass definition.

```
sumOdd' (In Nil) = 0
sumOdd' (In (Cons a b))
  | odd a = a + sumOdd' b | otherwise = sumOdd' b
```

This example demonstrates how shortcut fusion allows developers to have their cake and eat it: the source program is composed of decoupled components, enjoying high modularity, and the transformed program is indistinguishable from hand-optimized code.

### 2.2 The Gaps between Fusion Theory and Metaprogramming

Despite its success in data processing, the traditional fusion technique has not been applied to meta-programs — namely, programs that produce and consume other programs. We identify the following three problems that arise when applying existing fusion theory to meta-programs. We will show our solutions to these problems in Section 3.

**Absence of Scope Information** Managing variable bindings poses a significant challenge for traditional fusion theory that targets recursive functions on data types such as trees. Representing object programs by trees and variables by their names results in a subtle problem due to the absence of *scope information*. For each name binding, the scope information indicates the region of

a program in which the name binding is valid. Neglecting scope information in metaprogramming incurs the *name capturing problem* [14], which is notoriously elusive. This necessitates a program representation both capable of maintaining scope information and compatible with the fusion transformation.

**Opacity of Subexpressions** Existing fusion rules require that the recursive functions to be fused must be given by recursion schemes that cannot analyze the substructures of recursive data, making them opaque. We call it the "opacity of subexpressions" problem, which poses a significant restriction on meta-programs. In equation (2), the premise of applying the fusion rule is that `filterOdd` is defined in a special form `fold`($\tau$ `In`), where $\tau$ is required to be a polymorphic function:

$\tau$ `:: ∀x. (ListF Int x -> x) -> ListF Int x -> x`

Due to *parametricity* [28], `filterOdd` cannot analyze the value of `b` in `Cons a b`.

Although sufficient for most data processing [8], this opacity constraint is overly restrictive for meta-programs, which often need to analyze subexpressions. For instance, a meta-program may optimize the expression $a + b$ to $a$ under the condition $b = 0$, which requires the analysis of $b$. This limitation calls for new fusion rules tailored to metaprogramming.

**Integration of Computational Effects** Computational effects play an indispensable role in advanced code generation but complicate the fusion theory. The typical examples of code generation relying on computational effects involve mutable state for *assertion insertion* [13] and delimited continuation for *let-insertion* [10], which performs non-local code generation to generate highly efficient code. However, the presence of unrestricted computational effects obviously contradicts fusion, which shuffles the evaluation order, thereby breaking the equivalence of the transformation. Consequently, special care must be taken to safely accommodate such effectful metaprogramming within a fusion theory.

## 3 Fusion for Metaprogramming

In this section, we first present our solutions to the three problems discussed in the previous section. Subsequently, we propose a recipe for applying fusion to meta-programs consisting of a code generator and analyzer. Finally, we validate its applicability by conducting a case study in Section 4.

### 3.1 Hygienic Representation of Programs

As a solution to the first problem, we incorporate Parametric Higher-Order Abstract Syntax [4] into the data structures to enable hygienic metaprogramming.

Traditional fusion theory is based on *regular data types* [9, 26] defined by *functors* [3] (i.e., *type constructors* in type theory) constructed from identities,

[3] Throughout this paper, *functors* refer to endofunctors on $\mathcal{CPO}$, the category of pointed complete partial orders with continuous functions.

constants, products ($\times$), and sums ($+$). Such a functor F has a solution to the recursive equation $X \cong \mathrm{F}X$, called the *fixed point* of F [26]. We denote it by $\mu\mathrm{F}$, and the morphisms into and out of $\mu\mathrm{F}$ by $\mathbf{in}_{\mathrm{F}} : \mathrm{F}\,\mu\mathrm{F} \to \mu\mathrm{F}$ and $\mathbf{out}_{\mathrm{F}} : \mu\mathrm{F} \to \mathrm{F}\,\mu\mathrm{F}$ (cf. `Fix` in Section 2.1). Many recursive data types, including lists and trees, can be modeled as the fixed points of such functors.

However, as explained in Section 2.2, they are insufficient to model programs, due to the lack of scope information. To represent variable bindings in data structures, we add a new functor

$$\mathrm{F}_A(X) = [A \to X] \tag{3}$$

as an elementary component to construct data types, where $A$ is a fixed object and $[A \to X]$ is the set of continuous functions from $A$ to $X$ in the category $\mathcal{CPO}$. For example, the terms of untyped lambda calculus can be modeled as follows:

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

where we use the new elementary functor in `Lam (v -> x)` to model variable bindings. The fixed point of the functor `ULC v` models the recursive data structure of terms. In this definition, the variables in the object language coincide with variables in the meta-language, i.e., Haskell in this research. Consequently, the scope of variables in the object language is automatically maintained according to the lexical scope in Haskell.

Note that the functor parameter $X$ cannot appear in $A$ in equation (3). This is crucial for further development of fusion theory. It is sufficient to express scope information and guarantees the covariance of the newly introduced functor [18], thereby enabling fusable operations to be well-defined.

### 3.2 Fusion Rules for Pure Meta-Programs

As a solution to the second problem, we introduce two novel fusion rules that extend the traditional fusion rules. They remove the opacity constraint on subexpressions inherent in the traditional rules.

Similar to traditional fusion theory, our fusion rules handle the composition of functions producing or consuming recursive data structures in fixed recursion patterns. Two well-known patterns of recursive functions are *fold* and *unfold* [17]. Given a functor F and a function $\varphi : \mathrm{F}B \to B$, there exists a unique function $\texttt{fold}(\varphi) : \mu\mathrm{F} \to B$ called *fold* or *catamorphism* (cf. `sum` and `filterOdd` in Section 2.1). Dually, given a function $\psi : A \to \mathrm{F}A$, there exists a unique function $\texttt{unfold}(\psi) : A \to \mu\mathrm{F}$ called *unfold* or *anamorphism*. Our fusion rules target functions of a pattern called *hylomorphism*, which generalizes both fold and unfold to capture a wide spectrum of recursive functions [26].

**Definition 1 (Hylomorphism [26]).** *Given types $A$ and $B$, functors $\mathrm{F}_1$ and $\mathrm{F}_2$, and functions $\varphi : \mathrm{F}_2B \to B$, $\eta : \forall X.\ \mathrm{F}_1X \to \mathrm{F}_2X$ and $\psi : A \to \mathrm{F}_1A$, a hylomorphism is a function $[\![\varphi, \eta, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1}$ defined as follows.*

$$[\![\varphi, \eta, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1} : A \to B$$
$$[\![\varphi, \eta, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1} = \mathrm{fix}(\lambda f.\ \varphi \circ \eta \circ \mathrm{F}_1 f \circ \psi)$$

Hylomorphisms generalize catamorphisms and anamorphisms: $\texttt{fold}(\varphi)$ by $[\![\varphi, \mathrm{id}, \mathbf{out}_{\mathrm{F}}]\!]_{\mathrm{F},\mathrm{F}}$ and $\texttt{unfold}(\psi)$ by $[\![\mathbf{in}_{\mathrm{F}}, \mathrm{id}, \psi]\!]_{\mathrm{F},\mathrm{F}}$, where F is the underlying functor for those morphisms. In Definition 1, the second parameter $\eta$ is a *natural transformation* between $\mathrm{F}_1$ and $\mathrm{F}_2$. It allows functions that are both catamorphic and anamorphic to be neutrally represented by $[\![\mathbf{in}_{\mathrm{F}_2}, \eta, \mathbf{out}_{\mathrm{F}_1}]\!]_{\mathrm{F}_2,\mathrm{F}_1}$ [26].

Theorem 1 presents our novel fusion rule that transforms the composition of a lifted catamorphism and a hylomorphism. It is defined as an inference rule, where the conclusion (below the horizontal line) holds if the premise (above the line) is satisfied.

**Theorem 1 (Cata-HyloFusionF).** *Given functors* G *and* $\mathrm{F}_i (i = 1, 2, 3, 4)$,

$$\frac{\tau : \forall X.\ (\mathrm{F}_3 X \to X) \to \mathrm{F}_2(\mathrm{G}X) \to \mathrm{G}X}{\mathrm{G}[\![\varphi, \eta_2, \mathbf{out}_{\mathrm{F}_3}]\!]_{\mathrm{F}_4,\mathrm{F}_3} \circ [\![\tau\ \mathbf{in}_{\mathrm{F}_3}, \eta_1, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1} = [\![\tau(\varphi \circ \eta_2), \eta_1, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1}}$$

Intuitively, given a composition of two functions, if the left one is a lifted catamorphism and the right one is in the form of $[\![\tau\ \mathbf{in}_{\mathrm{F}_3}, \eta_1, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1}$ where $\tau$ is a polymorphic function, then we can transform it into one hylomorphism.

To understand Theorem 1, we compare it with the existing fusion rule, known as the *Acid Rain Theorem* [26], shown below.

$$\frac{\tau : \forall X.\ (\mathrm{F}_3 X \to X) \to \mathrm{F}_2 X \to X}{[\![\varphi, \eta_2, \mathbf{out}_{\mathrm{F}_3}]\!]_{\mathrm{F}_4,\mathrm{F}_3} \circ [\![\tau\ \mathbf{in}_{\mathrm{F}_3}, \eta_1, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1} = [\![\tau(\varphi \circ \eta_2), \eta_1, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1}} \tag{4}$$

As can be seen, the difference lies in the presence of functor G in our rule.

**Acid Rain Theorem [26]:** $\tau : \forall X.\ (\mathrm{F}_3 X \to X) \to \mathrm{F}_2 X \to X$
**Theorem 1:** $\tau : \forall X.\ (\mathrm{F}_3 X \to X) \to \mathrm{F}_2(\mathrm{G}X) \to \mathrm{G}X$

Our fusion rule generalizes it by relaxing the requirement of fusion rules (above the lines in the inference rules), where we allow an additional functor G in the type of $\tau$ to define a fusable hylomorphism. Our rule is consistent with that in the Acid Rain Theorem when G in Theorem 1 is taken as the identity functor.

To understand the effect of G, recall `filterOdd` introduced in Section 2.1, which is defined as $\texttt{fold}(\tau\ \texttt{In})$ and can be fused by the fusion rule in equation (4). As $\tau$ is polymorphic:

```
τ :: ∀x. (ListF Int x -> x) -> ListF Int x -> x
```

one has no knowledge on the sub-list when defining `filterOdd`. By contrast, our fusion in Theorem 1 allows $\tau$ to have type:

```
τ :: ∀x. (ListF Int x -> x) -> ListF Int (G x) -> (G x)
```

where one can convey additional information via the functor G around nodes.

As another example, consider the following arithmetic language that consists of literals and additions, and assume that we intend to optimize the terms by, e.g., transforming $a + 0$ to $a$.

```
data ExprF x = ELit Int | EAdd x x
```

To apply the traditional fusion rule to such an optimization, it must be defined in the form $\texttt{fold}(\tau_1\ \texttt{In})$, where

```
τ1 :: ∀x. (ExprF x -> x) -> ExprF x -> x
τ1 φ (EAdd a b) = ...
```

However, it is impossible, as subexpressions `a` and `b` are of type `x`, which is universally qualified, meaning that we have no knowledge (or no assumption) on them. On the other hand, the new fusion rule allows it to be defined as follows:

```
data ZeroTest x = Unknown x | IsZero
τ2 :: ∀x. (ExprF x -> x) -> ExprF (ZeroTest x) -> (ZeroTest x)
τ2 φ (ELit 0) = IsZero
τ2 φ (EAdd a IsZero) = a
τ2 φ (EAdd (Unknown a) (Unknown b)) = Unknown (φ (EAdd a b))...
```

where we use a functor `ZeroTest` to collect and propagate information throughout terms. $\texttt{fold}(\tau_2\ \texttt{In})$ can be fused with other functions by our fusion rule (as the second hylomorphism in the composition in Theorem 1), where `ZeroTest` corresponds to the newly introduced functor G. We will give more practical instances of the functor G in Section 4.

Dually, there exists a fusion rule for compositions of a hylomorphism and a lifted anamorphism in traditional fusion theory [26]. We generalize it in the same manner as explained above in Theorem 2.

**Theorem 2 (Hylo-AnaFusionF).** *Given functors* G *and* $\mathrm{F}_i (i = 1, 2, 3, 4)$,

$$\frac{\sigma : \forall X.\ (X \to \mathrm{F}_2 X) \to \mathrm{G}X \to \mathrm{F}_3(\mathrm{G}X)}{[\![\varphi, \eta_2, \sigma\ \mathbf{out}_{\mathrm{F}_2}]\!]_{\mathrm{F}_4,\mathrm{F}_3} \circ \mathrm{G}[\![\mathbf{in}_{\mathrm{F}_2}, \eta_1, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1} = [\![\varphi, \eta_2, \sigma(\eta_1 \circ \psi)]\!]_{\mathrm{F}_4,\mathrm{F}_3}}$$

This is the dual of Theorem 1. While Theorem 1 allows us to propagate information while *consuming* data structure, Theorem 2 allows us to propagate information while *producing* data structure. As we will show in Section 4.2, anamorphism naturally expresses *loop unrolling*, and can be fused by this fusion rule. We prove Theorem 1 in Appendix, and Theorem 2 can be proved similarly.

### 3.3 Coping with Effects

As a solution to the third problem, we incorporate *computational effects* into our theory through modeling effects by *monads* and modeling effectful metaprograms by *monadic hylomorphisms*. Previous work [21, 22] developed several fusion rules capable of fusing *monadic* hylomorphisms composed with a *pure* catamorphism or anamorphism. Building upon these studies, we propose two new fusion rules.

**Definition 2 (Monadic Hylomorphism [22]).** *Given a monad* M *defined by a Kleisli triple* $(\mathrm{M}, \eta_A, -^\star)$ *[19], a functor* F*, and functions* $\varphi : \mathrm{F}B \to \mathrm{M}B$*,* $\mathtt{dist}^{\mathrm{M}}_{\mathrm{F}} : \forall X.\ \mathrm{F}(\mathrm{M}X) \to \mathrm{M}(\mathrm{F}X)$ *and* $\psi : A \to \mathrm{M}(\mathrm{F}A)$*, a monadic hylomorphism* $[\![\varphi, \psi]\!]^{\mathrm{M}}_{\mathrm{F}}$ *is defined as follows.*

$$[\![\varphi, \psi]\!]^{\mathrm{M}}_{\mathrm{F}} : A \to \mathrm{M}B$$
$$[\![\varphi, \psi]\!]^{\mathrm{M}}_{\mathrm{F}} = \mathrm{fix}(\lambda f.\ \varphi^\star \circ (\mathtt{dist}^{\mathrm{M}}_{\mathrm{F}} \circ \mathrm{F}f)^\star \circ \psi)$$

The return types of $\varphi$, $\psi$, and the monadic hylomorphism itself are wrapped by a monad M, meaning that they are morphisms in the *Kleisli category* and thus can perform computations with an effect modeled by M in their body.

Previous researchers proposed fusion rules for (1) the composition of a pure catamorphism with a monadic hylomorphism, and (2) the composition of a monadic hylomorphism with a pure anamorphism [21, 22], mirroring the behavior of traditional fusion rules [26]. We generalize their fusion rules in the same manner as described in Section 3.2, allowing a functor G to propagate additional information throughout terms. Given a monad M, functors G and $\mathrm{F}_i (i = 1, 2, 3)$:

**Theorem 3 (Cata-HyloFusionFM).**

$$\frac{\tau : \forall X.\ (\mathrm{F}_2 X \to X) \to \mathrm{F}_1(\mathrm{G}X) \to \mathrm{M}(\mathrm{G}X)}{\mathrm{M}(\mathrm{G}[\![\varphi, \eta, \mathbf{out}_{\mathrm{F}_2}]\!]_{\mathrm{F}_3,\mathrm{F}_2}) \circ [\![\tau\ \mathbf{in}_{\mathrm{F}_2}, \psi]\!]^{\mathrm{M}}_{\mathrm{F}_1} = [\![\tau(\varphi \circ \eta), \psi]\!]^{\mathrm{M}}_{\mathrm{F}_1}}$$

**Theorem 4 (Hylo-AnaFusionFM).**

$$\frac{\sigma : \forall X.\ (X \to \mathrm{F}_2 X) \to \mathrm{G}X \to \mathrm{M}(\mathrm{F}_3(\mathrm{G}X))}{[\![\varphi, \sigma\ \mathbf{out}_{\mathrm{F}_2}]\!]^{\mathrm{M}}_{\mathrm{F}_3} \circ \mathrm{G}[\![\mathbf{in}_{\mathrm{F}_2}, \eta, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1} = [\![\varphi, \sigma(\eta \circ \psi)]\!]^{\mathrm{M}}_{\mathrm{F}_3}}$$

Theorem 3 enables the fusion of the composition of a lifted, pure catamorphism with a monadic hylomorphism. As the catamorphism is pure, the change in evaluation order due to fusion does not affect the result. In the context of metaprogramming, the monadic hylomorphism corresponds to advanced code generators that generate high-quality code relying on effects, and the pure catamorphism implements a local optimization (such as partial evaluation) or an analysis (such as abstract interpretation). Moreover, the information propagation in Section 3.2 is also available here thanks to the presence of G. Theorem 4 handles the dual. These theorems can be proved similarly as Theorem 1.

In Section 4.2, as a case study, we will demonstrate a practical code generator that relies on computational effect, and fuse the effectful code generator with a post-analysis by the fusion rules. Although omitted in this paper, we also realized as monadic hylomorphism the staged monadic combinator for memoization [25], which solves the code duplication problems when staging algorithms such as sparse matrix-vector multiplication and dynamic programming algorithms (generalized Fibonacci function, matrix-chain multiplication and the Floyd-Warshall algorithm). However, applying fusion transformations to these instances requires some non-trivial extensions to our fusion rules, which we leave as a future work.

### 3.4 Metaprogramming via Fusion

Based on our theory developed in the previous sections, we propose a recipe for implementing meta-programs capable of fusion as follows:

- Model the syntax of the object language as a functor (Section 3.1).
- Implement the meta-program based on
  - its recursion pattern (fold or unfold),

  - ability to define it as a polymorphic function as required by the fusion rules, and
  - whether it relies on computational effects.
- Compose the meta-programs together, and rearrange them in accordance with the fusion rules.

We then apply this recipe to a concrete case study in the next section.

## 4 Case Study: Lazy Reduction in NTT

In this section, we demonstrate how the recipe described in Section 3 facilitates the optimization for Number-Theoretic Transform (NTT). Our approach is modular, in that the optimization consists of several independent components, and efficient, since the chain of components is transformed into a single Gen-Analyzer of hand-optimized quality, without overhead caused by intermediate results[4]. We show that fusion improves the execution time of the optimization by a factor of 2.15. Moreover, the modularity without heavy performance penalty enables us to explore more optimization opportunities. As a by-product, we found a new NTT implementation that has 50% fewer *Barrett Reductions* compared to the best previous study [27].

We first review NTT and the *lazy reduction* in Section 4.1. Subsequently, we explain our optimization in Section 4.2. In Section 4.3, we present another NTT implementation inspired by the result of Section 4.2, and show how to reuse the components to verify such an ad hoc optimization.

### 4.1 NTT and Lazy Reduction

$$
\begin{array}{l}
\textbf{let}\ u = \mathtt{A}[i_1]; \\
\textbf{let}\ t = (\mathtt{A}[i_2] \times \Omega[i_3]) \bmod q; \\
\mathtt{A}[i_1] \leftarrow (u + t) \bmod q; \\
\mathtt{A}[i_2] \leftarrow (u - t) \bmod q;
\end{array}
$$

**Fig. 2.** The body of the innermost loop in NTT

NTT is a variant of Fast Fourier Transform (FFT) specialized to a field $\mathbb{Z}/q\mathbb{Z}$ (integers modulo $q$), where $q$ is a prime number. As NTT is heavily used in a certain class of algorithms[5], even a slight improvement in performance can be valuable [15, 27].

The standard NTT algorithm consists of three for-loops, and Figure 2 shows the body of the innermost loop, where $\mathtt{A}$ is the array to be manipulated, $i_1$ and $i_2$ are indices of the array, and $\Omega[i_3]$ is a pre-computed constant [5]. In NTT, each

[4] The source code is available online: https://github.com/dzangfan/RemMvF.

[5] Several post-quantum cryptographic algorithms [1, 24] are based on the Ring Learning with Errors problem, which uses NTT for polynomial multiplication.

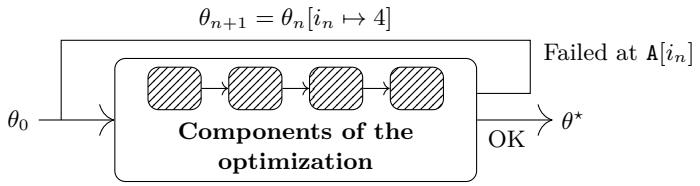


**Fig. 3.** Overview of our search algorithm

arithmetic operation such as addition must be followed by an expensive *modular reduction* that, given $x$, returns $y$ where $x \equiv y \bmod q$. They are the primary target for an optimization called *lazy reduction*; we elide modular reductions by utilizing the fact that data represented by fixed-width integers can be greater than $q$ as long as it does not exceed the maximum integer. This optimization leads to significant performance improvement [1,15,27]. Nevertheless, overly aggressive elimination triggers integer overflows, violating the soundness of optimization.

What further complicates things is that the modular reductions above are often implemented as low-level operations that are highly specialized for $q$. For example, when $q = 12289$ and array cells are 16-bit unsigned integers, the modular reductions for $u+t$ and $u-t$ above are implemented as *Barrett Reduction*s [27]:

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

which returns a 14-bit unsigned integer equal to `x` modulo $q$ (12289). It is much more efficient than the naive modular reduction, but does not guarantee that the result falls within the interval $[0, q)$. Such optimizations interact with lazy reduction and make the correctness of optimization far from trivial.

Consequently, not only is purely theoretical reasoning laborious and error-prone, but every subsequent ad hoc optimization also requires a complete re-evaluation of the correctness. To facilitate such optimization and verification, we propose an approach to mechanically search for efficient and reliable lazy reduction in the next section.

### 4.2 Search for Efficient and Reliable Lazy Reduction

Figure 3 shows the overview of our search algorithm. We generate NTT programs in which modular reductions are aggressively eliminated, and verify the generated program by an *interval analysis* to rule out unsafe programs. We repeat this process until we obtain an efficient and safe result. The strategy of lazy reduction is controlled by $\theta_n$, which will be explained later. The optimization is divided into four separate components defined in folds and unfolds, and we will eventually transform the composition of them into a one-pass Gen-Analyzer by fusion rules proposed in Section 3, with all intermediate data structures eliminated.

We will now explain the four components of our search algorithm in Figure 3: index-unrolling, modular-reduction insertion, low-level compilation, and interval analysis. Throughout the case study, we assume that the size of the input array $n = 1024$ and the modulus $q = 12289$, known as the NewHope protocol [1], and the elements of the array are 16-bit unsigned integers. Our approach can also be applied to other advanced protocols like CRYSTALS-Kyber [2].

**Index-Unrolling (unfold)** As we need to generate straight-line programs to avoid side-channel attack, the first component fully unrolls the three for-loops in NTT. Instead of generating the sequence of computations (Figure 2) directly, we represent it as a sequence of triples $(s, k, j)$, where $s$, $k$ and $j$ are the indices of the three for-loops used in the body. We pass the sequence to the next component to generate NTT code. Concretely, this component receives the initial values of the indices and yields the indices for each iteration of the innermost loop.

$$\texttt{unrolling}\ (1,0,0) = [(1,0,0),(1,2,0),(1,4,0),(1,6,0),(1,8,0),\dots]$$

This can be naturally expressed by unfold.

```
unrolling :: (Int, Int, Int) -> Fix (ListF (Int, Int, Int))
unrolling = unfold(ψ₁) where
  ψ₁ (s, k, j)
    | s <= log₂ n && k <= n - 1 && j <= m / 2 - 1 ❶
      = Cons (s, k, j) (s, k, j + 1)
    | s <= log₂ n && k <= n - 1 = ψ₁ (s, k + m, 0) ❷
    | s <= log₂ n               = ψ₁ (s + 1, 0, 0) ❸
    | otherwise = Nil
    where m = 2^s
```

where $n$ is the size of the input array. We borrow the definition of `ListF` from Section 2.1; `unfold`$(\psi_1)$ denotes an anamorphism, which is a special case of hylomorphism, as explained in Section 3.2. The four branches of $\psi_1$ mean to step the innermost loop (❶), to start a new iteration of the middle loop (❷), to start a new iteration of the outer-most loop (❸), and to halt, respectively.

**Modular-Reduction Insertion (monadic fold)** We instantiate the parameters $i_1$, $i_2$ and $i_3$ in Figure 2 with concrete values computed from each triple $(s, k, j)$ in the sequence to obtain a concrete NTT code.

$$(s,k,j) \mapsto \begin{pmatrix} \textbf{let}\ u = \texttt{A}[k+j]; & \\ \textbf{let}\ t = (\texttt{A}[k+j+m/2] \times \Omega[o+j]) \bmod q; & \\ \texttt{A}[k+j] \leftarrow u + t; & ❹ \\ \texttt{A}[k+j+m/2] \leftarrow u - t; & ❺ \end{pmatrix} \quad \begin{array}{l} \text{where} \\ m = 2^s \\ o\ = 2^{s-1} - 1 \end{array}$$

Unlike the standard implementation [5], we omit all but a few modular reductions in the additions (❹) and subtractions (❺). We will insert modular reductions into some of those operations in this component, and reject unsafe code by an analysis in subsequent components.

To narrow down the intractably large search space ($2^{10,240}$ candidates), we use the following strategy to identify the critical operations for inserting modular

reductions. As $t$ is always in $[0, q)$, and $u - t$ will be eventually compiled to $u + q - t$ to prevent underflow, the value of each cell $\mathtt{A}[i]$ increases relative to $u$ by a maximum of $q$ for each update in $\mathtt{A}[i] \leftarrow u + t$ and $\mathtt{A}[i] \leftarrow u - t$ without modular reduction. This observation allows us to estimate the value of each cell by *the number of times it is updated*. Take $u + t$ for example:

- If $u = \mathtt{A}[i]$ is updated 4 times, $u + t \leq 5q = 61445 < \mathtt{UINT16\_MAX} = 65535$ holds, indicating that integer overflows do not occur.
- If $u = \mathtt{A}[i]$ is updated 5 times, $u + t \leq 6q = 73734 > \mathtt{UINT16\_MAX}$ holds, indicating that integer overflows may occur.

Based on this estimation, we insert modular reductions into ❹ and ❺ to implement lazy reduction. A conservative strategy is to always insert a modular reduction when $\mathtt{A}[i]$ has been updated 4 times. However, since cells in different positions follow different updating patterns, integer overflows do not occur even when we allow some suitable cells to be updated 5 times. We realize the capability of $\mathtt{A}[i]$ to be updated as a *threshold* $\theta : \{0, 1, \ldots, 1023\} \to \{4, 5\}$. Given a set of indices $X \subseteq \{0, 1, \ldots, 1023\}$, where $\mathtt{A}[i]\,(i \in X)$ is supposedly capable of updating 5 times without modular reduction, we define the threshold as follows.

$$\theta(i) = 5 \quad \text{if } i \in X \qquad \text{and} \qquad \theta(i) = 4 \quad \text{otherwise}$$

We implement the modular-reduction insertion as a monadic hylomorphism, in which we use PHOAS to manage the variable bindings ($u$ and $t$), and a *reader monad* to trace the number of times each array cell is updated during code generation. The data structure for the generated high-level code is as follows.

```
data HiCode v x
  = HiLit Int | HiVar v | HiRead Int | HiWrite Int x x
  | HiMul x x | HiAdd x x | HiSub x x | HiMod x
  | HiSkip | HiLet x (v -> x)
```

It is worth noting that `HiCode` is defined in PHOAS style: **let** $u = M; N$ corresponds to `HiLet` $M$ `(\u ->` $N$`)`, by which newly introduced variables are automatically distinguished from others. The sketch of the modular-reduction insertion is as follows, where we count the number of updating for each array cell by a function of type `Int -> Int`, and insert modular reductions (`HiMod`) when the number is about to exceed the threshold ($\theta(i) <$ `count` $+ 1$).

```
type M = Reader (Int -> Int)
insertion :: Fix (ListF (Int, Int, Int)) -> M (Fix (HiCode v))
insertion = ⟦τ₂ In, return ∘ out⟧ᴹ where
 τ₂ :: ∀x. (HiCode v x -> x) -> ListF (Int, Int, Int) x -> M x
 τ₂ φ (Cons (s, k, j) x) = do
  count <- asks ($ k + j)
  if θ (k + j) >= count + 1 then
   return ...φ (HiWrite (k + j) (φ HiAdd...) $
             φ (HiWrite (k + j + m/2) (φ HiSub...) x))
  else
   return ...φ (HiWrite (k + j) (φ (HiMod (φ HiAdd...))) $
             φ (HiWrite (k + j + m/2) (φ (HiMod (φ HiSub...))) x))
```

The update of the counting function, i.e., the increase after skipping modular reductions and reset after applying modular reductions, is performed by a separate function $\texttt{dist}^{\texttt{M}}_{\texttt{L}} : \forall X.\, \texttt{L}(\texttt{M}X) \to \texttt{M}(\texttt{L}X)$, where `L = ListF (Int, Int, Int)` (cf. Definition 2). We omit further implementation details here.

**Low-Level Compilation (fold)** As explained in Section 4.1, modular reductions in the generated code are then compiled to low-level operations specialized to specific moduli. We follow previous studies [15, 27]:

- Implementing modular reductions in $a \times b \bmod q$ by *Montgomery Reduction*.
- Implementing those in $a + b \bmod q$ and $a - b \bmod q$ by *Barrett Reduction*.

While the translation is straightforward, implementing it as a fusable hylomorphism presents a technical challenge: we need to translate modular reductions (`HiMod`) based on the arithmetic type of their operand. As detailed in Section 3.2, such a subexpression-sensitive translation is intractable under traditional fusion rules, but readily handled by ours. Specifically, we use the following functor to distinguish the underlying arithmetic types of modular reductions:

```
data ModShp x = UNK x | MUL x | ADD x | SUB x
```

where `UNK` means unknown type, and others respectively denote multiplication, addition and subtraction. We then translate the high-level modular reductions to low-level operations as follows, where `LoCode` is the data structure of low-level code:

```
compilation :: Fix (HiCode v) -> ModShp (Fix (LoCode v))
compilation = fold(τ_𝒵 In) where
  τ_𝒵 :: ∀x. (LoCode v x -> x) -> HiCode v (ModShp x) -> ModShp x
```

in which the arithmetic types are recorded in each operations (`HiMul`, `HiAdd` and `HiSub`) and utilized in the translation of modular reduction:

```
  τ_𝒵 φ (HiMod (MUL x)) = ...(Montgomery Reduction)...
  τ_𝒵 φ (HiMod (ADD x | SUB x)) = ...(Barrett Reduction)...
```

**Interval Analysis (fold)** We use an interval analysis to detect potential integer overflows in the generated code after lazy reduction and low-level compilation. It differs from the standard interval analysis [6] in that (1) it reports the index of the array cell at which integer overflow occurred, which will be used in the search algorithm (explained below), and (2) it must be combined with a customized code analyzer for some low-level modular reductions[6] [15]. We implement this analysis as $\texttt{fold}(\varphi_4)$:

```
φ_4 :: LoCode Integer IA -> IA
```

[6] This is necessary because the state-of-the-art code analyzer does not take into account a tricky use of bit-level operations used in implementing modular reductions.

where `IA` is an action that, when executed, yields the analysis result. We omit the implementation details here.

**Search Algorithm** Built upon these components, we perform an iterative search to find a threshold $\theta^\star$ that generates an efficient and reliable NTT implementation, as shown in Figure 3. One iteration takes a threshold $\theta_n$ used in modular-reduction insertion, and returns the index of the array cell that may cause an integer overflow, if any, reported by the interval analysis. We start from the most aggressive threshold, $\theta_0(x) = 5$, find the array cell $\texttt{A}[i_0]$ that may cause an integer overflow, and decrease its threshold: $\theta_1 = \theta_0[i_0 \mapsto 4]$. We repeat this process until finding a threshold $\theta^\star$ that does not cause integer overflows.

**Accelerating the Search via Fusion** The search algorithm built from the direct composition of the four components above is inefficient, because not only is the intermediate code massive (tens of thousands of lines), but also the generate-analyze process is performed repeatedly, which amplifies the overhead.

Applying fusion rules proposed in Section 3, we can transform the composition of the four components above into a one-pass Gen-Analyzer as follows, eliminating all intermediately generated programs.

$$
\begin{aligned}
& \underline{\texttt{fmap}(\texttt{fmap}(\texttt{fold}(\varphi_4))) \circ \texttt{fmap}(\texttt{fold}(\tau_3\ \texttt{In}))} \circ [\![\tau_2\ \texttt{In}, \texttt{return} \circ \texttt{out}]\!]^{\texttt{M}} \circ \texttt{unfold}(\psi_1) \\
= & \quad \{\text{Law of functor: } \texttt{fmap}\ f \circ \texttt{fmap}\ g = \texttt{fmap}(f \circ g)\} \\
& \texttt{fmap}(\underline{\texttt{fmap}(\texttt{fold}(\varphi_4)) \circ \texttt{fold}(\tau_3\ \texttt{In})}) \circ [\![\tau_2\ \texttt{In}, \texttt{return} \circ \texttt{out}]\!]^{\texttt{M}} \circ \texttt{unfold}(\psi_1) \\
= & \quad \{\text{Cata-HyloFusionF (Theorem 1)}\} \\
& \underline{\texttt{fmap}(\texttt{fold}(\tau_3\ \varphi_4)) \circ [\![\tau_2\ \texttt{In}, \texttt{return} \circ \texttt{out}]\!]^{\texttt{M}}} \circ \texttt{unfold}(\psi_1) \\
= & \quad \{\text{Cata-HyloFusionFM (Theorem 3)}\} \\
& \underline{[\![\tau_2\ (\tau_3\ \varphi_4), \texttt{return} \circ \texttt{out}]\!]^{\texttt{M}} \circ \texttt{unfold}(\psi_1)} \\
= & \quad \{\text{Hylo-AnaFusionFM (Theorem 4)}\} \\
& [\![\tau_2(\tau_3\ \varphi_4), \texttt{return} \circ \psi_1]\!]^{\texttt{M}}
\end{aligned}
$$

Especially, to fuse the last two components, we instantiate Theorem 1 as follows:

$$
\frac{\tau_3 : \forall X.\ (\texttt{LoCode}_v X \to X) \to \texttt{HiCode}_v(\texttt{ModShp}\ X) \to \texttt{ModShp}\ X}{\texttt{fmap}_{\texttt{ModShp}}(\texttt{fold}(\varphi_4)) \circ \texttt{fold}(\tau_3\ \texttt{In}) = \texttt{fold}(\tau_3\ \varphi_4)}
$$

The presence of `ModShp` in the type of $\tau_3$ violates the requirement in traditional fusion rules [26], which thus fail to fuse $\texttt{fold}(\tau_3\ \texttt{In})$ with other catamorphisms. But our fusion rules handle them properly, where `ModShp` corresponds to the newly introduced functor G in Theorem 1.

We measure the execution time of a single iteration and the speedup in Table 1. The benchmark is evaluated on a Windows 11 Pro with a 2.70 GHz Intel Core i7-1265U CPU and 16GB RAM, running Ubuntu 22.04.5 LTS under WSL 2. It is compiled using GHC 9.4.8 with `-O2` flag. The execution time is evaluated by the `criterion` benchmarking framework with default configurations. The result indicates that our fusion transformations accelerate search iterations by a factor of 2.15.

**Table 1.** Execution times with fusion or without fusion

| With Fusion | Without Fusion | Speedup |
|---|---|---|
| $83.94 \pm 0.32$(s) | $180.43 \pm 6.36$(s) | 2.15 |

1: **for** $j = 0$ **to** $m/2 - 1$ **do**
2: **let** $u = \mathtt{A}[k + j]$;
3: **let** $t = \mathtt{mred}(\mathtt{A}[k + j + m/2] \times \Omega[o + j])$;
4: **if** $(s \bmod 4 = 0) \wedge \underline{(k + j \bmod 2^{s+1} < 2^s)}$ **then**
5: $\mathtt{A}[k + j] \leftarrow \underline{\mathtt{bred}(u + t)}$;
6: **else**
7: $\mathtt{A}[k + j] \leftarrow u + t$;
8: **end if**
9: **if** $(s \bmod 4 = 0) \wedge \underline{(k + j + m/2 \bmod 2^{s+1} < 2^s)}$ **then**
10: $\mathtt{A}[k + j + m/2] \leftarrow \underline{\mathtt{bred}(u - t)}$;
11: **else**
12: $\mathtt{A}[k + j + m/2] \leftarrow u - t$;
13: **end if**
14: **end for**

**Fig. 4.** The innermost loop of our improved NTT implementation

### 4.3 Beyond the Search Results

Along with our development, we have come up with a further improvement for lazy reductions. Thanks to the modularity of our construction, we verified the soundness of the improvement without much effort.

Figure 4 shows the innermost loop of our improved implementation of NTT, with all modifications to the best previous study [27] underlined. The $\mathtt{mred}(x)$ denotes the `csub`-equipped Montgomery Reduction[7], and the $\mathtt{bred}(x)$ denotes the Barrett Reduction [15, 27]. In the new implementation, we select $\mathtt{A}[i]$ that will be used as $u$ in the next iteration of the outer-most loop (by the underlined condition), and only apply Barrett Reduction to them. Further technical details are omitted due to space limitation.

Thanks to the modularity of our approach, we can reuse most of the components in Section 4.2 to verify our new implementation. As shown in Figure 5, by encoding our new implementation as a code generator $\mathtt{fold}(\tau_5\ \mathtt{In})$, we can plug it into the pipeline described in Section 4.2, replacing the modular-reduction insertion component. And, as expected, our fusion rules can eliminate the intermediate results in the new pipeline, yielding a one-pass Gen-Analyzer of hand-optimized quality.

Table 2 shows the final result of our optimization, where our approach significantly reduces the number of Barrett Reductions compared to previous studies [15, 27]. The result shows that, the most conservative threshold ($\theta^\bullet(x) = 4$)

[7] The function `csub` is a straight-line program that conditionally subtracts the modulus $q$ from its input. Concretely, $\mathtt{csub}(x) = x$ if $x < q$ and $\mathtt{csub}(x) = x - q$ otherwise.


**Fig. 5.** The pipeline for verifying our further improved implementation

**Table 2.** Numbers of modular reductions resulted from different strategies

| | **Barrett Reduction** | **Montgomery Reduction** |
|---|---|---|
| Masuda & Kameyama [15] | 6656 | 5120 |
| Tokuda & Kameyama [27] | 2048 | 5120 |
| Ours (Conservative $\theta^\bullet$) | 2048 | 5120 |
| Ours (Aggressive $\theta^\star$) | 1280 | 5120 |
| Ours (Figure 4) | 1024 | 5120 |

results in the same number of Barrett Reductions as that of the best previous study [27], the result of our aggressive search algorithm $\theta^\star$ outperforms it, and our new algorithm gives a drastic improvement.

## 5 Related Work

Shortcut fusion was initially introduced by Gill et al. as a lightweight yet effective method to fuse folds (i.e., folds or catamorphisms) in Haskell [8]. Later, Takano and Meijer generalized shortcut fusion to hylomorphisms, which provides a unified fusion framework for folds and unfolds over arbitrary tree structures [26]. Takano and Meijer's work sparked a series of studies aimed at implementing the fusion framework as a general optimization [7, 9, 20]. Beyond the standard hylomorphism, researchers have devoted significant efforts to extend it to other settings, such as allowing arbitrary functions in data types [18] and supporting monadic effects [21, 22]. Our research expands the scope of shortcut fusion to the domain of metaprogramming.

Our generalization of fusion rules has been inspired by tagless-final style, which is a methodology that offers a statically typed, highly modular approach to defining domain-specific languages [3]. Although the tagless-final style was not originally designed for fusion, there is a close affinity between the tagless-final style and shortcut fusion for folds [8], which is also mentioned by Matsuda et al. [16]. This affinity inspired us to generalize the fusion rules in Section 3.2; the introduction of the functor G in Theorem 1, in particular, enables the fusion of meta-programs that is valid in tagless-final style but invalid in traditional shortcut fusion theory. In contrast to the tagless-final style, we base our approach on hylomorphisms, taking into account the dual of folds and the monadic effects necessary for advanced code generation.

Petrashko et al. implemented "miniphases" in Scala's compiler, which fuse separately defined program transformations into a single traversal. Their work

is not based on recursion schemes, such as hylomorphism, and failed to guarantee that "fusing phases does not change their behavior" [23]. By contrast, the soundness of our framework can be formally proved by the free theorem [28].

## 6 Conclusion

We have proposed a methodology for modularly yet efficiently providing correctness guarantees for the generated code in metaprogramming. We achieved this by fusing independently defined code generators and analyzers into a one-pass Gen-Analyzer, completely eliminating the intermediate code. We validated our proposal by a case study, where our approach enabled the modular generation of a highly optimized, verified implementation of NTT.

For future work, we plan to make use of this methodology and clarify the boundaries of its applicability in metaprogramming. Based on the result, we are interested in generalizing our work to other advanced recursion schemes that are more suitable for metaprogramming.

**Acknowledgments.** We would like to thank Akimasa Morihata for his valuable comments on related work. We are grateful to the anonymous reviewers for their insightful feedback and constructive suggestions. This work was supported by JST SPRING, Japan Grant Number JPMJSP2124.

## Appendix

*Proof (Theorem 1).* Given a function $g : \forall A.\ (\mathrm{F}A \to A) \to B \to \mathrm{G}A$, one can derive the following theorem (the free theorem [28]) based on its parametricity.

$$f \circ x = x' \circ \mathrm{F}f \quad \Rightarrow \quad \mathrm{G}f \circ g\ x = g\ x' \tag{5}$$

where $f$ needs to be strict, i.e., $f(\bot) = \bot$, if arbitrary recursion is allowed. Instantiating $f$ to $[\![\varphi, \eta_2, \mathbf{out}_{\mathrm{F}_3}]\!]_{\mathrm{F}_4,\mathrm{F}_3}$, $g$ to $\lambda x.\ [\![\tau\ x, \eta_1, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1}$, $x$ to $\mathbf{in}_{\mathrm{F}_3}$, $x'$ to $\varphi \circ \eta_2$ and F to $\mathrm{F}_3$, $f$ is strict due to the strictness of $\mathbf{out}_{\mathrm{F}_3}$, and the following equation holds by the fixed point equation of hylomorphism's definition.

$$\begin{aligned} &[\![\varphi, \eta_2, \mathbf{out}_{\mathrm{F}_3}]\!]_{\mathrm{F}_4,\mathrm{F}_3} \circ \mathbf{in}_{\mathrm{F}_3} \\ &= \varphi \circ \eta_2 \circ \mathrm{F}_3[\![\varphi, \eta_2, \mathbf{out}_{\mathrm{F}_3}]\!]_{\mathrm{F}_4,\mathrm{F}_3} \circ \mathbf{out}_{\mathrm{F}_3} \circ \mathbf{in}_{\mathrm{F}_3} \\ &= (\varphi \circ \eta_2) \circ \mathrm{F}_3[\![\varphi, \eta_2, \mathbf{out}_{\mathrm{F}_3}]\!]_{\mathrm{F}_4,\mathrm{F}_3} \end{aligned} \tag{6}$$

meaning that the premise in equation (5) holds under this assignment. Consequently, the conclusion in equation (5) holds as follows,

$$\mathrm{G}[\![\varphi, \eta_2, \mathbf{out}_{\mathrm{F}_3}]\!]_{\mathrm{F}_4,\mathrm{F}_3} \circ [\![\tau\ \mathbf{in}_{\mathrm{F}_3}, \eta_1, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1} = [\![\tau(\varphi \circ \eta_2), \eta_1, \psi]\!]_{\mathrm{F}_2,\mathrm{F}_1} \tag{7}$$

which is precisely the desired conclusion of the fusion rule. □

Theorems 2, 3 and 4 can be similarly proved by the parametricity of $g : \forall A.\ (A \to \mathrm{F}A) \to \mathrm{G}A \to B$, $g : \forall A.\ (\mathrm{F}A \to A) \to B \to \mathrm{M}(\mathrm{G}A)$, and $g : \forall A.\ (A \to \mathrm{F}A) \to \mathrm{G}A \to B$, respectively. We omit those proofs due to space limitation.

## References


1. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum Key Exchange—A New Hope. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 327–343. USENIX Association, Austin, TX (Aug 2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim
2. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018. pp. 353–367. IEEE (2018). https://doi.org/10.1109/EuroSP.2018.00032, https://doi.org/10.1109/EuroSP.2018.00032
3. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming **19**(5), 509–543 (2009). https://doi.org/10.1017/S0956796809007205
4. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. pp. 143–156. ICFP '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1411204.1411226, https://doi.org/10.1145/1411204.1411226
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009), http://mitpress.mit.edu/books/introduction-algorithms
6. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021), https://mitpress.mit.edu/9780262044905/principles-of-abstract-interpretation/
7. Domínguez, F., Pardo, A.: Exploiting algebra/coalgebra duality for program fusion extensions. In: Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications. LDTA '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1988783.1988789, https://doi.org/10.1145/1988783.1988789
8. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture. pp. 223–232. FPCA '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/165180.165214, https://doi.org/10.1145/165180.165214
9. Johann, P., Visser, E.: Warm fusion in Stratego: A case study in generation of program transformation systems. Annals of Mathematics and Artificial Intelligence **29**(1-4), 1–34 (Feb 2000). https://doi.org/10.1023/A:1018956702672, https://link.springer.com/10.1023/A:1018956702672
10. Kameyama, Y., Kiselyov, O., Shan, C.c.: Shifting the stage – Staging with delimited control. J. Funct. Program. **21**(6), 617–662 (2011). https://doi.org/10.1017/S0956796811000256, https://doi.org/10.1017/S0956796811000256
11. Kiselyov, O.: Reconciling Abstraction with High Performance: A MetaOCaml approach. Found. Trends Program. Lang. **5**(1), 1–101 (2018). https://doi.org/10.1561/2500000038, https://doi.org/10.1561/2500000038
12. Kiselyov, O., Biboudis, A., Palladinos, N., Smaragdakis, Y.: Stream fusion, to completeness. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 285–299. POPL '17, Association for Computing

Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009880, https://doi.org/10.1145/3009837.3009880

13. Kiselyov, O., Kameyama, Y., Sudo, Y.: Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In: Igarashi, A. (ed.) Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings. pp. 271–291. Lecture Notes in Computer Science (2016). https://doi.org/10.1007/978-3-319-47958-3_15, https://doi.org/10.1007/978-3-319-47958-3_15
14. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming. pp. 151–161. LFP '86, Association for Computing Machinery, New York, NY, USA (1986). https://doi.org/10.1145/319838.319859, https://doi.org/10.1145/319838.319859
15. Masuda, M., Kameyama, Y.: Program generation meets program verification: A case study on number-theoretic transform. Sci. Comput. Program. **232**, 103035 (2024). https://doi.org/10.1016/J.SCICO.2023.103035, https://doi.org/10.1016/j.scico.2023.103035
16. Matsuda, K., Frohlich, S., Wang, M., Wu, N.: Embedding by Unembedding. Proc. ACM Program. Lang. **7**(ICFP) (Aug 2023). https://doi.org/10.1145/3607830, https://doi.org/10.1145/3607830
17. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture. pp. 124–144. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
18. Meijer, E., Hutton, G.: Bananas in space: extending fold and unfold to exponential types. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture. pp. 324–333. FPCA '95, Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/224164.224225, https://doi.org/10.1145/224164.224225
19. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55–92 (1991). https://doi.org/https://doi.org/10.1016/0890-5401(91)90052-4, https://www.sciencedirect.com/science/article/pii/0890540191900524
20. Onoue, Y., Hu, Z., Takeichi, M., Iwasaki, H.: A calculational fusion system HYLO. In: Proceedings of the IFIP TC 2 WG 2.1 International Workshop on Algorithmic Languages and Calculi. pp. 76–106. Chapman & Hall, Ltd., GBR (1997)
21. Pardo, A.: Fusion of recursive programs with computational effects. Theoretical Computer Science **260**(1), 165–207 (2001). https://doi.org/https://doi.org/10.1016/S0304-3975(00)00127-4, https://www.sciencedirect.com/science/article/pii/S0304397500001274
22. Pardo, A.: Combining Datatypes and Effects. In: Vene, V., Uustalu, T. (eds.) Advanced Functional Programming. pp. 171–209. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
23. Petrashko, D., Lhoták, O., Odersky, M.: Miniphases: compilation using modular and efficient tree transformations. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 201–216. PLDI 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062346, https://doi.org/10.1145/3062341.3062346

24. Seiler, G.: Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. IACR Cryptol. ePrint Arch. **2018**, 39 (2018), http://eprint.iacr.org/2018/039
25. Swadi, K.N., Taha, W., Kiselyov, O., Pasalic, E.: A monadic approach for avoiding code duplication when staging memoized functions. In: Hatcliff, J., Tip, F. (eds.) Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006. pp. 160–169. ACM (2006). https://doi.org/10.1145/1111542.1111570, https://doi.org/10.1145/1111542.1111570
26. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture. pp. 306–313. FPCA '95, Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/224164.224221, https://doi.org/10.1145/224164.224221
27. Tokuda, R., Kameyama, Y.: Generating Programs for Polynomial Multiplication with Correctness Assurance. In: Brady, E.C., Palsberg, J. (eds.) Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2023, Boston, MA, USA, January 16-17, 2023. pp. 27–40. ACM (2023). https://doi.org/10.1145/3571786.3573017, https://doi.org/10.1145/3571786.3573017
28. Wadler, P.: Theorems for free! In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. pp. 347–359. FPCA '89, Association for Computing Machinery, New York, NY, USA (1989). https://doi.org/10.1145/99370.99404, https://doi.org/10.1145/99370.99404
29. Wang, F., Zheng, D., Decker, J., Wu, X., Essertel, G.M., Rompf, T.: Demystifying differentiable programming: shift/reset the penultimate backpropagator. Proc. ACM Program. Lang. **3**(ICFP) (Jul 2019). https://doi.org/10.1145/3341700, https://doi.org/10.1145/3341700
30. Willis, J., Wu, N., Pickering, M.: Staged selective parser combinators. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020). https://doi.org/10.1145/3409002, https://doi.org/10.1145/3409002
31. Yang, Z., Wu, N.: Fantastic Morphisms and Where to Find Them: A Guide to Recursion Schemes (Jun 2022). https://doi.org/10.48550/arXiv.2202.13633, http://arxiv.org/abs/2202.13633, arXiv:2202.13633 [cs.PL]